Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Data-Abstraction Refinement: A Game Semantic Approach*

Adam Bakewell[2], **Aleksandar Dimovski**[1], **Dan R. Ghica**[2], **Ranko Lazić**[1]

[1] Department of Computer Science, Univ. of Warwick, Coventry, CV4 7AL, UK
[2] School of Computer Science, Univ. of Birmingham, Birmingham, B15 2TT, UK

Received: date / Revised version: date

Abstract. This paper presents a semantic framework for data abstraction and refinement for verifying safety properties of open programs with integer types. The presentation is focused on an Algol-like programming language that incorporates data abstraction in its type system. We use a fully abstract game semantics in the style of Hyland and Ong and a more intensional version of the model that tracks nondeterminism introduced by abstraction in order to detect false counterexamples. These theoretical developments are incorporated in a new model-checking tool, MAGE, which implements efficiently the data-abstraction refinement procedure using symbolic and on-the-fly techniques.

Keywords: software model checking, abstraction refinement, game semantics, full abstraction

1 Introduction

Game semantics emerged in the past decade as a potent framework for modeling programming languages, with interesting applications to program verification and analysis [Ghi09]. It is an alternative (to Scott-Strachey) denotational semantics which constructs models of terms by looking at the ways in which a term can observably *interact* with its context (environment). In this approach, a kind of game is played by two participants. The first, Proponent, represents the term under consideration, while the second, Opponent, represents the environment in which the term is used. The two take turns to make moves, each of which is either a question (a demand for information) or an answer (a supply of information). Opponent always plays first. What these moves are, and when they can be played, is determined by the rules of each particular game. For example, in the game for integers, Opponent has a single move, the question "What is the number?", and Proponent can then respond by playing a number.

The game involved in modelling a function of type $\mathsf{expint} \rightarrow \mathsf{expint}$ is formed from "two copies of the game for expint", one for input, and one for output. In the output copy, Opponent can demand output and Proponent can provide it. In the input copy, the situation is reversed, Proponent demands input and Opponent provides it. A play in this game when Proponent is playing the *successor* function might look like this:

Opponent	"What is the output?"
Proponent	"What is the input?"
Opponent	"The input is 5"
Proponent	"The output is 6"

So, the successor function becomes a strategy for Proponent: "When Opponent asks for output, Proponent responds by asking for input; when Opponent provides input n, Proponent supplies $n + 1$ as output". This is the key idea in game semantics. Types are interpreted as *games*, and terms are interpreted as *strategies* for Proponent to respond to the moves Opponent can make. Strategies compose, much like traces for CSP-style processes, which makes it possible to define denotational models.

The key advantage of game semantics is the fact that it can produce concrete representations of models of open terms, i.e. terms with function-identifiers. The model of function identifiers is language dependent and it encapsulates *the most general behaviour* that can be exhibited at that type. Consider for example a procedural (call-by-name) programming language that has a base type of commands, com. The basic actions (*moves*) for commands are running (*run*) the command and observing its termination (*done*).

* This research was supported by the EPSRC (GR/S52759/01 and EP/D034906/1). The fourth author was also supported by a grant from the Intel Corporation.

A procedure *proc* : com → com can perform the following basic actions (moves): running (or calling) the procedure (run_p), observing the termination of the procedure ($done_p$), running (or calling) the argument (run_a) and observing the termination of the argument ($done_a$). The behaviour of the function, i.e. the possible sequences of actions of the kind described above, is determined by the language we are modelling, for example:

Sequential. If the programming language is sequential then the only way a procedure can use its argument is by calling it zero, one or more times. The behaviour of the procedure in this language, as a set of possible traces, is then representable by the regular expression: $run_p \cdot (run_a \cdot done_a)^* \cdot done_p$. [GM03]

Concurrent. If the programming language is concurrent then the procedure can use its argument by calling it an arbitrary number of times in interleaved fashion. The set of possible traces of a procedure in this language will be representable by the expression $run_p \cdot (run_a \cdot done_a)^{\circledast} \cdot done_p$, where $-^{\circledast}$ is the iterated closure of the shuffle operation. [GMO06]

Control. If the language has control operators that allow a command to abort execution then the set of traces is $run_p \cdot (run_a \cdot done_a + run_a)^* \cdot done_p$. [Lai97]

The behavioural models constructed using game semantics are *sound*, i.e. if two programs behave differently in some context then their game models will be different, and *complete*, i.e. the behaviours constructed using game-semantic models correspond to actual terms in the programming languages. Such models are called *fully abstract* and are the most precise models we can use for a programming language.

Abstraction refinement has proved to be one of the most effective methods of automatic verification of systems with very large state spaces, especially software systems. Current state-of-the art tools implementing abstraction refinement algorithms (e.g. [BR01,HJMS03]) combine model checking and theorem proving: model checking is used to verify whether an abstracted program satisfies a property, while theorem proving is used to refine the abstraction using the counterexamples discovered by model checking. Since abstractions are conservative, the safety of any abstracted program implies the safety of the concrete program. The converse is not true, and, for basic computability reasons, the refinement process may not terminate if the concrete program is safe and has an infinite state space.

This paper introduces a purely semantic approach to (data) abstraction refinement, based on game semantics [DGL05]. In order to keep the presentation focused, the main vehicle of our development is Idealized Algol (IA) [Rey81], an expressive programming language combining imperative features, locally-scoped variables and (call-by-name) procedures.

The verification procedure for open programs of second-order recursion-free IA is illustrated in Fig. 1. It checks whether a program fragment is unsafe, i.e. it may execute some designated unsafe actions. Using the verification procedure we can perform any code-level safety checks for errors such as buffer overruns or assertion violations.

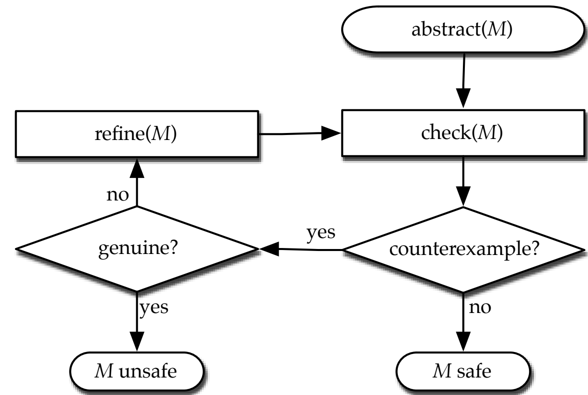


Fig. 1. Verification procedure

The procedure starts by transforming the concrete (input) program into the most abstract version of it, where all integers are abstracted to singleton sets. Then the game-semantic model of the abstracted program is extracted. Since our abstractions are safe, any abstracted program is an over-approximation of the concrete program, so if no designated unsafe action is reachable, the procedure terminates with answer *SAFE*. Otherwise, the counterexamples, i.e. traces to unsafe actions, are analysed and classified as either genuine, which correspond to execution traces in the concrete program, or (potentially) spurious, which can be introduced due to abstraction. If genuine counterexamples exist the program is deemed *UNSAFE*, otherwise the spurious counterexamples are used to refine the abstractions. The procedure is then repeated on the refined abstracted program. The abstraction refinement procedure is a semi-algorithm: it terminates and reports a genuine counterexample for unsafe programs, but it may diverge for safe programs if the integer type is taken to be infinite. For finite but large integer types the abstraction-refinement cycle can be seen as a heuristic meant to avoid the construction of unnecessarily large models.

