Slot Games

A quantitative model of computation

Dan R. Ghica
Oxford University Computing Laboratory

ABSTRACT

We present a games-based denotational semantics for a quantitative analysis of programming languages. We define a Hyland-Ong-style games framework called *slot games*, which consists of HO games augmented with a new action called token. We develop a slot-game model for the language Idealised Concurrent Algol by instrumenting the strategies in its HO game model with token actions. We show that the slot-game model is a denotational semantics induced by a notion of observation formalised in the operational theory of improvement of Sands, and we give a full abstraction result. A quantitative analysis of programs has many potential applications, from compiler optimisations to resource-constrained execution and static performance profiling. We illustrate several such applications with putative examples that would be nevertheless difficult, if not impossible, to handle using known operational techniques.

Categories and Subject Descriptors: F.3.2 [Semantics of Programming Languages]: Denotational semantics; C.4 [Performance of Systems]: Modeling techniques.

General Terms: Languages, Performance, Theory, Verification

Keywords: Game semantics, quantitative analysis, Algol

1. INTRODUCTION

1.1 Intension and extension in programming semantics

One of the essential requirements of a Tarski-style semantics is that two phrases should have the same *denotation* if and only if they can be replaced in any context without changing the overall meaning of the resulting phrase [34]. In programming languages this requirement is called *full abstraction*, and is interpreted as follows: two terms should have the same denotation if and only if they are *observationally equivalent.*

The notion of observational equivalence is defined with respect to a set of term-rewriting rules which are called an *evaluation relation* or an *operational semantics*, and which represent a definition of the language. Observational equivalence is defined as follows: for any *program* with a hole $\mathcal{C}[-]$, $\mathcal{C}[M_1]$ terminates if and only if $\mathcal{C}[M_2]$ also terminates. By *program* we understand a command-type closed term and by *termination* we understand the existence of a reduction path to the (unique) command-type constant.

The pioneering work of Scott [32], Plotkin [26], Milner [20] and others on defining and studying the full abstraction problem for the programming language PCF is an important milestone in the history of theoretical computer science. This problem turned out to be hard, remaining unsolved until recently [22, 13, 2, 24].

The full-abstraction problem, as originally formulated, is relative to a particular definition of *observability*: the contexts in which observations are to be made are *programming language contexts*, and the property being observed is *termination*. Although observation in such contexts induces an important notion of equivalence, it is by no means a definitive one. For example, equivalence in programming language contexts does not entail equivalence in all *logical* contexts, as defined by some specification or verification logic. The reason is that programming logics, as opposed to languages, can be more fine-grained in what they can distinguish. For example, Reynolds's specification logic of Idealised Algol [27] involves a predicate called *non-interference* which distinguishes between terms otherwise observationally indistinguishable in the language [7]. In the *separation logic* developed by O'Hearn, Reynolds and others [23] to reason about programs with dynamically allocated linked structures, there exist garbage-sensitive formulations of the logic, although the existence of garbage is not observable in programming language contexts.

It is obvious that the denotational model induced by a programming logic can be substantially different from that induced by the programming language alone. However, most of the semantic study has been focused on denotational models relative to programming-language contexts only. But it is equally interesting to find correspondences between various observational contexts and the denotational semantics they induce. For instance, much of the study of the process calculus CSP has been undertaken along these lines. An entire hierarchy of denotations has been developed, depending on the *specification languages* one uses to analyse CSP processes. [28]

It is customary to think of a denotational semantics as an extensional model of the language. However, the consid-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
POPL'05, January 12–14, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-830-X/05/0001 ...$5.00.

erations above suggest that the line between *extension* and *intension* in programming is quite blurry. In other words, the issue of *what* a program does is sometimes essentially related to *how* it does it, depending on the context in which the question is asked.

The subtle interplay between extension and intension in programming languages, relative to programming language contexts, was studied by Berry and Curien [4, etc.] and by Brookes and Geva [6]. The ideas of Berry and Curien had significant impact on the subsequent development of Game Semantics.

1.2 Quantitative modeling

The distinction between intension and extension becomes even more difficult when the context of the analysis involves a quantitative assessment of resource usage, such as time, memory, or power. There exist important situations where such an analysis is required: compiler optimisation, real-time computation, embedded devices. In such situations a meaningful notion of equivalence must reflect, in some well-defined way, the *efficiency* of a program (or program fragment) because resource-usage becomes observable.

For a quantitative analysis of computational phenomena, the appropriate notion of observation is formally described in Sands's operational theory of improvement [30, 31, 21], which was motivated by the need to study program transformations in various λ-calculi. In a program transformation two properties are important to prove: that the transformed version of a program is more efficient and that, otherwise, there are no behavioural differences between the two programs. In higher-order functional programs using call-by-name (or call-by-need) these two tasks are not trivial.

In this paper we will examine a cost-sensitive denotational semantics induced by the operational theory of improvement in the same way that a "standard" denotational semantics is induced by an operational semantics. To formulate the semantic model we use *slot games*, a version of Hyland-Ong-style game semantics [13], augmented with a novel type of action called *token*. The game model will be shown to be *fully-abstract* relative to operational improvement.

Incorporating highly intensional concepts such as costs in a denotational semantics is quite difficult (see Sec. 1.3 below). However, adding costs to a game model requires little revision of the game-theoretic apparatus. The game model will be also seen to be *flexible* and *robust* in the following sense: for any given programming language there is practically an endless variety of quantitative analyses, depending on the usage of what resource one needs to observe and on how the various operations of the language consume that resource. This also depends on the concrete execution model of the architecture on which a program is run. It would be utterly impractical to have a model which is not uniformly parameterised by such factors.

The motivation for having a *denotational* counterpart for the theory of improvement is the standard one. Operational theories are good tools for *defining* languages or properties of languages, but they can be awkward in *reasoning* about them. Using operational techniques it is especially difficult to reason about *open terms*, i.e. programs that use non-locally defined identifiers. Moreover, a denotational semantics is *compositional*, allowing analyses about program fragments to be combined directly into an analysis of a larger program.

In order to apply operational techniques to open terms the language must satisfy so-called *context lemmas*. The λ-calculi studied by Sands satisfy such context lemmas, because they are functionally pure. However, languages with imperative constructs are known in general not to satisfy strong context lemmas.

We emphasise the advantage of having a denotational theory of improvement by analysing a language which combines higher-order procedures (call-by-name), local state and concurrency: Idealised Concurrent Algol (ICA). We do not know if the language has contextual properties strong enough to allow a meaningful operational theory of improvement. However, using the denotational model we can produce such analyses. We conclude by showing some applications of slot games, including automated quantitative analyses of resource-usage using model checking. Except for function inlining the examples we consider would be difficult, if not impossible, to prove using any known operational approaches because they involve open programs that are structurally very different. However, semantic reasoning offers rather straightforward answers.

1.3 Related work

Some of the earliest work on quantitative analysis using semantics was that of Bjerner, Holmström [5] and Wadler [36], for analysing first-order lazy languages. David Sands's doctoral dissertation [29] extended the techniques to languages with higher-order functions. This approach was improved and extended to lazy data structures by Van Stone in her dissertation [33].

A semantic-based analysis of *complexity* (rather than *costs*) was undertaken by Doug Gurr in his dissertation [11], but his work has a different emphasis. Gurr proposes a category-theoretical framework based on monadic constructions in order to allow comparison of complexity of programs even if they are written in different programming languages. Programs and complexity properties are interpreted in different categories, with a functor assigning complexity to meanings of programs. A monadic calculus for costing programs with concurrency and arrays is also used by [14].

Our work is innovative in defining a notion of full abstraction for quantitative analyses through a connection with Sands's notion of improvement. Using game semantics we then give a denotational model that is fully abstract.[1]

2. ICA AND ITS GAME MODEL

Idealised Concurrent Algol (ICA) is Idealised Algol extended with parallel composition ($||$) and binary semaphores. Semaphores can be manipulated using two (blocking) primitives, $\mathbf{grb}(-)$ and $\mathbf{rls}(-)$, which grab and respectively release a semaphore.

The combination of call-by-name and fine-grained concurrency forms a programming idiom in which programming efficiently is a nontrivial exercise. This justifies the need for a denotational framework that can support a quantitative analysis of computation.

The semantics is defined using a (small-step) transition relation $\Sigma \vdash M, s \longrightarrow M', s'$. Σ is a set of names of variables denoting *memory cells* and semaphores denoting *locks*;

[1]One of the referees pointed out recent (unpublished) independent work of Benjamin Leperchey [18] which studies a similar game model and the categorical structures that arise from it.

$$\Sigma \vdash \mathbf{skip} \,||\, \mathbf{skip},\ s \longrightarrow \mathbf{skip},\ s$$
$$\Sigma \vdash \mathbf{skip}; c,\ s \longrightarrow c,\ s$$
$$\Sigma \vdash \mathbf{newvar}\, x := n\, \mathbf{in}\, c,\ s \longrightarrow c,\ s$$
$$\Sigma \vdash \mathbf{newsem}\, x := n\, \mathbf{in}\, c,\ s \longrightarrow c,\ s$$
$$\Sigma \vdash \{ n_1 \} \star n_2,\ s \longrightarrow n,\ s, \quad \text{where } n = \{ n_1 \} \star n_2$$
$$\Sigma \vdash \mathbf{if}\, 0\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2,\ s \longrightarrow M_1,\ s$$
$$\Sigma \vdash \mathbf{if}\, n\, \mathbf{then}\, M_1\, \mathbf{else}\, M_2,\ s \longrightarrow M_2,\ s, \quad \text{where } n \neq 0$$
$$\Sigma \vdash !v,\ s \otimes (v \mapsto n) \longrightarrow n,\ s \otimes (v \mapsto n)$$
$$\Sigma \vdash v := n',\ s \otimes (v \mapsto n) \longrightarrow \mathbf{skip},\ s \otimes (v \mapsto n')$$
$$\Sigma \vdash \mathbf{grb}(v),\ s \otimes (v \mapsto 0) \longrightarrow \mathbf{skip},\ s \otimes (v \mapsto 1)$$
$$\Sigma \vdash \mathbf{rls}(v),\ s \otimes (v \mapsto n) \longrightarrow \mathbf{skip},\ s \otimes (v \mapsto 0), \quad \text{where } n \neq 0$$
$$\Sigma \vdash (\lambda x.M)M',\ s \longrightarrow M[M'/x],\ s$$
$$\Sigma \vdash \mathbf{fix}\, M,\ s \longrightarrow M(\mathbf{fix}\, M),\ s$$