Example. We illustrate the technique with a simple example which will show the operations of the abstraction-refinement cycle and make clear the role of fully abstract models in the verification of open terms. Consider this term, with procedure identifiers f and c:

$$\begin{aligned}&\mathsf{newint}\ x := 0\ \mathsf{in}\\&\mathsf{let}\ p\ \mathsf{be}\ x{+}{+}; c; x{-}{-}; \mathsf{assert}(x = 0)\ \mathsf{in}\\&f(p)\end{aligned}$$

The verification question is, can the assertion fail—does the special action associated to a failing assertion occur

in any computation? The answer is language-dependent, for example:

Sequential. If f and c can only exhibit sequential behaviours then the term is safe. On each call to its argument, x will be incremented then decremented, always being 0 for the assertion.

Concurrent. If f can start evaluating concurrently two threads of p then it is possible that the first thread increments to 1 then waits, so the second threads increments to 2, then decrements to 1. In this case the assertion in the second thread may fail.

Control. If c can abort execution then the first call to p may just increment x to 1, so that a second call to p may cause the assertion to fail.

The safety of the program is contingent on the behaviours of procedure identifiers f and c, which are language-dependent. Game semantics is a trace-like semantics that assigns such meanings to procedure identifiers so that open terms can be modelled correctly.

Suppose that we are in the sequential setting and we want to prove the safety of the program using data abstraction refinement. A data abstraction is any finite partition of the set of integers. We can start by using one partition equal to $\mathbb{Z}$, the coarsest abstraction. Let us call this unique *abstract value* $*$, and it satisfies the following properties:

$$
\begin{aligned}
* + 1 &= * \\
* - 1 &= * \\
* = 0 &\in \{\text{true}, \text{false}\}.
\end{aligned}
$$

Because of the non-deterministic evaluation it is possible to have a sequence of actions leading to a failed assertion. However, we can detect the fact that this trace was made possible only by the non-deterministic evaluation of the assertion test, due to *over-abstraction*.

We must use a more precise abstraction, for example consisting of partitions $\overline{N} = \{n < 0\}, \overline{0} = \{0\}, \overline{1} = \{1\}, \overline{P} = \{n > 1\}$. In this case, using interval-style arithmetic we can trace precisely the values of x so that $x = 0$ is evaluated precisely to true.

Using a precise, i.e. fully abstract, model is important in the refinement process because it allows the identification of spurious error traces. If the model of the language itself was imprecise it would be impossible to tell whether error traces are genuine, caused by approximation errors or caused by an imprecise model.

Contributions. Several features of game semantics make it very promising for software verification:

Modularity. To our knowledge, examples such as the one described above, cannot be generally handled by known inter-procedural abstraction-refinement techniques. [CC02] cogently advocates a need for such techniques, and we believe that we are meeting the challenge, although our approach is technically different.

Completeness and correctness. A concrete representation of a fully abstract (sound and complete) semantic model is guaranteed to be accurate and is set on a firm theoretical foundation.

Compositionality. The semantic model is denotational, i.e. defined recursively on the syntax, therefore the model of a larger program is constructed from the models of its constituting subprograms. This entails an ability to model program fragments, containing non-locally defined procedures as in the example above. This feature is the key for achieving *scalability*, i.e. the possibility to break up a larger program into smaller subprograms which can be modelled and verified independently.

Simplicity. The semantics-direct construction of an abstracted model is conceptually simpler than a syntactic abstraction. It requires no static analysis of the target program. Each construct of the language is abstracted separately and the abstracted program model is derived compositionally.

Flexibility. Each occurrence of each programming language construct is abstracted separately, which means that in the abstract syntax tree some branches can use different abstraction schemes than others. This level of flexibility virtually removes the need for other syntax-based heuristics such as slicing because branches of the syntax tree that could be removed by slicing will have a very small model anyway. This comes for free from the counterexample analysis.

Implementation. In this paper, we also describe MAGE [BG08], a tool which implements efficiently the abstraction refinement procedure above, using symbolic representations and on-the-fly modelling.[1]

2 IA and Its Game Model

Our prototypical procedural language is a simply-typed call-by-name lambda calculus with basic types of expressions over data-types D such as booleans or integers ($\mathsf{exp}D$), assignable variables ($\mathsf{var}D$) and commands (com). We denote the basic types by B and the function types by T. *Assignable* variables, storing integers, form the state while commands change the state. In addition to function definition ($\lambda x : B.M$) and application (FM), other terms of the language are conditionals, uniformly applied to any type, ($\mathsf{if}\, B \,\mathsf{then}\, M \,\mathsf{else}\, N$), iteration ($\mathsf{while}\ B\ \mathsf{do}\ C$), constants (integers, booleans) and arithmetic-logic operators; we also have command-type terms which are the standard imperative actions: dereferencing (if necessary made explicit in the syntax, $!V$), assignment ($V := N$), sequencing ($C; M$), no-op (skip) and local variable block ($\mathsf{new}\, x \,\mathsf{in}\, M$). We write $M : T$ to indicate that term M has type T.

[1] MAGE is available from `http://www.cs.bham.ac.uk/research/projects/mage/`.

If the programming language is restricted to recursion-free first-order procedures, (more precisely, we restrict types to $T ::= B \mid B \to T$) and finite data-types then the Abramsky-McCusker fully abstract game model for this language [AM96] has a regular-language representation [GM00]. Here we adopt a slightly different but equivalent presentation [AGMO04] because it is more uniform and more compact.

The syntax of IA is given using *typing judgements* of the form $\Gamma \vdash M : T$ where Γ is a *type assignment* of the form $x_0 : T_0, \ldots, x_k : T_k$. The main rule for term formation is that of function application:

$$\frac{\Gamma \vdash F : B \to T \qquad \Gamma \vdash M : B}{\Gamma \vdash F(M) : T}\ .$$

It will actually be convenient to break-up this rule into two finer-grained rules, *linear* function application and *contraction*:

$$\frac{\Gamma \vdash F : B \to T \qquad \Delta \vdash M : B}{\Gamma, \Delta \vdash F(M) : M}$$

$$\frac{\Gamma, x : T, y : T \vdash M : T'}{\Gamma, z : T \vdash M[z/x, z/y] : T'}\ .$$

The linear application rule prohibits sharing of identifiers between a function and its argument, but different identifiers can be given the same name if so desired via an explicit contraction. These two rules taken together are just as expressive as conventional function application but their presence simplifies the presentation of the semantics.

A final structural rule allows the swapping of identifiers in Γ:

$$\frac{\Gamma, x_1 : T_1, x_2 : T_2, \Delta \vdash M : T}{\Gamma, x_2 : T_2, x_1 : T_1, \Delta \vdash M : T}\ .$$

The rule (axiom) for free identifiers is

$$\Gamma, x : T \vdash x : T.$$

Terms are formed of free identifiers, constants and language constructors given in a functional form:

$$\begin{aligned}
&\mathsf{true}, \mathsf{false} : \mathsf{expbool}\\
&0, 1, \ldots : \mathsf{expint}\\
&\mathsf{seq} : \mathsf{com} \to \mathsf{com} \to \mathsf{com}\\
&\mathsf{asg} : \mathsf{varD} \to \mathsf{expD} \to \mathsf{com}\\
&\mathsf{der} : \mathsf{varD} \to \mathsf{expD}\\
&\mathsf{if} : \mathsf{expbool} \to \mathsf{com} \to \mathsf{com} \to \mathsf{com}\\
&\mathsf{while} : \mathsf{expbool} \to \mathsf{com} \to \mathsf{com}\\
&\mathsf{plus}, \mathsf{minus}, \mathsf{times}, \mathsf{div} : \mathsf{expint} \to \mathsf{expint} \to \mathsf{expint}\\
&\mathsf{eq}, \mathsf{neq}, \mathsf{lt}, \mathsf{gt} : \mathsf{expint} \to \mathsf{expint} \to \mathsf{expbool}\\
&\mathsf{and}, \mathsf{or}, \mathsf{not} : \mathsf{expbool} \to \mathsf{expbool} \to \mathsf{expbool}.
\end{aligned}$$

Finally, we give local variable binding via a special rule:

$$\frac{x : \mathsf{var}D, \Gamma \vdash M : \mathsf{com}}{\Gamma \vdash \mathsf{new}D\,x\,\mathsf{in}\,M : \mathsf{com}}\ .$$

We sometimes used initialised variables $\mathsf{new}D\,x := k\,\mathsf{in}\,M$ as syntactic sugar for $\mathsf{new}D\,x\,\mathsf{in}\,x := k; M$.

A program can be presented either in the "functional" form or in the more traditional form, as is convenient depending on context. For example,

$$x := 0; x := x + 1$$

is just syntactic sugar for

$$\mathsf{seq}(\mathsf{asg}\ x\ 0)(\mathsf{asg}\ x(\mathsf{add}(\mathsf{der}\ x)1)).$$

A final point on notation. For simplicity we will sometimes write types $B_1 \to B_2 \to \cdots \to B_k \to B$ as $B_1, B_2, \ldots, B_k \to B$.

Extended Regular Expressions. Terms are interpreted by languages over alphabets of moves $\mathcal{A}$. The languages, denoted by $\mathcal{L}(R)$, are specified using regular expressions R extended with intersection and images under homomorphism. The standard regular expression operators consist of the empty language $\emptyset$, the empty sequence ϵ, concatenation $R \cdot S$, union $R + S$, Kleene star R^*, intersection $R \cap S$ and the elements of the alphabet taken as sequences of unit length.

The languages denoted by these extended regular expressions will be taken to be the *prefix-closure* of the regular language conventionally denoted by regular expressions:

$$\begin{aligned}
\mathcal{L}(\epsilon) &= \{\epsilon\}\\
\mathcal{L}(a) &= \{\epsilon, a\}\\
\mathcal{L}(R \cdot S) &= \mathcal{L}(R) \cup \{s \cdot t \mid s \in \mathcal{L}_{max}(R), t \in \mathcal{L}(S)\}\\
\mathcal{L}(R + S) &= \mathcal{L}(R) \cup \mathcal{L}(S)\\
\mathcal{L}(R^1) &= \mathcal{L}(R)\\
\mathcal{L}(R^{n+1}) &= \mathcal{L}(R \cdot R^n)\\
\mathcal{L}(R^*) &= \bigcup_{i \in \mathbb{N}} \mathcal{L}(R^i)\\
\mathcal{L}(R \cap S) &= \mathcal{L}(R) \cap \mathcal{L}(S).
\end{aligned}$$

By $\mathcal{L}_{max}(R)$ we denote the set of maximal words in a language, i.e. those words that are not a proper prefix of any other word in the language. It can be easily shown that all the operations above produce prefix-closed languages if applied to languages that already have this property.

We also use the additional constructs of direct $\phi(R)$ and inverse image $\phi^{-1}(R)$ under homomorphism ϕ:

$$\begin{aligned}
\mathcal{L}(\phi(R)) &= \{s \mid s = \phi(t), t \in \mathcal{L}(R)\}\\
\mathcal{L}(\phi^{-1}(R)) &= \{t \mid s = \phi(t), s \in \mathcal{L}(R)\}
\end{aligned}$$

Any extended regular expression over finite alphabets constructed from the operations described above denotes a regular language, which can be recognized by a finite automaton which can be effectively constructed from the

regular expression [HU79]. Inverse image under homomrophism preserves prefix-closure while direct image preserves prefix-closure if it maps any symbol in another symbol or ϵ. In our subsequent definition we only use such homomorphisms.

We will often use the disjoint union of two alphabets to create a larger alphabet: $\mathcal{A}_1+\mathcal{A}_2$. The disjoint union gives rise to the canonical homomorphisms:

$$\mathcal{A}_1 \underset{\text{outl}}{\overset{\text{inl}}{\rightleftarrows}} \mathcal{A}_1 + \mathcal{A}_2 \underset{\text{outr}}{\overset{\text{inr}}{\leftrightarrows}} \mathcal{A}_2 \ .$$

The homomorphisms have to be such that:

$$\begin{aligned} \text{outl}(\text{inl}\, a) &= a & \qquad \text{outr}(\text{inl}\, a) &= \epsilon \\ \text{outl}(\text{inr}\, b) &= \epsilon & \qquad \text{outr}(\text{inr}\, b) &= b \end{aligned}$$

If $\phi : \mathcal{A} \to \mathcal{B}^*$ and $\psi : \mathcal{C} \to \mathcal{D}^*$ are homomorphisms then we define their sum $\phi + \psi : \mathcal{A} + \mathcal{C} \to (\mathcal{B} + \mathcal{D})^*$ as $(\phi + \psi)(\text{inl}\, a) = \text{inl}(\phi a)$, respectively $(\phi + \psi)(\text{inr}\, c) = \text{inr}(\psi c)$.

Definition 1 (Composition). If R is a regular expression over alphabet $\mathcal{A}+\mathcal{B}$ and S a regular expression over alphabet $\mathcal{B} + \mathcal{C}$ we define the *composition* $R \circ S$ as the regular expression

$$R \circ S = \text{out}\big(\text{out}_1^{-1}(R) \cap \text{out}_2^{-1}(S)\big),$$

over alphabet $\mathcal{A} + \mathcal{C}$, with homomorphisms

$$\mathcal{A} + \mathcal{B} \underset{\text{out}_1}{\overset{\text{in}_1}{\rightleftarrows}} \mathcal{A} + \mathcal{B} + \mathcal{C} \underset{\text{out}_2}{\overset{\text{in}_2}{\leftrightarrows}} \mathcal{B} + \mathcal{C}$$

and

$$\mathcal{A} + \mathcal{C} \underset{\text{out}}{\overset{\text{in}}{\rightleftarrows}} \mathcal{A} + \mathcal{B} + \mathcal{C} \ .$$

It is a simple exercise to show that regular language composition also preserves prefix closure of languages.