Figure 1: Basic reduction rules for ICA

s, s' are states, i.e. functions $s, s' : \Sigma \to \mathbb{N}$, and M, M' are terms. c stands for any language constant (n or **skip**) and $\star$ for any binary operator (arithmetic-logic or sequential composition). The basic rules are given in Fig. 1. The rules for local variable and semaphore are:

$$\frac{\Sigma, v \vdash C[v/x], s \otimes (v \mapsto n) \longrightarrow C', s' \otimes (v \mapsto n')}{\Sigma \vdash \mathbf{newvar}\, x := n\, \mathbf{in}\, C, s \longrightarrow \mathbf{newvar}\, x := n'\, \mathbf{in}\, C'[x/v], s'}$$

$$\frac{\Sigma, v \vdash C[v/x], s \otimes (v \mapsto n) \longrightarrow C', s' \otimes (v \mapsto n')}{\Sigma \vdash \mathbf{newsem}\, x := n\, \mathbf{in}\, C, s \longrightarrow \mathbf{newsem}\, x := n'\, \mathbf{in}\, C'[x/v], s'}$$

The in-context reduction rules are given by

$$\frac{\Sigma \vdash M,\ s \longrightarrow M',\ s'}{\Sigma \vdash \mathcal{E}[M],\ s \longrightarrow \mathcal{E}[M'],\ s'},$$

where the evaluation contexts are defined by the grammar

$$\begin{aligned}\mathcal{E} ::= &[-] \mid C \,||\, \mathcal{E} \mid \mathcal{E} \,||\, C \mid \mathcal{E} \star M \mid c \star \mathcal{E} \\ &\mid \mathbf{if}\, \mathcal{E}\, \mathbf{then}\, M\, \mathbf{else}\, N \mid !\mathcal{E} \mid \mathcal{E}M \mid \mathcal{E} := n \mid V := \mathcal{E} \\ &\mid \mathbf{grb}(\mathcal{E}) \mid \mathbf{rls}(\mathcal{E}).\end{aligned}$$

Other constructs (**while**, **for**, **let**, **or** etc.) can be easily introduced as syntactic sugar.

We consider an *angelic* notion of termination: we say that a term M terminates in state s, written $M, s \Downarrow$, if there exists a terminating evaluation at start state s: $\exists s',\ M, s \longrightarrow^* c, s'$, with $c \in \mathbb{N} \cup \{\mathbf{skip}\}$. If M is closed and $M, \emptyset \Downarrow$ we write $M \Downarrow$. We define the *observational approximation* relation contextually: $\Gamma \vdash M_1 \sqsubsetapprox M_2$ by $\forall \mathcal{C}[-] : \mathbf{com}, \mathcal{C}[M_1] \Downarrow$ implies $\mathcal{C}[M_2] \Downarrow$, where $\mathcal{C}[M_i]$ are closed programs of **com** type. *Observational may-equivalence* ($\Gamma \vdash M_1 \cong M_2$) is defined as $\Gamma \vdash M_1 \sqsubsetapprox M_2$ and $\Gamma \vdash M_2 \sqsubsetapprox M_1$.

In [10] we have given a game model which is fully abstract for $\sqsubsetapprox$ and $\cong$. Below we give a sketch of the model.

An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$ where M_A is a set of *moves*, $\lambda_A : M_A \to \{ O, P \} \times \{ Q, A \}$ is a function determining for each $m \in M_A$ whether it is an *Opponent* or a *Proponent move*, and a *question* or an *answer*. We write $\lambda_A^{OP}, \lambda_A^{QA}$ for the composite of λ_A with respectively the first and second projections. $\vdash_A$ is a binary relation on M_A, called *enabling*, satisfying: if $m \vdash_A n$ for no m then $\lambda_A(n) = (O, Q)$, if $m \vdash_A n$ then $\lambda_A^{OP}(m) \neq \lambda_A^{OP}(n)$, and if $m \vdash_A n$ then $\lambda_A^{QA}(m) = Q$. If $m \vdash_A n$ we say that m *enables* n. We shall write I_A for the set of all moves of A which have no enabler; such moves are called *initial*. Note that an initial move must be an Opponent question.

The *product* ($A \times B$) and *arrow* ($A \Rightarrow B$) arenas are defined by:

$$\begin{aligned} M_{A\times B} &= M_A + M_B \\ \lambda_{A\times B} &= [\lambda_A, \lambda_B] \\ \vdash_{A\times B} &= \vdash_A + \vdash_B \end{aligned}$$

$$\begin{aligned} M_{A\Rightarrow B} &= M_A + M_B \\ \lambda_{A\Rightarrow B} &= [\langle \lambda_A^{PO}, \lambda_A^{QA} \rangle, \lambda_B] \\ \vdash_{A\Rightarrow B} &= \vdash_A + \vdash_B + \{ (b, a) \mid b \in I_B \text{ and } a \in I_A \} \end{aligned}$$

where $\lambda_A^{PO}(m) = O$ iff $\lambda_A^{OP}(m) = P$.

An arena is called *flat* if its questions are all initial (consequently the P-moves can only be answers). In arenas used to interpret base types all questions are initial and P-moves answer them as detailed in the table below.

Arena	O-question	P-answers
$[\![\mathbf{com}]\!]$	*run*	*ok*
$[\![\mathbf{exp}]\!]$	q	n
$[\![\mathbf{var}]\!]$	*read*	n
	write(n)	*ok*
$[\![\mathbf{sem}]\!]$	*grb*	*ok*
	rls	*ok*

A *justified sequence* in arena A is a finite sequence of moves of A equipped with pointers. The first move is initial and has no pointer, but each subsequent move n must have a unique pointer to an earlier occurrence of a move m such that $m \vdash_A n$. We say that n is (explicitly) justified by m or, when n is an answer, that n answers m. Note that interleavings of several justified sequences may not be justified sequences; instead we shall call them *shuffled sequences*. If a question does not have an answer in a justified sequence, we say that it is *pending* (or *open*) in that sequence. In what follows we use the letters q and a to refer to question- and answer-moves respectively, m denotes arbitrary moves and m_A a move from M_A.

In order to constitute a legal play, a justified sequence must satisfy a well-formedness condition which reflects the "static" style of concurrency of our programming language: any process starting sub-processes must wait for the children to terminate in order to continue. In game terms, if a question is answered then that question and all questions justified by it must have been answered (exactly once).

Formally, the definition is:

DEFINITION 1. *The set P_A of* positions *(or* plays*) over A consists of the justified sequences s over A which satisfy the two conditions below.*

FORK *: In any prefix $s' = \cdots\ q \cdots\ m$ of s, the question q must be pending before m is played.*

WAIT *: In any prefix $s' = \cdots\ q \cdots\ a$ of s, all questions justified by q must be answered.*

For two shuffled sequences s_1 and s_2, $s_1 \amalg s_2$ denotes the set of all interleavings of s_1 and s_2. For two sets of shuffled sequences S_1 and S_2, $S_1 \amalg S_2 = \bigcup_{s_1 \in S_1, s_2 \in S_2} s_1 \amalg s_2$. Given a set X of shuffled sequences, we define $X^0 = X$, $X^{i+1} = X^i \amalg X$. Then $X^{\circledast}$, called *iterated shuffle* of X, is defined to be $\bigcup_{i \in \mathbb{N}} X^i$.

We denote O-moves by o. A strategy σ is *O-complete* if $s \in \sigma$ and $so \in P_A$ imply $so \in \sigma$.

Definition 2. *A strategy σ on A (written $\sigma : A$) is a prefix-closed O-complete subset of P_A.*

Strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ are composed in the standard way, by considering all possible interactions of positions from τ with shuffled sequences of $\sigma^{\circledast}$ in the shared arena B then hiding the B moves.

More formally, let u be a sequence of moves from arenas A, B and C with justification pointers from all moves except those initial in C such that pointers from moves in C cannot point to moves in A and vice versa. Define $u \upharpoonright B, C$ to be the subsequence of u consisting of all moves from B and C (pointers between A-moves and B-moves are ignored). $u \upharpoonright A, B$ is defined analogously (pointers between B and C are then ignored). We say that u is an *interaction sequence* of A, B and C if $u \upharpoonright A, B \in P^{\circledast}_{A \Rightarrow B}$ and $u \upharpoonright B, C \in P_{B \Rightarrow C}$. The set of all such sequences is written as $int(A, B, C)$. Then the interaction sequence $\sigma \lightning \tau$ of σ and τ is defined by

$$\sigma \lightning \tau = \{\, u \in int(A, B, C) \mid u \upharpoonright A, B \in \sigma^{\circledast},\ u \upharpoonright B, C \in \tau \,\}$$

Suppose $u \in int(A, B, C)$. Define $u \upharpoonright A, C$ to be the subsequence of u consisting of all moves from A and C only together with new pointers from all moves $m_A \in M_A$ to any (initial) moves $m_C \in M_C$ such that there exist (initial) moves $m_B \in M_B$ with pointers from m_A to m_B and from m_B to m_C:

$$\cdots \quad m_A \;\cdots\; m_B \;\cdots\; m_C \quad \cdots \upharpoonright A, C$$
$$= \cdots \quad m_A \;\cdots\; m_C \quad \cdots$$

Then the composite strategy $\sigma;\tau$ is defined to be

$$\sigma;\tau = \{\, u \upharpoonright A, C \mid u \in \sigma \lightning \tau \,\}.$$

The model consists of *saturated* strategies only: the saturation condition stipulates that all possible (sequential) observations of (parallel) interactions must be present in a strategy: actions of the environment can always be observed earlier if possible, actions of the program can always be observed later. To formalise this, for any arena A a preorder $\preceq$ on P_A is defined as the least transitive relation $\preceq$ satisfying $s_0 \cdot o \cdot s_1 \cdot s_2 \preceq s_0 \cdot s_1 \cdot o \cdot s_2$ and $s_0 \cdot s_1 \cdot p \cdot s_2 \preceq s_0 \cdot p \cdot s_1 \cdot s_2$ for all s_0, s_1, s_2 where o is an O-move and p is a P-move. In the above pairs of positions moves on the lhs of $\preceq$ have the same justifier as on the rhs.

The two saturation conditions, in various formulations, have a long pedigree in the semantics of concurrency. For example, they have been used by Udding to describe propagation of signals across wires in delay-insensitive circuits [35] and by Josephs *et al* to specify the relationship between input and output in asynchronous systems with channels [15]. Laird has been the first to adopt them in game semantics, in his model of Idealised CSP [17].

Definition 3. *A strategy σ is* saturated *iff $s \in \sigma$ and $s' \preceq s$ imply $s' \in \sigma$.*

Arenas and saturated strategies form a Cartesian closed category $\mathcal{G}_{\text{sat}}$ in which $\mathcal{G}_{\text{sat}}(A, B)$ consists of saturated strategies on $A \Rightarrow B$. The identity strategy is defined by "saturating" the alternating positions $s \in P_{A_1 \Rightarrow A_2}$ such that $\forall\ t \sqsubseteq_{\text{even}} s,\ \ t \upharpoonright A_1 = t \upharpoonright A_2$, which gives rise to the behaviour of an unbounded buffer.

Let us denote by $\mathsf{strat}(P)$ the least saturated strategy generated by a set of plays P, i.e. its saturated set of O-complete prefixes.