Alphabets. We interpret each type T by a language over an alphabet $\mathcal{A}[\![T]\!]$, containing the *moves* from the game model. For basic types B the basic type alphabets are:

$$\begin{aligned} \mathcal{A}[\![\mathsf{exp}D]\!] &= \{q\} \cup D \\ \mathcal{A}[\![\mathsf{var}D]\!] &= \{\mathit{read}, \mathit{ok}\} \cup \{n, \mathit{write}(n) \mid n \in D\} \\ \mathcal{A}[\![\mathsf{com}]\!] &= \{\mathit{run}, \mathit{done}\}, \end{aligned}$$

where D is a data-type. Alphabets of function types are defined by $\mathcal{A}[\![B \to T]\!] = \mathcal{A}[\![B]\!] + \mathcal{A}[\![T]\!]$.

The alphabet of moves indicates all observable actions at that type. For example, for commands $\mathcal{A}[\![\mathsf{com}]\!]$ the possible observations are *run*, running the command, and *done*, termination. For expressions $\mathcal{A}[\![\mathsf{exp}D]\!]$ the observations are evaluating the expression q and the expression producing a value $n \in D$. For assignable variables we have actions for writing to the variable, $\mathit{write}(n), n \in D$, acknowledged by *ok*, and reading from the variable *read* a value $n \in D$. For function types, e.g. $\mathsf{com} \to \mathsf{com}$, the alphabet will consist of moves associated with running $\text{inr}(\mathit{run})$ and terminating $\text{inr}(\mathit{done})$ the function body and, respectively, its argument $\text{inl}(\mathit{run}), \text{inl}(\mathit{done})$.

Semantics. We will give semantic evaluation rules for terms in β-normal forms. If a term has β-redexes (i.e. calls to defined functions) we simply reduce them syntactically by substitution (i.e. "inlining"). This is in order to streamline the presentation and focus on the verification of open terms, where application is only to procedure-identifiers.

The most distinctive feature of game semantics is giving a concrete interpretation to free identifiers. This interpretation is captured by the idea of *legal play* which captures all legal behaviours at all types. The set of legal plays for each type $\mathcal{P}[\![T]\!]$ can be given as a regular expression over its alphabet $\mathcal{A}[\![T]\!]$.

$$\begin{aligned} \mathcal{P}[\![\mathsf{com}]\!] &= \mathit{run}\ \mathit{done} \\ \mathcal{P}[\![\mathsf{exp}D]\!] &= \sum_{n \in D} q\, n \\ \mathcal{P}[\![\mathsf{var}D]\!] &= \sum_{m,n \in D} \mathit{write}(m)\ \mathit{ok} + \mathit{read}\, n. \end{aligned}$$

This extends to function types as follows:

$$\begin{aligned} &\mathcal{P}[\![B_1, \ldots, B_k \to \mathsf{com}]\!] \\ &\qquad = \text{in}_{k+1}(\mathit{run})\Big(\sum_{i=1}^{k} \mathit{in}_i\big(\mathcal{P}[\![B_i]\!]\big)\Big)^* \text{in}_{k+1}(\mathit{done}). \end{aligned}$$

For function types of the form $B_1, \ldots, B_k \to \mathsf{exp}D$ and $B_1, \ldots, B_k \to \mathsf{var}D$ the definitions are similar [AGMO04], where

$$\begin{aligned} \text{in}_i : \mathcal{A}[\![B_i]\!] &\to \left(\sum_{i=1}^{k} \mathcal{A}[\![B_i]\!] + \mathcal{A}[\![\mathsf{com}]\!]\right) \\ \text{in}_{k+1} : \mathcal{A}[\![\mathsf{com}]\!] &\to \left(\sum_{i=1}^{k} \mathcal{A}[\![B_i]\!] + \mathcal{A}[\![\mathsf{com}]\!]\right) \end{aligned}$$

The meaning of a sequential procedure of type $\mathsf{com} \to \mathsf{com}$ used as an example in the Introduction uses this definition, where concretely $\text{inl}\, x = x_p$ and $\text{inr}\, x = x_a$.

These regular expressions denote behaviour that depends on the syntax of the language fragment both in terms of constructors and in terms of functional order. At second-order [Ong02] and above [HO95], or in concurrent settings [GM08], legal plays become more complicated and cannot be defined using regular expressions only.

A legal set of plays $\mathcal{P}$ over alphabet $\mathcal{A}$ gives rise to a so-called *copy-cat* behaviour $\mathcal{K}$ over alphabet $\mathcal{A} + \mathcal{A}$ in which each symbol in each word in $\mathcal{P}$ is replaced by sequences $\text{inr}(m)\text{inl}(m)$ and $\text{inl}(m)\text{inr}(m)$ in alternating fashion [GB09] so that, for example:

$$\begin{aligned} \mathcal{K}[\![\mathsf{com}]\!] &= \text{inr}(\mathit{run})\, \text{inl}(\mathit{run})\, \text{inl}(\mathit{done})\, \text{inr}(\mathit{done}) \\ \mathcal{K}[\![\mathsf{exp}D]\!] &= \sum_{n \in D} \text{inr}(q)\, \text{inl}(q)\, \text{inl}(n)\, \text{inr}(n) \end{aligned}$$

and so on. Copy-cat behaviours play an important role in game semantics in the representation of *structural* constructions such as identity and projection. The name of

the behaviour is apt because it consists of simply replicating actions with two distinct tags.

We interpret terms using an evaluation function $[\![-]\!]$ mapping a (beta-normal) term $\Gamma \vdash M : T$ into a regular language over alphabet $\sum_{x_i:T_i\in\Gamma} \mathcal{A}[\![T_i]\!] + \mathcal{A}[\![T]\!]$. The evaluation function is defined by recursion on the syntax.

Identifiers. Identifiers are interpreted by a copy-cat behaviour at that type:

$$[\![\Gamma, x : T \vdash x : T]\!] = (\alpha \circ \mathrm{inr})(\mathcal{K}[\![T]\!]),$$

where inr is an injection and α the associativity isomorphism (necessary to properly realign the tags):

$$\mathrm{inr} : \mathcal{A}[\![T]\!] + \mathcal{A}[\![T]\!] \to \mathcal{A}[\![\Gamma]\!] + (\mathcal{A}[\![T]\!] + \mathcal{A}[\![T]\!])$$
$$\alpha : \mathcal{A}[\![\Gamma]\!] + (\mathcal{A}[\![T]\!] + \mathcal{A}[\![T]\!]) \xrightarrow{\sim} (\mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![T]\!]) + \mathcal{A}[\![T]\!].$$

Linear application.

$$[\![\Gamma, \Delta \vdash FM : T]\!] = \alpha_3(\alpha_1([\![\Gamma \vdash F : B \to T]\!]) \circ \alpha_2([\![\Delta \vdash M : B]\!]^*)),$$

where α_i are the following isomorphisms formed by compositions of associativity and commutativity (necessary to realign the tags properly):

$$\alpha_1 : \mathcal{A}[\![\Gamma]\!] + (\mathcal{A}[\![B]\!] + \mathcal{A}[\![T]\!]) \xrightarrow{\sim} (\mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![T]\!]) + \mathcal{A}[\![B]\!]$$
$$\alpha_2 : \mathcal{A}[\![\Delta]\!] + \mathcal{A}[\![B]\!] \xrightarrow{\sim} \mathcal{A}[\![B]\!] + \mathcal{A}[\![\Delta]\!]$$
$$\alpha_3 : \mathcal{A}[\![\Gamma]\!] + (\mathcal{A}[\![T]\!] + \mathcal{A}[\![\Delta]\!]) \xrightarrow{\sim} (\mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![\Delta]\!]) + \mathcal{A}[\![T]\!].$$

Regular language composition $- \circ -$ is defined as before, so the synchronisation (intersection) must act on the alphabet associated with type B of function F's argument.

Contraction.

$$[\![z : T', \Gamma \vdash M[z/x, z/y] : T]\!] = (\delta + \mathrm{id})\big([\![x : T', y : T', \Gamma \vdash M : T]\!]\big),$$

where id is the identity on $\mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![T]\!]$. The homomorphism $\delta : \mathcal{A}[\![T']\!] + \mathcal{A}[\![T']\!] \to \mathcal{A}[\![T']\!]$ de-tags the two occurrences of alphabet $\mathcal{A}[\![T]\!]'$:

$$\delta(\mathrm{inl}\, m) = m, \qquad \delta(\mathrm{inr}\, m) = m,$$

therefore identifying actions associated with x and y. Note that this interpretation is specific to ground and first-order types. In higher-order types this interpretation of contraction by un-tagging will result in ambiguities.

Commutativity. For the sake of completeness, we also need a rule for swapping the order of two identifiers:

$$[\![\Gamma, x : T, x' : T', \Delta \vdash M : U]\!] = \alpha([\![\Gamma, x' : T', x : T, \Delta \vdash M : U]\!]),$$

where

$$\alpha : \mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![T]\!] + \mathcal{A}[\![T']\!] + \mathcal{A}[\![\Delta]\!] + \mathcal{A}[\![U]\!] \xrightarrow{\sim} \mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![T']\!] + \mathcal{A}[\![T]\!] + \mathcal{A}[\![\Delta]\!] + \mathcal{A}[\![U]\!]$$

is the obvious isomorphism.

Block Variables. We can represent the read-write causal behaviour in a variable by the following regular expression:

$$\mathcal{V} = \Big(\sum_{n\in D} \mathit{write}(n)\ \mathit{ok}\,(q\,n)^*\Big)^*.$$

Intuitively, this regular expression describes the sequential behaviour of a memory cell: if a value n is written, then the same value is read back until the next write, and so on. Then,

$$[\![\Gamma \vdash \mathsf{new}D\, x\, \mathsf{new}\, M : \mathsf{com}]\!] = (\mathrm{inl}\ \mathcal{V}) \circ [\![x : \mathsf{var}, \Gamma \vdash M : \mathsf{com}]\!].$$

Constants. To simplify the definition of languages for the interpretation of constants we use particular injections, tagging each type occurrence with its index in the typing judgement of the constant.

The interpretation of constants is:

$$[\![k : \mathsf{exp}D]\!] = q\, k,$$
$$[\![\mathsf{op} : \mathsf{exp}D, \mathsf{exp}D \to \mathsf{exp}D']\!] = \sum_{\substack{m,n\in D, p\in D'\\ p = m\ op\ n}} q_3\, q_1\, m_1\, q_2\, n_2\, p_3$$
$$[\![\mathsf{der} : \mathsf{var}D \to \mathsf{exp}D]\!] = \sum_{n\in D} q_2\, q_1\, n_1\, n_2$$
$$[\![\mathsf{asg} : \mathsf{var}D, \mathsf{exp}D \to \mathsf{com}]\!] = \sum_{n\in D} \mathit{run}_3\, q_2\, n_2\, \mathit{write}(n)_1\, \mathit{ok}_1\, \mathit{done}_3$$
$$[\![\mathsf{if} : \mathsf{expbool}, \mathsf{com}, \mathsf{com} \to \mathsf{com}]\!] = \mathit{run}_4\, q_1\, t_1\, \mathit{run}_2\, \mathit{done}_2\, \mathit{done}_4 + \mathit{run}_4\, q_1\, f_1\, \mathit{run}_3\, \mathit{done}_3\, \mathit{done}_4$$
$$[\![\mathsf{seq} : \mathsf{com}, \mathsf{com} \to \mathsf{com}]\!] = \mathit{run}_3\, \mathit{run}_1\, \mathit{done}_1\, \mathit{run}_2\, \mathit{done}_2\, \mathit{done}_3$$
$$[\![\mathsf{while} : \mathsf{expbool}, \mathsf{com} \to \mathsf{com}]\!] = \mathit{run}_3\big(q_1\, t_1\, q_2\, \mathit{done}_2\big)^* q_1\, f_1\, \mathit{done}_3.$$

The operator op ranges over the usual arithmetic-logic operators, and op is its standard interpretation.

Note that branching and sequential composition can be defined for arbitrary expressions as well as for commands. It is convenient to use in examples a non-terminating command $\mathsf{div} \stackrel{\mathsf{def}}{=} \mathsf{while\, true\, skip}$:

$$[\![\mathsf{div} : \mathsf{com}]\!] = \mathit{run}.$$

The semantics given in this section is fully abstract (in the precise sense defined in the Introduction) for the programming language fragment presented if the regular-language semantics is quotiented by the set of its so

called *complete plays* [AM96, GM03]. The technical definition requires some other notions that, for the sake of simplicity, we omitted in this presentation but the intuition is simple: a complete play is a play corresponding to a terminating computation. For example,

$$[\![\mathsf{div} : \mathsf{com}]\!]_{\text{complete}} = \emptyset.$$

Theorem 1. *Two terms M, N are observationally equivalent $\Gamma \vdash M \equiv_T N$ if and only if their sets of complete plays are equal $[\![\Gamma \vdash M : T]\!]_{\text{complete}} = [\![\Gamma \vdash N : T]\!]_{\text{complete}}$.*

This property means that the denotational semantics coincides in some technical sense with the operational definition of the language. We have already explained in the definition why this is an essential property in modelling and verifying open terms, especially in the context of abstraction refinement. The technical details, including the definition of operational observation equivalence are beyond the scope of this paper so the interested reader is referred to the existing literature [AM96, GM03].