The constants of ICA are interpreted by:

$$\llbracket -;- : \mathbf{com} \to \theta_0 \to \theta_1 \rrbracket$$
$$= \mathsf{op}_{;} = \mathsf{strat}(q_1\, run\, ok\, q_0\, a_0\, a_1)$$
$$\llbracket - \,||\, - : \mathbf{com}_0 \to \mathbf{com}_1 \to \mathbf{com}_2 \rrbracket$$
$$= \mathsf{op}_{||} = \mathsf{strat}(run_2\, run_0\, run_1\, ok_0\, ok_1\, ok_2)$$
$$\llbracket - := - : \mathbf{var} \to \mathbf{exp} \to \mathbf{com} \rrbracket$$
$$= \mathsf{op}_{:=} = \mathsf{strat}(run\, q\, n\, write(n)\, ok\, ok)$$
$$\llbracket !- : \mathbf{var} \to \mathbf{exp} \rrbracket = \mathsf{op}_{!} = \mathsf{strat}(q\, read\, n\, n)$$
$$\llbracket \mathbf{grb} : \mathbf{sem} \to \mathbf{com} \rrbracket$$
$$= \mathsf{op}_{\mathbf{grb}} = \mathsf{strat}(run\, grb\, ok\, ok)$$
$$\llbracket \mathbf{rls} : \mathbf{sem} \to \mathbf{com} \rrbracket$$
$$= \mathsf{op}_{\mathbf{rls}} = \mathsf{strat}(run\, rls\, ok\, ok)$$

Stateful behaviour is induced by local variable and semaphore binding:

$$\llbracket \mathbf{newvar}\, x := n \rrbracket = \mathsf{cell}_n$$
$$= \mathsf{strat}\Big(q\, q\, (read\, n)^*\, (\sum_{i=0}^{N} (write(i)\, ok\, (read\, i)^*))^*\, a\, a\Big)$$
$$\llbracket \mathbf{newsem}\, x := 0 \rrbracket = \mathsf{lock}_0$$
$$= \mathsf{strat}\Big(q\, q\, (grb\, ok\, rls\, ok)^*\, (grb\, ok + \epsilon)\, a\, a\Big)$$
$$\llbracket \mathbf{newsem}\, x := 1 \rrbracket = \mathsf{lock}_1$$
$$= \mathsf{strat}\Big(q\, q\, (rls\, ok\, grb\, ok)^*\, (rls\, ok + \epsilon)\, a\, a\Big).$$

This is the standard way of introducing stateful behaviour in game models [3]. Free variables of type **var** are in general not constrained to "stateful" behaviour, i.e. the last variable written is not necessarily the next one read. Only when a variable of this type is bound to **new** is such behaviour enforced, through composition with a strategy that behaves like a memory cell. For semaphores, the proper "locking" behaviour consists of strictly alternating *grab* and *release* operations.

As shown in [10], $\mathcal{G}_{\text{sat}}$ is fully abstract for $\cong$ in the sense mentioned below. Let $\mathsf{comp}(\sigma)$ be the set of non-empty complete plays of a strategy σ.

Theorem 1 (Full abstraction). $\Gamma \vdash M_1 \sqsubsim M_2 \iff \mathsf{comp}(\llbracket \Gamma \vdash M_1 \rrbracket) \subseteq \mathsf{comp}(\llbracket \Gamma \vdash M_2 \rrbracket)$.

3. SLOT GAMES

The game model presented in the previous section is fully abstract relative to a notion of observability defined by (angelic) termination in programming-language contexts. All other behaviour is abstracted away. For example, there exist only two identifiable behaviours for closed terms of **com**mand type: the terminating command and the non-terminating command. As a consequence, all sorting programs have exactly the same semantic model (insofar as they

are correct), irrespective of how efficient or inefficient they are.

For a quantitative account of the behaviour we must introduce a new observable in the model, with the property that, unlike moves, this new observable *cannot be hidden* through composition or otherwise. This is essential if we wish to model the consumption of a resource which is not re-usable, such as time or power, and suggests that resource consumption should not be modeled by *moves*. Moves can always be hidden through composition and they should be used only to model the "input-output" interactive behaviour of the program.

We therefore expand the game paradigm by introducing a new kind of action, called *token-action*, and denoted by the reserved symbol $\textcircled{\$}$. It represents the intuition that, in order to continue playing, one of the protagonists needs to make a *payment*. The analogy is borrowed from slot machines: at various points in the game one is required to insert tokens in order to be allowed to continue playing. We denote sequences of n token-actions by $\textcircled{n}$.

The game apparatus introduced so far does not have to change too much in order to accommodate token-actions. We only need to introduce the new concept of strategy-with-costs:

Definition 4. *A* strategy-with-costs σ *on an arena* A *is a prefix-closed O-complete set of sequences such that there exists an* A*-strategy* $\sigma^\dagger$ *such that* $\forall s \in \sigma,\ s \upharpoonright M_A \in \sigma^\dagger$*.*

To wit, a strategy-with-costs is simply a strategy instrumented with the distinguished token actions $\textcircled{\$}$.

We call $\sigma^\dagger$ the *underlying* strategy of σ; we also call $s^\dagger = s \upharpoonright M_A$ the underlying position (or play) of s. We call $|s| = n$, where $s \upharpoonright \textcircled{\$} = \textcircled{n}$ the *cost* of s.

Sequences in strategies-with-costs, which we shall call *plays-with-costs*, therefore are interleavings of the form $s \amalg \textcircled{n}$, where s is a play, and are interpreted as "play s has cost n". Note that plays-with-costs give information not only about what the overall costs are, but also about the moments when costs are incurred, relative to moves.

Strategies-with-costs $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$ are composed in the standard way, by considering all possible interactions of plays-with-costs from τ with shuffled plays-with-costs of $\sigma^{\circledast}$ in the shared arena B then hiding the B moves. Notice that token actions, not being moves, do not participate in the composition. Also, notice that although the B moves are hidden, no tokens are ever hidden. Therefore tokens can only accumulate through composition.

Example 1. *Consider strategies-with-costs* $\sigma : \mathbf{com}_1 \Rightarrow (\mathbf{exp}_2 \Rightarrow \mathbf{exp}_3)$ *and* $\tau : (\mathbf{exp}_2 \Rightarrow \mathbf{exp}_3) \Rightarrow (\mathbf{exp}_4 \Rightarrow \mathbf{exp}_5)$*:*

$$\sigma = \mathsf{strat}(q_3\ run_1\ \textcircled{2}\ ok_1\ q_2\ \textcircled{5}\ 1_2\ 3_3)$$
$$\tau = \mathsf{strat}(q_5\ q_3\ q_2\ q_4\ 1_4\ \textcircled{6}\ 1_2\ 3_3\ \textcircled{2}\ 3_5)$$

Such plays may occur for example in computing terms which approximate $p : \mathbf{com} \vdash \lambda x.p; x + 2 : \mathbf{exp} \to \mathbf{exp}$ *and* $f : \mathbf{exp} \to \mathbf{exp}, x : \mathbf{exp} \vdash f(x) : \mathbf{exp}$*, respectively. The cost of* $\textcircled{2}$ *in* σ *can be interpreted as a cost incurred by the non-local command* p*, and so on.*

The interaction sequence of the two strategies has the following complete play-with-costs

$$q_5\ q_3\ run_1\ \textcircled{2}\ ok_1\ q_2\ \textcircled{5}\ q_4\ 1_4\ \textcircled{6}\ 1_2\ 3_3\ \textcircled{2}\ 3_5$$

After hiding the moves from the shared component $\mathbf{exp}_2 \Rightarrow \mathbf{exp}_3$ *we are left with*

$$\sigma;\tau = \mathsf{strat}(q_5\ run_1\ \textcircled{2}\ ok_1\ \textcircled{5}\ q_4\ 1_4\ \textcircled{6}\ \textcircled{2}\ 3_5)$$
$$= \mathsf{strat}(q_5\ run_1\ \textcircled{2}\ ok_1\ \textcircled{5}\ q_4\ 1_4\ \textcircled{8}\ 3_5)$$

3.1 A case study

In this section we informally present a *particular* cost-analysis of ICA, in order to illustrate the model of the previous section.

As mentioned in the introduction, for any given language there are many quantitative analyses that make sense. They are relative to the resource we wish to track and the operational specifics of the architecture on which a program is executed. Suppose we wish to analyse ICA in an environment where there is significant overhead in the following operations: assignment and dereferencing (because of memory access), parallel composition (because of process management) and semaphore manipulation (because of lock management). We do this by modeling these operations using strategies-with-costs with the sets of complete plays-with-costs of the following shapes. We use $(\!| - |\!)$ to denote interpretation in the slot-game model.

$$(\!| := |\!) = \mathsf{strat}(run_2\ \textcircled{\$}\ q_1\ n_1\ write(n)_0\ ok_0\ ok_2)$$
$$(\!| ! |\!) = \mathsf{strat}(q\ \textcircled{\$}\ read\ n\ n)$$
$$(\!| \,||\, |\!) = \mathsf{strat}(run_2\ \textcircled{4}\ run_0\ run_1\ ok_0\ ok_1\ ok_2)$$
$$(\!| \mathbf{grb} |\!) = \mathsf{strat}(run_1\ \textcircled{2}\ grb_0\ ok_0\ ok_1)$$
$$(\!| \mathbf{rls} |\!) = \mathsf{strat}(run_1\ \textcircled{2}\ rls_0\ ok_0\ ok_1).$$

For all other constructs, $(\!| - |\!) = [\![-]\!]$.

In our particular cost model, the overhead of memory access is the unit, that of parallel composition is 4 units and that of semaphore manipulation is 2 units.