Given an alphabet $\mathcal{E}$ of selected actions considered *unsafe*, a term is said to be *safe* if it does not contain any such action in its denotation. The rest of the actions in the semantics of a term are said to be *safe*.

Definition 2 (Safety). Let $\mathcal{E} + \mathcal{S} = \mathcal{A}[\![\Gamma \vdash M : T]\!]$. Term M is said to be $\mathcal{E}$-safe if $\mathrm{outl}[\![\Gamma \vdash M : T]\!] = \emptyset$.

Observation. The reason for using the unquotiented prefix-closed model rather than the complete-play model which is used in other papers (e.g. [AGMO04]) is to avoid the situation where non-termination hides unsafe behaviour. For example

$$[\![\mathsf{abort} : \mathsf{com} \vdash \mathsf{abort}; \mathsf{div}]\!] = run\, run_{abort}\, done_{abort}$$

but

$$[\![\mathsf{abort} : \mathsf{com} \vdash \mathsf{abort}; \mathsf{div}]\!]_{comp} = \emptyset.$$

The more intensional, prefix-closed set of plays is not $\mathcal{A}[\![\mathsf{abort}]\!]$-safe but the more extensional maximal-play semantics is, because the unsafe symbols do not appear in any *terminating* computations. This distinction, akin to the difference between total and partial correctness, is perhaps not very important, but we feel that for practical purposes it is preferable to be able to check for illegal actions even if they occur in non-terminating program runs.

2.1 A Simple Example

This simple example illustrates the way the game-based model works. It is a toy abstract data type (ADT): a switch that can be flicked on, with implementation:

$$\begin{aligned} &client : \mathsf{com} \to \mathsf{expint} \to \mathsf{com} \vdash \\ &\quad \mathsf{newint}\, v := 0\, \mathsf{in} \\ &\quad \mathsf{let}\, set = (v := 1)\, \mathsf{in} \\ &\quad \mathsf{let}\, get = (!v)\, \mathsf{in} \\ &\quad client(set, get) : \mathsf{com}. \end{aligned}$$

The local definitions of *set, get* are given only for clarity and they can be inlined before the semantic model is constructed.

The code consists of local integer variable v, storing the state of the switch, together with functions *set*, to flick the switch on, and *get*, to get the state of the switch. The initial state of the switch is *off*. The non-local, undefined, identifier *client* is declared at the left of the turnstile $\vdash$. It takes a command and an expression-returning functions as arguments. It represents, intuitively, "the most general context" in which this ADT can be used.

A key observation about the model is that the *internal state* of the program is abstracted away, and only the observable actions, of the *nonlocal* entity *client*, are represented, insofar as they contribute to terminating computations 2.

Notice that no references to v, *set*, or *get* appear in the model. The model is only that of the possible behaviours of the *client*: whenever the client is executed, if it evaluates its second argument (get the state of the switch) it will receive the value 0 as a result; if it evaluates the first argument (set the switch on), one or more times, then the second argument (get the state of the switch) will always evaluate to 1. The model does not, however, assume that *client* uses its arguments, or how many times or in what order.

2.2 Data Abstraction

Although the semantics given above can be represented, for finite data types, as a finite state system, in practice the large size of the data types makes a concrete encoding impractical. However, because the semantics is parametrised by the choice of data types D, support for abstract data types is immediate. In this way we can easily obtain approximate representations not only for large data types but also for infinite ones.

Supposed that the data type D is a finite partition of the set of integers (either the infinite set of integers or a large finite subset) given by

$$D = \big\{\{n < 0\}, \{0\}, \{n > 0\}\big\} \stackrel{\text{def}}{=} \{(<0), 0, (>0)\}. \quad (1)$$

The only semantic rules in which data abstraction plays a role at all are those for operators:

$$[\![\mathsf{op} : \mathsf{exp}D, \mathsf{exp}D \to \mathsf{exp}D']\!] = \sum_{\substack{m,n \in D, p \in D' \\ p = m\ op\ n}} q_3\, q_1\, m_1\, q_2\, n_2\, p_3,$$

where now the interpretation *op* of the operator op must be generalized to operate over partitions of the set of

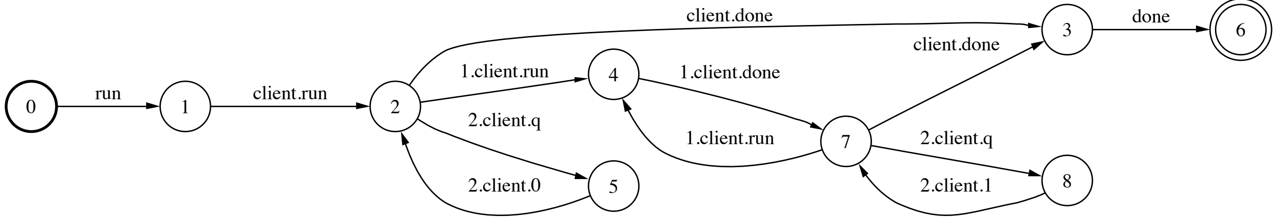


Fig. 2. A simple switch

integers. These interpretation can be obtained in a standard way using well established techniques from abstract interpretation [Cou96]. For example, the addition operator for the data abstraction D in Eq. 1 is, concretely:

$$\begin{aligned}[\![+ : \mathsf{exp}D, \mathsf{exp}D \to \mathsf{exp}D]\!] = &\ q_3\, q_1\, (<0)_1\, q_2\, (<0)_2\, (<0)_3 \\ &+ q_3\, q_1\, (<0)_1\, q_2\, 0_2\, (<0)_3 \\ &+ q_3\, q_1\, (<0)_1\, q_2\, (>0)_2\, (<0)_3 \\ &+ q_3\, q_1\, (<0)_1\, q_2\, (>0)_2\, 0_3 \\ &+ q_3\, q_1\, (<0)_1\, q_2\, (>0)_2\, (>0)_3 \\ &+ q_3\, q_1\, 0_1\, q_2\, (<0)_2\, (<0)_3 \\ &+ q_3\, q_1\, 0_1\, q_2\, 0_2\, 0_3 \\ &+ q_3\, q_1\, 0_1\, q_2\, (>0)_2\, (>0)_3 \\ &+ q_3\, q_1\, (>0)_1\, q_2\, (<0)_2\, (<0)_3 \\ &+ q_3\, q_1\, (>0)_1\, q_2\, (<0)_2\, 0_3 \\ &+ q_3\, q_1\, (>0)_1\, q_2\, (<0)_2\, (>0)_3 \\ &+ q_3\, q_1\, (>0)_1\, q_2\, 0_2\, (>0)_3 \\ &+ q_3\, q_1\, (>0)_1\, q_2\, (>0)_2\, (>0)_3. \end{aligned}$$

Note the introduction of nondeterminism in addition over abstracted integers: $(<0) + (>0)$ can yield as values any of $(<0), 0, (>0)$.

Also note that there is no need to use the same data abstraction everywhere in a program, or even everywhere in the types of an operator. The arguments of any operator can be independently abstracted, as can be the return type.