Using this cost model we can achieve precise quantitative comparisons of programs. Consider for example the two programs:

$$M_1 \equiv\ test : \mathbf{exp}_1, proc : \mathbf{exp}_2 \to \mathbf{com}_3 \vdash$$
$$\mathbf{new}\ x := 0\ \mathbf{in}\ \mathbf{while}\ test\ \mathbf{do}\ x := 1;\ proc(!x) : \mathbf{com}_4$$

and

$$M_2 \equiv test : \mathbf{exp}_1, proc : \mathbf{exp}_2 \to \mathbf{com}_3 \vdash$$
$$\mathbf{while}\ test\ \mathbf{do}\ proc(1) : \mathbf{com}_4.$$

Using the ICA game model we can show they are observationally equivalent. Using the cost model we can (informally) see that the former is less efficient that the latter. The first program is modeled by:

$$(\!| M_1 |\!) = \mathsf{strat}\{run_4\ (q_1\ n_1\ \textcircled{\$}\ run_3\ q_2^l\ \textcircled{l}\ 1_2^l\ ok_3)^k\ q_1\ 0_1\ ok_4 \mid k, l, n \in \mathbb{N},\ n \neq 0\},$$

The second program is modeled by:

$$(\!| M_2 |\!) = \mathsf{strat}\{run_4\ (q_1\ n_1\ run_3\ q_2^l\ 1_2^l\ ok_3)^k\ q_1\ 0_1\ ok_4 \mid k, l, n \in \mathbb{N},\ n \neq 0\}.$$

This means that for any fixed k and l, the second program will save $k\times(l+1)$ units. Informally, k represents the number of iterations of the while-loop and l the number of times the procedure uses its argument, and both are controlled by the

$$\Sigma \vdash \mathbf{skip} \,||\, \mathbf{skip},\ s \longrightarrow^{k_{\text{par}}} \mathbf{skip},\ s$$
$$\Sigma \vdash \mathbf{skip}; c,\ s \longrightarrow^{k_{\text{seq}}} c,\ s$$
$$\Sigma \vdash \mathbf{newvar}\, x := n \,\mathbf{in}\, c,\ s \longrightarrow^{k_{\text{var}}} c,\ s$$
$$\Sigma \vdash \mathbf{newsem}\, x := n \,\mathbf{in}\, c,\ s \longrightarrow^{k_{\text{sem}}} c,\ s$$
$$\Sigma \vdash \{\, n_1 \,\} \star n_2,\ s \longrightarrow^{k_\star} n,\ s, \quad \text{where } n = \{\, n_1 \,\} \star n_2$$
$$\Sigma \vdash \mathbf{if}\, 0 \,\mathbf{then}\, M_1 \,\mathbf{else}\, M_2,\ s \longrightarrow^{k_{\text{if}}} M_1,\ s$$
$$\Sigma \vdash \mathbf{if}\, n \,\mathbf{then}\, M_1 \,\mathbf{else}\, M_2,\ s \longrightarrow^{k_{\text{if}}} M_2,\ s, \quad \text{where } n \neq 0$$
$$\Sigma \vdash !v,\ s \otimes (v \mapsto n) \longrightarrow^{k_{\text{asg}}} n,\ s \otimes (v \mapsto n)$$
$$\Sigma \vdash v := n',\ s \otimes (v \mapsto n) \longrightarrow^{k_{\text{der}}} \mathbf{skip},\ s \otimes (v \mapsto n')$$
$$\Sigma \vdash \mathbf{grb}(v),\ s \otimes (v \mapsto 0) \longrightarrow^{k_{\text{grb}}} \mathbf{skip},\ s \otimes (v \mapsto 1)$$
$$\Sigma \vdash \mathbf{rls}(v),\ s \otimes (v \mapsto n) \longrightarrow^{k_{\text{rel}}} \mathbf{skip},\ s \otimes (v \mapsto 0), \quad \text{where } n \neq 0$$
$$\Sigma \vdash (\lambda x.M)M',\ s \longrightarrow^{k_{\text{app}}} M[M'/x],\ s$$
$$\Sigma \vdash \mathbf{fix}\, M,\ s \longrightarrow^{k_{\text{app}}} M(\mathbf{fix}\, M),\ s$$

Figure 2: Basic reduction rules with costs for ICA

environment (O), hence unknown. So the second program saves k assignments and $k \times l$ dereferencings.

Any play of the first program can be found in the second program, but with no costs. We can (informally) conclude that the latter is an *improvement* of the former — if this was the result of a compiler optimisation, such as constant propagation, then we can also conclude that the optimisation was successful. We will make the notion of improvement precise, operationally and denotationally, in the following section, and will show that the two notions coincide through a full abstraction result.

4. AN OPERATIONAL THEORY OF IMPROVEMENT FOR ICA

Sands [30, 31] and Moran and Sands [21] have introduced an operational theory of improvement to analyse quantitatively and qualitatively the effect of program transformations in various purely functional languages. We will adapt the theory of improvement to ICA.

Sands assigns a unitary cost to each reduction step. As argued previously, it makes sense to assign different (non-negative) costs to each reduction rule. We decorate the reduction rules in Fig. 1 with costs, as in Fig. 2. Assigning the same cost k_{app} to application and each iteration of the fix-point operator is not essential but it makes sense in most execution models.

Rules for local variables, semaphores and recursion do not incur additional costs. The in-context reduction rules propagate the costs:

$$\frac{\Sigma \vdash M,\ s \longrightarrow^n M',\ s'}{\Sigma \vdash \mathcal{E}[M],\ s \longrightarrow^n \mathcal{E}[M'],\ s'},$$

where the evaluation contexts $\mathcal{E}[-]$ are as before.

We take the transitive closure of the reduction relation by adding the costs along the derivation:

$$\frac{\Sigma \vdash M,\ s \longrightarrow^n M',\ s'}{\Sigma \vdash M,\ s \rightsquigarrow^n M',\ s'}$$

$$\frac{\Sigma \vdash M,\ s \rightsquigarrow^n M',\ s' \quad \Sigma \vdash M',\ s' \rightsquigarrow^{n'} M'',\ s''}{\Sigma \vdash M,\ s \rightsquigarrow^{n+n'} M'',\ s''}.$$

We introduce costs in the notion of termination. We write $M, s \Downarrow^n$ if there exists a constant c and state s' such that $\Sigma \vdash M,\ s \rightsquigarrow^n c,\ s'$. If M is closed and $M, \emptyset \Downarrow^n$ we write $M \Downarrow^n$. If $M \Downarrow^n$, $n \leq n'$ we write $M \Downarrow^{\leq n'}$.

Definition 5 (Improvement). *We say M* may be improved *by N, written $M \gtrsim N$ if $\forall \mathcal{C}[-]$, if $\mathcal{C}[M] \Downarrow^n$ then $\mathcal{C}[N] \Downarrow^{\leq n}$.*

Note that the definition of improvement is angelic, just like the definition of termination. The definition above entails that N is both more efficient and terminates more often than M.

An operational theory of improvement relies crucially on the existence of a strong enough context lemma. Indeed, the varieties of λ-calculi analysed by Sands satisfy the following context lemma:

Lemma 1 (Context [21]). *For all terms M, N, of the lazy lambda calculus, if $\mathcal{E}[M] \gtrsim \mathcal{E}[N]$ for all evaluation contexts $\mathcal{E}[-]$, then $M \gtrsim N$.*

That means that only configuration contexts of a certain form, with the hole $[-]$ occurring exactly once need be considered. This reduces the number of contexts to be considered in proofs requiring induction on contexts.

In general it is known that purely functional programming languages satisfy strong context lemmas. However, formulating context lemmas for imperative programming languages is more difficult (see [25] or [19]). It is not known whether the language ICA satisfies a strong enough context lemma to allow for a usable theory of improvement to be developed along operational lines. We will pursue an alternative, semantic, approach.

5. A SLOT-GAME SEMANTICS OF IMPROVEMENT FOR ICA

To create a slot-game model we add quantitative information to the game-semantic model, in a way that is consistent with the operational semantics with costs of the previous section.

If s, t are sequences and m a move, let $ms \frown t = mts$, i.e. insert t into ms just after the first move m. If σ is a strategy, we define

$$\sigma \frown t = \{s \frown t \mid s \in \sigma\}.$$

The slot game model is defined by instrumenting strategies with tokens representing costs incurred during execution. Some of the strategies-with-costs are shown in Fig. 3. The operator $\star$ ranges over the binary and unary operators of the language, and $\mathsf{op}_\star$ is its game-semantic interpretation. The strategy ev is the evaluation morphism in $\mathcal{G}_{sat}$. Λ is a relabelling of moves which also represents the currying isomorphism in $\mathcal{G}_{sat}$. Finally, cell and lock are the $\mathcal{G}_{sat}$ strategies used to interpret stateful behaviour.

The interpretation of the fix-point operator is a little more complicated. Let $[\![\mathbf{fix}_\theta : (\theta_1 \to \theta_2) \to \theta_3]\!]$ be the game-semantic

$$\llparenthesis\Gamma \vdash MN : \theta\rrparenthesis = \big\langle \llparenthesis\Gamma \vdash M : \theta' \to \theta\rrparenthesis ; \llparenthesis\Gamma \vdash N : \theta'\rrparenthesis\big\rangle ; (\mathsf{ev} \frown \textcircled{k_{\mathrm{app}}})$$
$$\llparenthesis\Gamma \vdash \lambda x.M : \theta' \to \theta\rrparenthesis = \Lambda\big(\llparenthesis\Gamma, x : \theta' \vdash M : \theta\rrparenthesis\big)$$
$$\llparenthesis\Gamma \vdash M_1 \star M_2 : \theta\rrparenthesis = \big\langle \llparenthesis\Gamma \vdash M_1 : \theta_1\rrparenthesis, \llparenthesis\Gamma \vdash M_2 : \theta_2\rrparenthesis\big\rangle ; (\mathsf{op}_\star \frown \textcircled{k_\star})$$
$$\llparenthesis\Gamma \vdash \star M : \theta\rrparenthesis = \llparenthesis\Gamma \vdash M : \theta'\rrparenthesis ; (\mathsf{op}_\star \frown \textcircled{k_\star})$$
$$\llparenthesis\Gamma \vdash \mathbf{newvar}\, x := n \,\mathbf{in}\, N\rrparenthesis = \Lambda\big(\llparenthesis\Gamma, x : \mathbf{var} \vdash M : \theta\rrparenthesis\big) ; (\mathsf{cell}^\theta_n \frown \textcircled{k_{\mathrm{var}}})$$
$$\llparenthesis\Gamma \vdash \mathbf{newsem}\, x := n \,\mathbf{in}\, N\rrparenthesis = \Lambda\big(\llparenthesis\Gamma, x : \mathbf{sem} \vdash M : \theta\rrparenthesis\big) ; (\mathsf{lock}^\theta_n \frown \textcircled{k_{\mathrm{sem}}})$$

Figure 3: Slot-game valuations

interpretation of fix-point at type θ. The strategy-with-costs $\llparenthesis\mathbf{fix}_\theta\rrparenthesis$ is obtained by inserting $\textcircled{k_{\mathrm{app}}}$ before each opening question in type component θ_2.

For constants and free variables the interpretations are the same as in the game model:

$$\llparenthesis n\rrparenthesis = [\![n]\!] = \mathsf{strat}(q\ n)$$
$$\llparenthesis\mathbf{skip}\rrparenthesis = [\![\mathbf{skip}]\!] = \mathsf{strat}(run\ ok)$$
$$\llparenthesis x : \theta \vdash x : \theta\rrparenthesis = [\![x : \theta \vdash x : \theta]\!] = \mathsf{id}_{[\![\theta]\!]}.$$

Note that we attach the costs of a strategy just after the initial question, with the exception of fix-point, which incurs the cost at each iteration. This is only a matter of convention. For our analysis, it is not relevant at what point in time costs are incurred. In fact, because strategies for ICA are characterised by their sets of complete plays, our model could be presented by assigning *global* costs to plays rather than interleaving tokens into plays. However, this presentation would be cumbersome for several reasons. The definition of composition would become more complicated. Also, identifying languages or fragments of languages admitting finitary representations would become more difficult. Finally, such a presentation of the model would not scale at all to models where strategies are not characterised by complete plays or to more refined analyses that may exploit the possibility of recognising the temporal significance of occurrences of costs.

In the following, we relate slot games to the operational notion of improvement.

For the purpose of relating our model with the operational semantics we need to recover explicit state. We represent a state $s : \Sigma \to \mathbb{N}$ by a strategy.

$$\llparenthesis s\rrparenthesis^\beta : \big([\![\theta_1]\!] \Rightarrow \cdots \Rightarrow [\![\theta_m]\!] \Rightarrow [\![\beta]\!]\big) \Rightarrow [\![\beta]\!]\,.$$

$\llparenthesis s\rrparenthesis^\beta$ first copies the initial O-question q to the other $[\![\beta]\!]$ subgame. Then it behaves in each θ_i component like suitably initialised $\mathsf{cell}^\theta_{n_i}$ and $\mathsf{lock}^\theta_{n_i}$ strategies. Finally, when the copy of the initial question is answered by O, $\llparenthesis s\rrparenthesis_\beta$ answers the initial question with the same answer. This strategy does not incur any costs.

The strategy with costs $\llparenthesis\Sigma \vdash M : \theta\rrparenthesis ; \llparenthesis s\rrparenthesis^\beta$ will be the interpretation of $\Sigma \vdash M : \beta$ at state s. If clear from the context we omit the β superscript.

Lemma 2. *If $\Sigma \vdash M$, $s \longrightarrow^n M'$, s' is a basic reduction then $\llparenthesis\Sigma \vdash M\rrparenthesis ; \llparenthesis s\rrparenthesis = \llparenthesis\Sigma \vdash M'\rrparenthesis ; \llparenthesis s'\rrparenthesis \frown \textcircled{n}$.*

Proof: Immediate, for each basic reduction rule with costs, from the definitions. □

Lemma 3. *If $\Sigma \vdash M$, $s \longrightarrow^n M'$, s' is a reduction then there exists a strategy with costs $\sigma \subseteq \llparenthesis\Sigma \vdash M\rrparenthesis ; \llparenthesis s\rrparenthesis \amalg \textcircled{n}$ such that $\sigma \supseteq \llparenthesis\Sigma \vdash M'\rrparenthesis ; \llparenthesis s'\rrparenthesis$.*

Proof: If $\Sigma \vdash M$, $s \longrightarrow^n M'$, s' then there must be an evaluation context $\mathcal{E}$ such that $M = \mathcal{E}[M_0]$, $M' = \mathcal{E}[M'_0]$, and $\Sigma \vdash M_0$, $s \longrightarrow^n M'_0$, s' a basic reduction rule.

The argument is by induction on evaluation contexts $\mathcal{E}$. Lem. 2 gives the base case; the inductive cases require an analysis of the interaction sequences in $\llparenthesis\Sigma \vdash M\rrparenthesis ; \llparenthesis s\rrparenthesis$ similar to that in the analogous lemma of [10]. □

Lemma 4. *If $\Sigma \vdash M$, $s \rightsquigarrow^n M'$, s' then there exists a strategy with costs $\sigma \subseteq \llparenthesis\Sigma \vdash M\rrparenthesis ; \llparenthesis s\rrparenthesis \amalg \textcircled{n}$ such that $\sigma \supseteq \llparenthesis\Sigma \vdash M'\rrparenthesis ; \llparenthesis s'\rrparenthesis$.*

Proof: By induction on the derivation of $\Sigma \vdash M$, $s \rightsquigarrow^n M'$, s'. The base case is Lem. 3, and the induction is immediate. □

An immediate corollary of Lem 4 is a soundness result:

Lemma 5 (Soundness). *If $M, s \Downarrow^n$ then $\exists t \in \mathsf{comp}\big(\llparenthesis\Sigma \vdash M\rrparenthesis ; \llparenthesis s\rrparenthesis\big)$ such that $|t| = n$.*

To wit, if a term M in state s may be evaluated at cost n then the slot-game model of the term contains a play which uses precisely n tokens. Using logical relations, along the same lines as in [9], we can also show the converse:

Lemma 6 (Computability). *If $\exists t \in \mathsf{comp}\big(\llparenthesis\Sigma \vdash M\rrparenthesis ; \llparenthesis s\rrparenthesis\big)$ such that $|t| = n$ then $M, s \Downarrow^n$.*

Soundness and computability immediately entail:

Lemma 7 (Computational adequacy). *$M, s \Downarrow^n$ if and only if $\exists t \in \mathsf{comp}\big(\llparenthesis\Sigma \vdash M\rrparenthesis ; \llparenthesis s\rrparenthesis\big)$ such that $|t| = n$.*

We can now relate a semantic notion of improvement to the operational one.

Let Σ be the game with a single question q and answer a, such that $q \vdash_\Sigma a$. We consider the strategies-with-costs $\top$ for Σ such that $\mathsf{comp}(\top)$ has only complete plays of the form $q\ \textcircled{n}\ a$. If the *shortest* complete play in a strategy-with-costs uses n tokens we say $|\sigma| = n$. For two strategies $\top$, $\top'$, if $|\top| \geq |\top'|$ we say $\top \gtrsim \top'$. We denote by $\bot$ the strategies-with-costs without complete plays. We also have $\bot \gtrsim \bot$ and for any $\top$, $\bot \gtrsim \top$. The strategy $\bot$ is the *least defined* strategy therefore any other strategy represents an improvement of it.

Definition 6 (Semantics of improvement). *Given two strategies-with-costs σ, τ on arena A we say that σ is* improved *by τ, denoted by $\sigma \gtrsim \tau$, if for any strategy-with-costs α on $A \Rightarrow \Sigma$ we have $\sigma;\alpha \gtrsim \tau;\alpha$.*

Note that improvement is defined quite similarly to the *intrinsic preorder* on games.

It is straightforward to show that improvement represents a partial order on strategies, and that composition of strategies is monotonic relative to it.

For our language we can also give a more effective characterisation of improvement: τ improves σ if it contains all its plays but with smaller costs.

Lemma 8 (Characterisation). *$\sigma \gtrsim \tau$, if $\forall s \in \mathsf{comp}(\sigma)$, $\exists t \in \mathsf{comp}(\tau)$ such that $s^\dagger = t^\dagger$ and $|s| \geq |t|$.*

Theorem 2 (Soundness of improvement). *For any terms $\Gamma \vdash M, N : \theta$, if $\llparenthesis M \rrparenthesis \gtrsim \llparenthesis N \rrparenthesis$ then $M \gtrsim N$.*

Proof: Suppose that for some context $\mathcal{C}[-]$ we have $\mathcal{C}[M] \Downarrow^n$. By adequacy, $\exists s \in \mathsf{comp}\big(\llparenthesis \mathcal{C}[M] \rrparenthesis\big)$ such that $|s| = n$. From $\llparenthesis M \rrparenthesis \gtrsim \llparenthesis N \rrparenthesis$, using the monotonicity of the semantics, it follows that $\llparenthesis \mathcal{C}[M] \rrparenthesis \gtrsim \llparenthesis \mathcal{C}[N] \rrparenthesis$. Together these imply that $\exists t \in \mathsf{comp}\big(\llparenthesis \mathcal{C}[N] \rrparenthesis\big)$ such that $|t| = m \leq n$. Using adequacy it follows $\mathcal{C}[N] \Downarrow^m$, which implies $M \gtrsim N$. □

For full abstraction we can rely on definability in the cost-free model:

Theorem 3 ($\mathcal{G}_{sat}$ definability [10]). *For any type θ and play in $s \in P_{\llbracket\theta\rrbracket}$ there exists a term $M : \theta$ such that $\llbracket M \rrbracket = \mathsf{strat}(s)$.*

There is no similar definability result for slot-games, i.e. we cannot always find terms that use an arbitrary number of tokens in an arbitrary fashion. This is because we cannot always construct terms which are arbitrarily efficient. There is a minimum amount of tokens that must be spent in order to produce any given trace, but no more than the amount of tokens used by the terms produced by the definability algorithm for $\mathcal{G}_{sat}$ (as given in [10]). Also note that we do not insist on a particular scheduling of costs relative to moves, as long as we control the overall cost of a play. However, definability for underlying strategies is enough to prove the following:

Theorem 4 (Full abstraction of improvement). *For any terms $\Gamma \vdash M, N : \theta$, $\llparenthesis M \rrparenthesis \gtrsim \llparenthesis N \rrparenthesis$ iff $M \gtrsim N$.*

Proof: The left-to-right direction is soundness of improvement. For the right-to-left direction we prove by contradiction. Assume $\llparenthesis M \rrparenthesis \not\gtrsim \llparenthesis N \rrparenthesis$. There are two cases:

1. $\llparenthesis M \rrparenthesis^\dagger \not\subseteq \llparenthesis N \rrparenthesis^\dagger$, i.e. the underlying strategy interpreted M is not included into that for N. Using full abstraction for may-termination for ICA it follows immediately that $M \not\sqsupseteq N$, hence $M \not\gtrsim N$, which contradicts the hypothesis.

2. $\exists s \in \llparenthesis M \rrparenthesis$, $t \in \llparenthesis N \rrparenthesis$ such that $s^\dagger = t^\dagger = u$ and $|s| < |t|$. Let us consider a strategy (without costs) in $\mathcal{G}_{sat}$, $\sigma^\dagger = \mathsf{strat}(v)$ where v is a play in $\llbracket\theta\rrbracket \Rightarrow \Sigma$ such that $\mathsf{strat}(u);\sigma^\dagger = \mathsf{strat}(q\ a)$. From the definability theorem for ICA (Thm. 3) we know that there exists a term $M : \theta \rightarrow \mathbf{com}$, which can be written also as $x : \theta \vdash \mathcal{C}[x] : \mathbf{com}$, such that $\llbracket \mathcal{C}[x] \rrbracket = \sigma^\dagger$. Let $\sigma = \llparenthesis \mathcal{C}[x] \rrparenthesis$. By definition of semantic improvement and by monotonicity of composition, $\llparenthesis M \rrparenthesis;\sigma \not\gtrsim \llparenthesis N \rrparenthesis;\sigma$. But $\llparenthesis M \rrparenthesis;\sigma = \llparenthesis M \rrparenthesis;\llparenthesis \mathcal{C}[x] \rrparenthesis = \llparenthesis \mathcal{C}[M] \rrparenthesis$ and $\llparenthesis N \rrparenthesis;\sigma = \llparenthesis N \rrparenthesis;\llparenthesis \mathcal{C}[x] \rrparenthesis = \llparenthesis \mathcal{C}[N] \rrparenthesis$. This implies $\llparenthesis \mathcal{C}[M] \rrparenthesis \not\gtrsim \llparenthesis \mathcal{C}[N] \rrparenthesis$. By adequacy of improvement (Lem. 7) this implies $\mathcal{C}[M] \not\gtrsim \mathcal{C}[N]$ therefore, by definition, $M \not\gtrsim N$ which contradicts the hypothesis.