Finally, if we allow unrestricted use of multiple abstractions in a program we must have a way to "glue" together terms of non-matching abstractions, using approximating copy-cat-like behaviours defined by the following regular expressions:

$$[\![\overline{\mathcal{K}}_{D,D'} : \mathsf{exp}D \to \mathsf{exp}D']\!] = \sum_{\substack{m\in D, n\in D\\ m\cap n\neq\emptyset}} q_2\, q_1\, m_1\, n_2.$$

A similar approximated copy-cat can be defined for $\mathsf{var}D$.

Since abstract data types consist of sets of concrete values, the approximating copy-cat matches any abstract value in D with all abstract values in D' such that there is an overlap between them. Note that this also allows the use of concrete values in abstracted operators, e.g., using the data abstraction D in Eqn. 1 we have:

$$\begin{aligned}&[\![\overline{\mathcal{K}}_{\mathsf{int},D} : \mathsf{expint} \to \mathsf{exp}D]\!] \\ &\quad = \sum_{n<0} q_2\, q_1\, n_1\, (<0)_2 + q_2\, q_1\, 0_1\, 0_2 + \sum_{m>0} q_2\, q_1\, m_1\, (>0)_2.\end{aligned}$$

This allows, for example, having "increment by one" in an abstract domain of integers. Using the same D, directly from the definitions we get

$$\begin{aligned}&[\![(+1) : \mathsf{exp}D \to \mathsf{exp}D]\!] \\ &= [\![\mathsf{add}(\overline{\mathcal{K}}_{\mathsf{int},D}\ 1) : \mathsf{exp}D \to \mathsf{exp}D]\!] \\ &= q_2\, q_1\, ((<0)_1\, (<0)_2 + (<0)_1\, 0_2 + 0_1\, (>0)_2 + (>0)_1\, (>0)_2).\end{aligned}$$

The copy-cat-like approximations can be introduced automatically whenever the data type of an argument and its function do not match.

We shall often use the following abstract data types:

$$\begin{aligned}[a, b] &= \big\{\{m < a\}, \{a\}, \ldots, \{b\}, \{n > b\}\big\} \\ &= \{(<a), a, \ldots, b, (>b)\}.\end{aligned}$$

We also write abstracted operators as

$$\mathsf{op} : \mathsf{exp}D_1, \mathsf{exp}D_2 \to \mathsf{exp}D_3 \stackrel{def}{=} \mathsf{op}_{D_1,D_2,D_3},$$

or just op_D if $D_1 = D_2 = D_3 = D$.

2.3 Interaction Game Semantics

Let us define an alternative, more intensional, semantics, where moves which interact are not hidden. Consider composing $R : \mathcal{A} + \mathcal{B}$ and $S : \mathcal{B} + \mathcal{C}$ to obtain $R \bullet S : \mathcal{A} + \mathcal{B} + \mathcal{C}$ in a way that is similar to $- \circ -$, except for not removing moves in $\mathcal{B}$:

$$R \bullet S = \mathrm{out}_1^{-1}(R) \cap \mathrm{out}_2^{-1}(S).$$

We call a model constructed using this composition rule rather than $-\circ-$ an *interaction semantics*, and its building blocks interaction plays and interaction strategies; we denote the interaction semantics by $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$. It will be a regular expression over a larger alphabet, determined not just by the type signature of M but by its syntactic structure as well; it will be the disjoint union of all alphabets of all sub-terms of M. The conventional

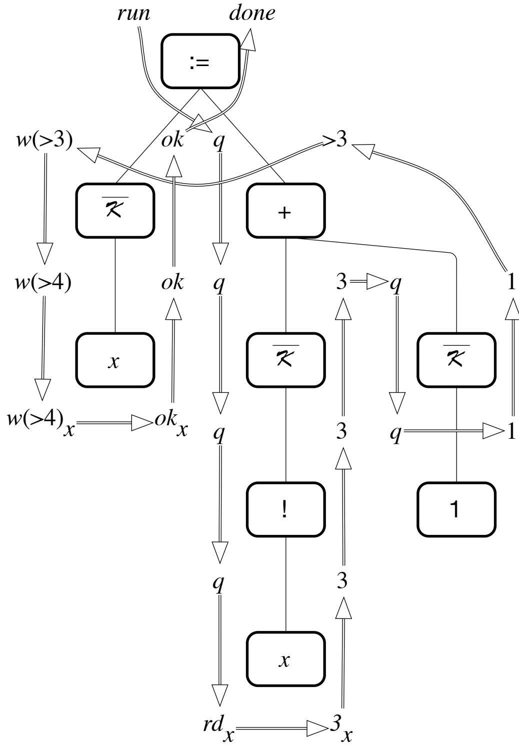


Fig. 3. Flow of data in a term

semantics $[\![\Gamma \vdash M : T]\!]$ can be immediately retrieved from the interaction semantics:

$$[\![\Gamma \vdash M : T]\!] = \mathrm{out}\big(\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle\big). \quad (2)$$

where $\mathrm{out} : \mathcal{A}[\![M]\!] \to \mathcal{A}[\![\Gamma]\!] + \mathcal{A}[\![T]\!]$ is the homomorphism that deletes all symbols from the alphabet of M which are not in $\mathcal{A}[\![\Gamma]\!]$ or $\mathcal{A}[\![T]\!]$.

Unlike standard plays, interaction plays exhibit their "internal" moves, which do not interact in subsequent compositions, but which record all intermediate steps taken during the computation.

Example 1. Consider the interaction strategy of this term:

$$\langle\!\langle x : \mathsf{var}[0,4] \vdash x := !x +_{[0,3]} 1 : \mathsf{com} \rangle\!\rangle$$

One of its interaction plays is:

$$run\, q_2\, q_{21}\, q_{211}\, q_{2111}\, read_{21111}\, 3_{21111}\, 3_{2111}$$
$$3_{211}\, 3_{21}\, q_{22}\, q_{221}\, 1_{221}\, 1_{22}\, ({>}3)_2\, write({>}3)_1$$
$$write({>}4)_{11}\, write({>}4)_{111}\, ok_{111}\, ok_{11}\, ok_1\, done.$$

We use tags on internal moves to indicate the coordinates of the sub-term that corresponds to each move. For instance, q_{21} is the question to the sub-term $!x$, which is the 1st immediate sub-term of $!x + 1$, which in turn is the 2nd immediate sub-term of $x := !x + 1$. The flow of data in this interaction play is represented in Fig. 3 by the double arrow. The approximating copy-cats used to compose terms with different abstractions are represented explicitly. Note that in the left-hand sub-tree writing the abstract value $({>}3) \in [0,3]$ is mapped into $({>}4) \in [0,4]$.

The interaction semantics, rather than the standard semantics, will be used for the purpose of abstraction refinement. The reason is that, given a standard play of an abstracted term, it does not in general contain sufficient information to decide that it can be produced by the concrete version of the term i.e. that if it is an error then it is a genuine counterexample. In state-based abstraction refinement an abstract counterexample to a safety property is guaranteed to be genuine if the computation was deterministic (or, at least, the nondeterminism was not caused by over-abstraction). In standard game semantics, however, all internal steps within a computation are hidden. This results in standard models of abstracted terms in general not containing all information about sources of their nondeterminism.