□

This result shows that the operational and denotational theories of improvement are identical. Both theories are parameterised by the costs of the basic operations of the language in a flexible and uniform way.

5.1 Strong improvement

In practice it is also useful to work with a stronger notion of improvement which compares the costs of programs which are otherwise equivalent.

Definition 7 (Strong improvement). *We say that M* is strongly improved *by N, written $M \trianglerighteq N$, if $M \gtrsim N$ and $M \cong N$.*

Definition 8 (Semantics of strong improvement). *We say that strategy σ* is strongly improved *by τ, written $\sigma \trianglerighteq \tau$, if $\sigma \gtrsim \tau$ and $\sigma^\dagger = \tau^\dagger$.*

It is straightforward to see that the semantics of strong improvement is also fully abstract.

Theorem 5 (Full abstr. of strong improvement). *For any terms $\Gamma \vdash M, N : \theta$, $\llparenthesis M \rrparenthesis \trianglerighteq \llparenthesis N \rrparenthesis$ iff $M \trianglerighteq N$.*

Proof: We use the full abstraction of improvement and the full abstraction of the game model, along with the observation that $\llparenthesis M \rrparenthesis^\dagger = \llbracket M \rrbracket$. □

6. APPLICATIONS AND EXAMPLES

In Sec. 3.1 we looked at a concrete example of improvement. In this section we will consider several more general examples of improvement for programs (or program fragments). We will not attempt to develop a full-blown general syntactic theory of improvement akin to Moran and Sands *tick algebra* because the equational theory of a language such as ICA, which combines higher order procedures with local state, semaphores and concurrency is not known and likely hard to define. However, we can find interesting, albeit ad hoc, improvement schemata and prove their soundness semantically (cf. [16]).

6.1 Simple general laws

ICA has few simple general schemata for improvement. Two of them are *function inlining* and *dead variable elimination.*

Function inlining (strong) improvement is:

$$(\lambda x.F)(M) \trianglerighteq F[x/M].$$

Proof: $\llparenthesis (\lambda x.F)(M) \rrparenthesis^\dagger = \llbracket (\lambda x.F)(M) \rrbracket = \llbracket F[x/M] \rrbracket = \llparenthesis F[x/M] \rrparenthesis^\dagger$. $\llparenthesis (\lambda x.F)(M) \rrparenthesis = \langle \llparenthesis \lambda x.F \rrparenthesis, \llparenthesis M \rrparenthesis \rangle; (\mathsf{ev} \frown \textcircled{k_{\mathrm{app}}})$, but $\llparenthesis F[M/x] \rrparenthesis = \langle \llparenthesis \lambda x.F \rrparenthesis, \llparenthesis M \rrparenthesis \rangle; \mathsf{ev}$. This means that $\forall s \in \llparenthesis (\lambda x.F)(M) \rrparenthesis$ s.t. $|s| = k$, $\exists t \in \llparenthesis F[M/x] \rrparenthesis$ such that $|t| = k + k_{\mathrm{app}}$. As expected, for any play we save the cost of one application. So

$[\![(\lambda x.F)(M)]\!] \trianglerighteq [\![F[x/M]]\!]$ and we can use the full abstraction result. $\Box$

The result above is tied to our particular cost assignment scheme for application, which assigns the same overhead cost to a function notwithstanding its strictness properties. For example, the same overhead is incurred by function $\lambda x.0$ and $\lambda x.x + x$. If this is considered unrealistic for a particular execution (or compilation) model we can give the evaluation strategy ev at type $\theta_1 \to (\theta_1' \to \theta_2) \to \theta_2'$ a different cost assignment, for example adding a cost $\textcircled{k_{\mathrm{app}}}$ after each opening move in θ_1. This would make functions incur costs proportional to the number of times they use the argument. To preserve the full abstraction property of the model a similar change to the cost scheme must be applied to the operational semantics as well.

Dead variable elimination (strong) improvement is:

$$\mathbf{newvar}\, x\, \mathbf{in}\, C \trianglerighteq C, \qquad x \notin \mathit{freevar}(C).$$

PROOF: $[\![\mathbf{newvar}\, x\, \mathbf{in}\, C]\!] = [\![C]\!]; (\mathsf{cell} \frown \textcircled{k_{\mathrm{var}}})$. However, since variable x does not occur free in C, $[\![C]\!]; \mathsf{cell} = [\![C]\!]$. It follows that $[\![\mathbf{newvar}\, x\, \mathbf{in}\, C]\!] = [\![C]\!] \frown \textcircled{k_{\mathrm{var}}}$, so obviously $[\![\mathbf{newvar}\, x\, \mathbf{in}\, C]\!] \trianglerighteq [\![C]\!]$, and we can use full abstraction. $\Box$

6.2 Local variable manipulation

This transformation reflects the intuition that operations only involving a local variable may be ignored:

$$\mathbf{newvar}\, x\, \mathbf{in}\, F(X; n) \gtrsim F(n), \quad n \in \mathbb{N}$$

if $x \notin \mathit{freevar}(F)$ and $\mathit{freevar}(X) = \{x\}$. Note that in this situation the relation is not one of strong improvement, because the removed operation X may cause divergence.

PROOF: $\mathit{freevar}(X) = \{x\}$ implies that $\mathsf{comp}([\![X; n]\!])$ has only sequences of the form $q\ s\ n$ where s consists only of moves in $[\![\mathbf{var}]\!]$ and $\textcircled{\$}$. Composition with cell will therefore produce a strategy with complete plays of the form $q\ \textcircled{k}\ n$ only.

$$\begin{aligned}
&[\![\Gamma \vdash \mathbf{newvar}\, x\, \mathbf{in}\, F(X; n) : \mathbf{com}]\!] \\
&\quad = [\![\Gamma, x : \mathbf{var} \vdash F(X; n) : \mathbf{com}]\!]; \mathsf{cell}_x' \\
&\quad = \langle [\![\Gamma, x : \mathbf{var} \vdash F : \mathbf{exp} \to \mathbf{com}]\!], \\
&\qquad [\![\Gamma, x : \mathbf{var} \vdash X; n : \mathbf{exp}]\!] \rangle; \mathsf{ev}'; \mathsf{cell}_x'.
\end{aligned}$$

where $\mathsf{cell}_x' = \mathsf{cell} \frown \textcircled{k_{\mathrm{var}}}$ and $\mathsf{ev}_x' = \mathsf{ef} \frown \textcircled{k_{\mathrm{app}}}$

Every play in $[\![\Gamma, x : \mathbf{var} \vdash F(X; n) : \mathbf{com}]\!]$ is an interleaving of a play from $[\![F]\!]$ and several plays from $[\![X; n]\!]$. Since $\mathit{freevar}(V) \not\ni x$, composition with cell_x' has no effect over these plays. On the other hand $\mathit{freevar}(X; n) = \{x\}$, so composition with cell_x' will hide all actions on variable x, leaving only plays of the form $q\ \textcircled{k}\ n$ or incomplete plays.

$$\begin{aligned}
&[\![\Gamma \vdash \mathbf{newvar}\, x\, \mathbf{in}\, F(X; n) : \mathbf{com}]\!] \\
&\quad \subseteq \langle [\![\Gamma \vdash F : \mathbf{exp} \to \mathbf{com}]\!], \mathsf{strat}(q\ \textcircled{k}\ n) \rangle; \mathsf{ev}' \\
&\quad \trianglerighteq \langle [\![\Gamma \vdash F : \mathbf{exp} \to \mathbf{com}]\!], \mathsf{strat}(q\ n) \rangle; \mathsf{ev}' \\
&\quad = [\![\Gamma \vdash F(n) : \mathbf{com}]\!].
\end{aligned}$$

Which implies:

$$[\![\Gamma \vdash \mathbf{newvar}\, x\, \mathbf{in}\, F(X; n) : \mathbf{com}]\!] \gtrsim [\![\Gamma \vdash F(n) : \mathbf{com}]\!].$$

Then we use the full abstraction result. $\Box$

6.3 Forced evaluation

In call-by-name languages one major source of inefficiency is the fact that an argument is unnecessarily re-evaluated several times. If an argument computes a value and it has no side-effects then we can improve a program by storing its value in a local variable:

$$f(E) \trianglerighteq \mathbf{newvar}\, x\, \mathbf{in}\, x := E; f(!x).$$

This is subject to the following assumptions:

- f is strict: $f\Omega \cong \Omega$
- E has no side-effects: $[\![E]\!] = \mathsf{strat}(q\ \textcircled{k}\ n)$ for some $k, n \in \mathbb{N}$
- the overhead of local variable manipulation and dereferencing are (relatively) negligible: $k_{\mathrm{var}} = k_{\mathrm{der}} = 0$.

PROOF: We use q_f, n_f to indicate moves in the *argument* of function f. Using the semantic definitions, and the fact that f must be strict,

$$[\![f(E)]\!] = \mathsf{strat}\big(\mathit{run} \sum_{i>0} \coprod_{1 \le j \le i} (q_f\ \textcircled{k}\ n_f)\ \mathit{ok}\big).$$

On the other hand,

$$\begin{aligned}
&[\![x := E; f(!x)]\!] = \\
&\qquad \mathsf{strat}\big(\mathit{run}\ \textcircled{k}\ \mathit{write}(n)_x\ \mathit{ok}_x \\
&\qquad\quad \sum_{i>0} \coprod_{1 \le j \le i} (q_f\ \mathit{read}_x\ (n_{i,j})_x\ (n_{i,j})_f)\ \mathit{ok}\big),
\end{aligned}$$

since the cost of ! is deemed negligible. Upon composition with cell we have:

$$\begin{aligned}
&[\![\mathbf{newvar}\, x\, \mathbf{in}\, x := E; f(!x)]\!] = \\
&\qquad\qquad \mathsf{strat}\big(\mathit{run}\ \textcircled{k} \sum_{i>0} \coprod_{1 \le j \le i} (q_f\ n_f)\ \mathit{ok}\big).
\end{aligned}$$

It is obvious that $[\![f(E)]\!] \trianglerighteq [\![\mathbf{newvar}\, x\, \mathbf{in}\, x := E; f(!x)]\!]$, since the cost $\textcircled{k}$ is only incurred exactly once in the rhs.

Note that the strictness of f is important: if the function does not evaluate the argument then $f(E)$ does not incur the cost of E at all, whereas $\mathbf{newvar}\, x := E\, \mathbf{in}\, f(!v)$ does; moreover, if E diverges the former term does not diverge, but the latter does. $\Box$

Since $f : \mathbf{exp} \to \mathbf{com}$ is a free identifier, the condition *f is strict* must be interpreted as *f must be bound only to a strict function F*. We have chosen to work with an identifier (rather than a term F, as in the previous example) for presentational reasons, because the proof is clearer. Since we are in a call-by-name language, it is indifferent whether we are working with a term or an identifier, and we will choose whatever is more convenient. This should not cause any confusions.

6.4 Loop unrolling

A common compiler optimisation is to "unroll" for-loops.

$$\begin{aligned}
&\mathbf{newvar}\, x := 0\, \mathbf{in}\, \mathbf{while}(x < n)\ \mathbf{do}\ C; x := !x + 1 \\
&\qquad\qquad \trianglerighteq \underbrace{C; \cdots; C}_{n \text{ times}}, \qquad x \notin \mathit{freevar}(C)
\end{aligned}$$

The proof of this is immediate from the semantic definitions.

6.5 Sequentialisation

In our cost model parallel execution may incur additional costs. Therefore, it is an improvement wherever possible to execute a program sequentially rather than concurrently. For example:

$$\begin{aligned}&\mathbf{newsem}\, s := 0\, \mathbf{in}\\ &\quad (\mathbf{grb}(s); C_1; \mathbf{rls}(s)) \,||\, (\mathbf{grb}(s); C_2; \mathbf{rls}(s))\\ &\qquad \trianglerighteq C_1; C_2 \;\mathbf{or}\; C_2; C_1, \quad s \notin \mathit{freevar}(C_1, C_2)\end{aligned}$$

where

$$\begin{aligned}&[\![\mathbf{or} : \mathbf{com}_1 \to \mathbf{com}_2 \to \mathbf{com}]\!]\\ &\qquad = \mathsf{strat}\{\mathit{run}\ \mathit{run}_1\ \mathit{ok}_1\ \mathit{ok},\ \mathit{run}\ \mathit{run}_2\ \mathit{ok}_2\ \mathit{ok}\}\end{aligned}$$

is nondeterministic choice. Introducing this does not affect in any way the full abstraction result.

Proof: Let us assume for simplicity the only relevant costs are those associated with semaphore manipulation. Then we have:

$$\begin{aligned}&[\![(\mathbf{grb}(s); C_1; \mathbf{rls}(s)) \,||\, (\mathbf{grb}(s); C_2; \mathbf{rls}(s))]\!] =\\ &\quad \mathsf{strat}\{\mathit{run}\ (\mathit{grb}\ \text{\textcircled{\$}}\ \mathit{ok}\ c_1\ \mathit{rls}\ \text{\textcircled{\$}}\ \mathit{ok}\ \mathit{ok}\\ &\qquad \amalg\ \mathit{grb}\ \text{\textcircled{\$}}\ \mathit{ok}\ c_2\ \mathit{rls}\ \text{\textcircled{\$}}\ \mathit{ok})\ \mathit{ok}\ \mid \mathit{run}\ c_i\ \mathit{ok} \in \mathsf{strat}([\![C_i]\!])\}\end{aligned}$$

Upon composition with lock we get

$$\begin{aligned}&[\![\mathit{LHS}]\!] = \mathsf{strat}\{\mathit{run}\ \text{\textcircled{\$}}\ c_1\ \text{\textcircled{2}}\ c_2\ \text{\textcircled{\$}}\ \mathit{ok},\\ &\qquad \mathit{run}\ \text{\textcircled{\$}}\ c_2\ \text{\textcircled{2}}\ c_1\ \text{\textcircled{\$}}\ \mathit{ok}\ \mid \mathit{run}\ c_i\ \mathit{ok} \in \mathsf{strat}([\![C_i]\!])\},\end{aligned}$$

whereas

$$\begin{aligned}&[\![\mathit{RHS}]\!] = \mathsf{strat}\{\mathit{run}\ c_1\ c_2\ \mathit{ok},\ \mathit{run}\ c_2\ c_1\ \mathit{ok}\\ &\qquad\qquad \mid \mathit{run}\ c_i\ \mathit{ok} \in \mathsf{strat}([\![C_i]\!])\}.\end{aligned}$$

□

6.6 Abstract data types

One of the advantages of a semantic model is that it allows us to reason about open programs. A particularly interesting kind of open programs are *abstract data types* (ADT). In this case, the "missing" part of the program is the context in which the ADT is used. In [1] we have examined how games-based techniques can be used to model-check safety properties of ADTs. We can use a similar approach to reason about improvement of ADTs.

Let us consider the following toy ADT implementing a switch, which has two methods in its interface:

on evaluate the state of the switch, which is initially *off*

flick change the state of the switch to *on*.

Let us look at two possible implementations of this switch ADT:

Method	Implementation (1)	Implementation (2)
on	$\mathbf{grb}\ s; !x > 0; \mathbf{rls}\ s$	$\mathbf{grb}\ s; !x > 0; \mathbf{rls}\ s$
flick	$\mathbf{grb}\ s; x := !x + 1; \mathbf{rls}\ s$	$\mathbf{grb}\ s; x := 1; \mathbf{rls}\ s$,

where v is a global variable and s is a global semaphore. It is clear that the second implementation is an improvement, since it changes the state using only one assignment as opposed to an assignment, an addition and a dereferencing.

This can be formalised as:

$$\begin{array}{lcl}\multicolumn{3}{l}{\mathit{context} : \mathbf{exp} \to \mathbf{com} \to \mathbf{com} \vdash}\\ \mathbf{newvar}\, x := 0\, \mathbf{in} & & \mathbf{newvar}\, x := 0\, \mathbf{in}\\ \mathbf{newsem}\, s := 0\, \mathbf{in} & & \mathbf{newsem}\, s := 0\, \mathbf{in}\\ \mathbf{let}\ \mathit{on}\ \mathbf{be} & & \mathbf{let}\ \mathit{on}\ \mathbf{be}\\ \quad \mathbf{grb}\ s; !x > 0; \mathbf{rls}\ s\ \mathbf{in} & \trianglerighteq & \quad \mathbf{grb}\ s; !x > 0; \mathbf{rls}\ s\ \mathbf{in}\\ \mathbf{let}\ \mathit{flick}\ \mathbf{be} & & \mathbf{let}\ \mathit{flick}\ \mathbf{be}\\ \quad \mathbf{grb}\ s; x := !x + 1; \mathbf{rls}\ s\ \mathbf{in} & & \quad \mathbf{grb}\ s; x := 1; \mathbf{rls}\ s\ \mathbf{in}\\ \mathit{context}(\mathit{on})(\mathit{flick}) & & \mathit{context}(\mathit{on})(\mathit{flick}).\end{array}$$

This is proved directly from the semantic definitions, like in the previous examples. More complicated examples can be dealt with similarly and, subject to several assumptions, even verified automatically, as in [1].

6.7 Sorting algorithms

Finally, we use improvement to illustrate how we can compare programs for efficiency. Consider two implementation of sorting, bubble-sort (Fig. 4) and insert-sort (Fig. 5). The $\%k$ annotation means that the variable takes on values in the range $\{-k+1, \ldots, k-1\}$.

We can use (semantic) improvement to show that insert-sort is indeed an improvement over bubble-sort.

The complete plays for the strategies of the two programs are regular languages, so they can be represented as finite-state automata. Moreover, let us assume that we are only interested in comparing the efficiency of the two programs relative to the number of *swap* operations. This is quite standard in comparing sorting algorithms.

Let us add an operation $\mathbf{swap} : \mathbf{var} \to \mathbf{var} \to \mathbf{com}$ to our language, with the obvious semantics for swapping the values of 2 variables, and with cost $\text{\textcircled{2}}$:

$$\begin{aligned}&[\![\mathbf{swap} : \mathbf{var}_1 \to \mathbf{var}_2 \to \mathbf{com}]\!] =\\ &\quad \mathsf{strat}\big(\{\mathit{run}\ \text{\textcircled{2}}\ \mathit{read}_1\ m_1\ \mathit{read}_2\ n_2\\ &\quad \mathit{write}(n)_1\ \mathit{ok}_1\ \mathit{write}(m)_2\ \mathit{ok}_2\ \mathit{ok} \mid m, n \in \mathbb{N}\}\big)\end{aligned}$$

We have adapted the model-checker used in [1] to generate the finite-state model of the two programs.

We show the models for insert-sort and bubble-sort for a small array (3 elements, storing ternary data) in Fig. 6. We choose a small array in order to have a model which can be displayed and analysed informally, but we can also automatically analyse much larger arrays using the methods from [1].

The models contain only the read and write moves on the free variable v : **var**, used to initialise then read the array, and the token actions. These models can be seen as statically generated profiling information. Transitions corresponding to token actions appear as a thicker line. The models above are finite-state and improvement can be easily encoded as a regular-language property, which gives a decision method for checking it. Informally, one can see that the "worst" and the "best" paths in the two models have equal costs (6 and 0 tokens, respectively) but the bubble-sort model has otherwise more costly computation paths than the insert-sort. So even though the best-case and worst-case costs are the same, insert-sort always represents an improvement over bubble-sort.

```
v:var%2 |-
  array%2 a[4] in
  let n be 4 in
  new var%5 i := 0 in
  while !i < n  do
    a[!i] := v;
    i := !i + 1
  od;
  new var%2 flag := 1 in
  while !flag do
  new var%5 i := 0 in
    flag := 0;
    while !i < n - 1 do
      if !a[!i] > !a[!i + 1] then
        flag := 1;
        swap(a[!i], !a[!i+1]);
        else skip fi;
      i := !i + 1
    od
  od;
  new var%5 i := 0 in
  while !i < n  do
    v := !a[!i];
    i := !i + 1
  od;
: com.
```

Figure 4: Bubble-sort

```
v:var%2 |-
  array%2 a[4] in
  let n be 4 in
  new var%5 i := 0 in
  while !i < n  do
    a[!i] := !v;
    i := !i + 1
  od;
  new var%5 i := n in
  while !i > 0 do
    i := !i - 1;
    new var%5 max := 0 in
    new var%5 j := 0 in
    while !j < !i do
      if !a[!max] < !a[!j] then
         max := !j
      else skip fi;
      j := !j + 1
    od;
    if !a[!max] > !a[!i] then
      swap(a[!max],!a[!i])
      else skip fi
  od;
  new var%5 i := 0 in
  while !i < n  do
    v := !a[!i];
    i := !i + 1
  od;
: com.
```

Figure 5: Insert-sort

7. FURTHER WORK

In this paper we have seen how game semantics can be adapted relatively easily to give a quantitative analysis of programming languages, using a new action called *token*. We have also seen that the resulting semantics is exactly that induced by the operational theory of improvement, and made this correspondence rigorous through a full abstraction result. A quantitative analysis of programs has many potential applications, from compiler optimisations to resource-constrained execution and static performance profiling.

Our analysis used the ICA programming language in order to stay focused, but it is quite clear that a similar approach can be extended in a straightforward manner to any language for which a fully abstract game semantics exists, by adding costs consistently to the reduction rules in the operational semantics and to the strategies in the game model. However, formalising the way in which a cost model can be in general derived from a game model is a topic for further research.

The cost model we focused on was a linear one, and it is suitable to modeling resources that are consumed linearly and cannot be reused, such as time or power. More sophisticated cost models must be created for resources that can be reused (e.g. memory) using a suitable notion of observation [12]. Our analysis also assumes a sequential operational model; execution in a concurrent multi-processor environment requires yet a different analysis. However, in the instances above the difficulties are technical rather than conceptual.

In our model we have also abstracted the temporal significance of costs. This was possible because our language does not contain control operators. Consider the two cost schemes for assignment below. For our purposes they are indistinguishable, although the cost is incurred at different moments in time:

$$\mathit{run}\ q\ n\ \mathit{write}(n)\ \mathit{ok}\ \textcircled{\$}\ \mathit{ok}\ \text{vs.}\ \mathit{run}\ \textcircled{\$}\ q\ n\ \mathit{write}(n)\ \mathit{ok}\ \mathit{ok}$$

However, in a language with control where the arguments may interrupt execution the two are distinguishable. If a non-local jump occurs, in the former the cost will not have been incurred whereas in the latter it will have.

We also do not assign polarities to tokens; we do not distinguish whether it is P or O incurring the cost. Assigning polarities to costs can help give semantics to assume-guarantee type systems which *control* resource usage, along the lines of [8]. In contrast, our polarity-free semantics can only analyse resource usage.

Finally, our semantic model is helpful in assigning costs to computations and thus compares programs for efficiency. However, it is not clear how this approach can be used to give a complexity-theoretical characterisation to programs, i.e. relating asymptotically the costs to the size of the inputs. But the concrete nature of the game model could be useful in handling this problem as well.

Acknowledgements The author is indebted to Jakob Rehof for his helpful suggestions.

8. REFERENCES

[1] Abramsky, S., Ghica, D. R., Murawski, A. S., and Ong, C.-H. L. Applying game semantics to

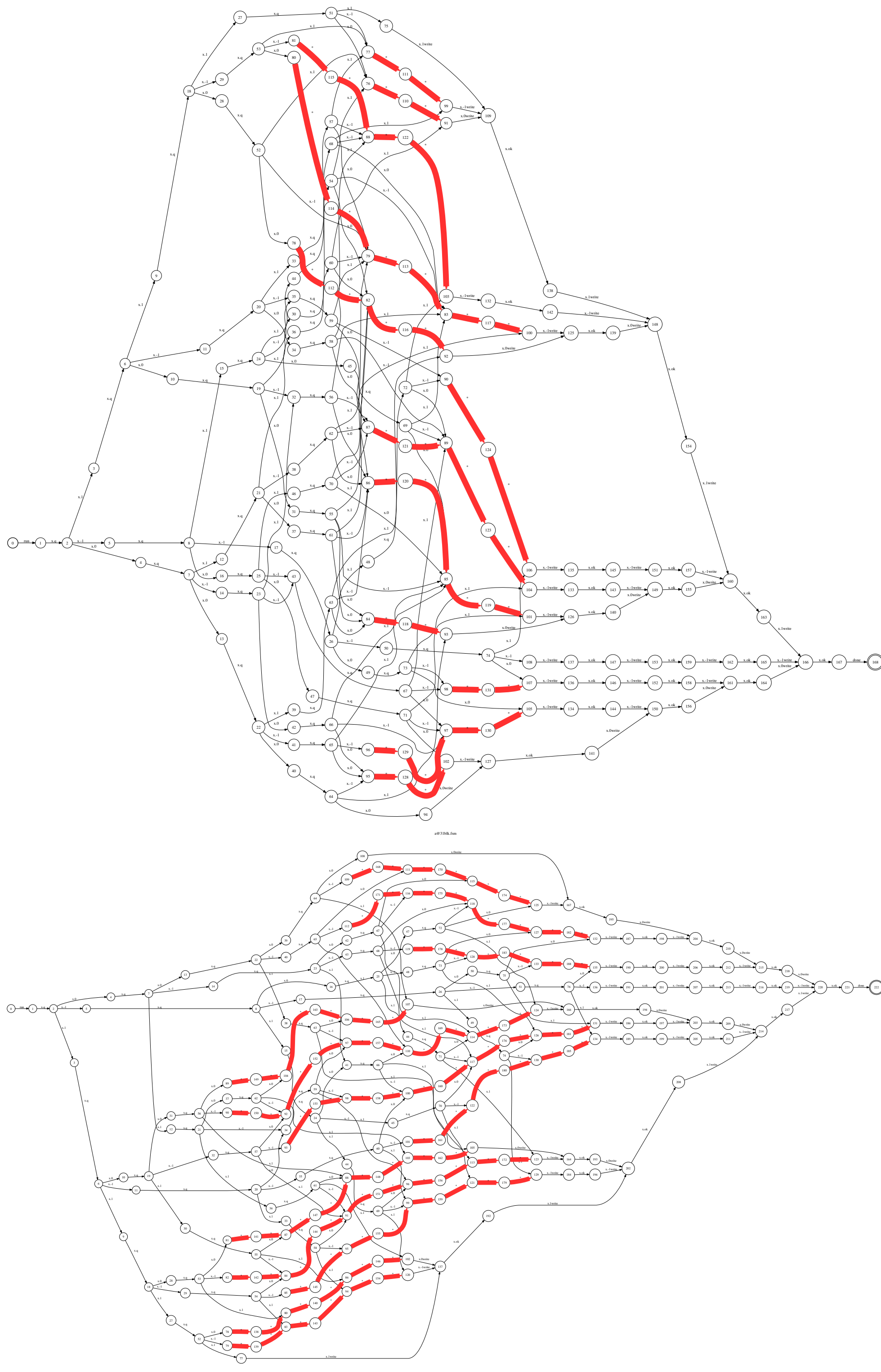

Figure 6: Models for insert-sort and bubble-sort

compositional software modeling and verification. In *TACAS* (Barcelona, 2004), LNCS 2988.

[2] Abramsky, S., Jagadeesan, R., and Malacaria, P. Full abstraction for PCF. *Information and Computation 163* (2000).

[3] Abramsky, S., and McCusker, G. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions (extended abstract). *ENTCS 3*, (1996).

[4] Berry, G., and Curien, P.-L. Sequential algorithms on concrete data structures. *TCS 20* (1982), 265–321.

[5] Bjerner, B., and Holmström, S. A compositional approach to time analysis of first order lazy functional programs. In *Fourth International Conference on Functional Programming Languages and Computer Architecture* (1989).

[6] Brookes, S., and Geva, S. Computational comonads and intensional semantics. In *Applications of Categories in Computer Science: Proceedings LMS Symposium* (Durham, UK, 1991).

[7] Ghica, D. R. Semantical analysis of specification logic, 3: An operational approach. In *ESOP* (Barcelona, 2004), LNCS 2986.

[8] Ghica, D. R., Murawski, A., and Ong, C.-H. L. Syntactic control of concurrency. In *ICALP* (Turku, Finland, 2004), LNCS 3142.

[9] Ghica, D. R., and Murawski, A. S. Angelic semantics of fine-grained concurrency. Tech. Rep. PRG-RR-03-20, Oxford University Computing Laboratory, 2003. `http://users.ox.ac.uk/~coml0074/papers/asfgc.pdf`.

[10] Ghica, D. R., and Murawski, A. S. Angelic semantics of fine-grained concurrency. In *FOSSACS* (Barcelona, 2004), LNCS 2987.

[11] Gurr, D. J. *Semantic frameworks for complexity*. PhD thesis, University of Edinburgh, 1991.

[12] Gustavsson, J., and Sands, D. A foundation for space-safe transformations of call-by-need programs. In *Higher Order Operational Techniques* (Paris, France, 1999), *ENTCS 26*.

[13] Hyland, J. M. E., and Ong, C.-H. L. On full abstraction for PCF: I, II and III. *Information and Computation 163*, 8 (Dec. 2000).

[14] Jay, C. B., Cole, M. I., Sekanina, M., and Steckler, P. A monadic calculus for parallel costing of a functional language of arrays. *LNCS 1300* (1990), 650–665.

[15] Jifeng, H., Josephs, M. B., and Hoare, C. A. R. A theory of synchrony and asynchrony. In *Programming Concepts and Methods*. Elsevier, 1990, pp. 459–473.

[16] Lacey, D., Jones, N. D., Wyk, E. V., and Frederiksen, C. C. Proving correctness of compiler optimizations by temporal logic. In *29th POPL* (Portland, OR, USA, 2002)

[17] Laird, J. A games semantics for idealized CSP. In *MFPS 17* (Aarhus, Denmark, 2001), ENTCS 45.

[18] Leperchey, B. Time and games. `http://www.pps.jussieu.fr/~leperche/pub/time.ps.gz`.

[19] Mason, I. A., and Talcott, C. L. Equivalence in functional languages with effects. *Journal of Functional Programming 1* (1991), 287–327.

[20] Milner, R. Processes: a mathematical model of computing agents. In *Logic Colloquium'73* (1975), H. E. Rose and J. C. Shepherdson, Eds., North-Holland, Amsterdam, pp. 157–174.

[21] Moran, A., and Sands, D. Improvement in a lazy context: An operational theory for call-by-need. In *26th POPL* (San Antonio, Texas, 1999).

[22] Nickau, H. Hereditarily sequential functionals. *LNCS 813* (1994).

[23] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic* (2001), *LNCS 2142*.

[24] O'Hearn, P. W., and Riecke, J. G. Kripke logical relations and PCF. *Information and Computation 120*, 1 (July 1995), 107–116.

[25] Pitts, A. M. Reasoning about local variables with operationally-based logical relations. In *11th LICS* (1996), Washington, USA.

[26] Plotkin, G. D. Lcf considered as a programming language. *TCS 5* (1977), 223–255.

[27] Reynolds, J. C. Idealized Algol and its specification logic. In *Tools and Notions for Program Construction* (Nice, France, 1981), D. Néel, Ed., Cambridge University Press, Cambridge, 1982,

[28] Roscoe, W. A. *Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[29] Sands, D. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Unversity of London, 1990.

[30] Sands, D. Operational theories of improvement in functional languages (extended abstract). In *Proceedings of the Fourth Glasgow Workshop on Functional Programming* (Skye, August 1991), Workshops in Computing Series.

[31] Sands, D. Improvement theory and its applications. In *Higher Order Operational Techniques in Semantics*, A. D. Gordon and A. M. Pitts, Eds., Publications of the Newton Institute. Cambridge University Press, 1998.

[32] Scott, D. S. A type-theoretical alternative to Cuch, Iswim, Owhy. Privately circulated memo, Oxford University, Oct. 1969. Published in *Theoretical Computer Science*, 121(1/2):411–440, 1993.

[33] van Stone, K. *A denotational approach to measuring complexity in functional programs*. PhD thesis, Carnegie Mellon University, 2003.

[34] Tarski, A. *Logic, Semantics, and Meta-Mathematics*. Oxford University Press, Oxford, 1956.

[35] Udding, J. T. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing 1(4)* (1986), 197–204.

[36] Wadler, P. Strictness analysis aids time analysis. In *15th POPL* (January 1988).