Example 2. Consider the following abstracted term:

$$\mathsf{abort} : \mathsf{com} \vdash$$
$$\mathsf{new}[]\ \mathsf{in}\ x := 0; \mathsf{if}\ (!x \neq 0)\ \mathsf{then\ abort} : \mathsf{com},$$

where [] is the trivial approximation mapping all integers to a single abstract value. Its standard interpretation is

$$run\, done + run\, run_{\mathsf{abort}}\, done_{\mathsf{abort}}\, done.$$

However, the trace that contains the supposedly undesired action run_{abort} indicating an error is spurious, and the abstraction of x needs to be refined. However, internal moves which point to this abstraction as the source of nondeterminism have been hidden.

3 Conservativity of Abstraction

For safety, we want to show that an abstracted term is a conservative over-approximation of the concrete term, i.e. if the abstracted term is safe then the concrete term is also safe.

For abstractions D and D', we say that D' *refines* D if, for any partition (i.e. abstract value) c' of D', there exists a (unique) partition c of D such that $c' \subseteq c$. We say that c is the *corresponding abstracted value* of c' in D. When D' refines D, and c is a partition of D, we say that D' *splits* c if c is not a partition of D'. We extend the *refine* relation to data types as follows: bool *refines* bool, and $\mathsf{int}_{D'}$ *refines* int_D if D' refines D.

Definition 3. Let types D'_1, D'_2 and D' refine D_1, D_2 and D respectively. We say that an abstracted operation $\mathsf{op}_{D'_1,D'_2,D'}$ is *safely approximated* by abstracted operation $\mathsf{op}_{D_1,D_2,D}$ iff for every c'_1,c'_2,c' of type D'_1, D'_2, D' and c_1, c_2 of type D_1,D_2 respectively, if $c'_1 \subseteq c_1$, $c'_2 \subseteq c_2$ and $c' \in \mathsf{op}_{D'_1,D'_2D'}(c'_1, c'_2)$, then there exists a c of type D such that $c \in \mathsf{op}_{D_1,D_2,D}(c_1, c_2)$ and $c' \subseteq c$.

Example 3. Abstraction $[0,1]$ refines $[0,0]$ such that $[0,1]$ splits >0 and >0 is the corresponding abstracted value of 1 and >1 in $[0,0]$.

We note that abstracted terms may contain non-deterministic branching, because the outcome of integer conversions and arithmetic-logic operations might not be a unique value. As interaction plays contain internal moves, we can distinguish those whose underlying computation did not pass through any non-deterministic branching that is due to abstraction.

Definition 4. (a) Given integer abstractions D and D', and an abstracted value (i.e. partition) c of D, we say that converting c to D' is *deterministic* if there exists an abstracted value c' of D' such that $c \subseteq c'$.
(b) Given an abstracted operation $\mathsf{op}_{D_1,D_2,D}$ and abstracted values c_1 and c_2 of type D_1 and D_2 respectively, we say that the application of op to c_1 and c_2 is *deterministic* if there exists an abstracted value c of type D such that $\forall v_1 \in c_1, v_2 \in c_2,\ v_1 \,\mathsf{op}\, v_2 \ \in\ c$.
(c) An interaction play $u \in \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$ is *deterministic* if each conversion of an abstracted integer value in u is deterministic, and each application of an arithmetic-logic operator in u is deterministic.

We say that a term $\Gamma' \vdash M' : T'$ *refines* a term $\Gamma \vdash M : T$ if they have the same abstract syntax tree, each abstraction in $\Gamma' \vdash M' : T'$ refines the corresponding abstraction in $\Gamma \vdash M : T$ and each abstracted operation in $\Gamma' \vdash M' : T'$ is safely approximated by the corresponding operation in $\Gamma \vdash M : T$. To make the syntax tree match we may assume the presence of approximating copy-cats $\overline{\mathcal{K}}$ even when the abstraction of a function matches that of an argument so they are not strictly required.

For any play t of $\Gamma' \vdash M' : T'$, $\bar{t}$ denotes the image play of $\Gamma \vdash M : T$ obtained by replacing each abstracted integer c' in t by its partition c (such that $c' \subseteq c$) in the corresponding abstraction in $\Gamma \vdash M : T$.

Theorem 2. *Suppose $\Gamma' \vdash M' : T'$ refines $\Gamma \vdash M : T$.*

(i) For any $t \in [\![\Gamma' \vdash M' : T']\!]$, we have $\bar{t} \in [\![\Gamma \vdash M : T]\!]$. The same is true for the $\langle\!\langle - \rangle\!\rangle$ semantics.
(ii) For any deterministic $u \in \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$, there exists $t \in \langle\!\langle \Gamma' \vdash M' : T' \rangle\!\rangle$ such that $u = \bar{t}$.

Proof. By induction on the typing rules of AIA. We only consider the most interesting cases which involve integer abstractions. Proofs for other cases are similar. We prove (i) for the $\langle\!\langle - \rangle\!\rangle$ semantics since the proof for the $[\![-]\!]$ semantics follows from the former.

Consider the case of any constant v. We have that $q \cdot d' \in \langle\!\langle \Gamma' \vdash v : \mathsf{exp}D' \rangle\!\rangle$ and $q \cdot d \in \langle\!\langle \Gamma \vdash v : \mathsf{exp}D \rangle\!\rangle$, such that $v \in d'$, $v \in d$ and $d' \subseteq d$. Moreover, $q \cdot d$ is deterministic and $q \cdot d = \overline{q \cdot d'}$.

Consider the case of any arithmetic-logic operator op. Let $t \in \langle\!\langle \Gamma' \vdash M' \mathsf{op}_{D',E'F'} N' : \mathsf{exp}F' \rangle\!\rangle$ be of the form $q \cdot q_1 \cdot s_1 \cdot d'_1 \cdot q_2 \cdot s_2 \cdot e'_2 \cdot f'$ and let $f' \in \mathsf{op}_{D',E',F'}(d', e')$. From the induction hypothesis, there exist partitions d of type D and e of type E such that $q \cdot \overline{s_1} \cdot d \in \langle\!\langle \Gamma \vdash M : \mathsf{exp}D \rangle\!\rangle$, $q \cdot \overline{s_2} \cdot e \in \langle\!\langle \Gamma \vdash N : \mathsf{exp}E \rangle\!\rangle$, and $d' \subseteq d$, $e' \subseteq e$. From the safety of $\mathsf{op}_{D,E,F}$, there exists f of type F such that $f \in \mathsf{op}_{D,E,F}(d,e)$ and $f' \subseteq f$. Then, $\bar{t} = q \cdot q_1 \cdot \overline{s_1} \cdot d_1 \cdot q_2 \cdot \overline{s_2} \cdot e_2 \cdot f \in \langle\!\langle \Gamma \vdash M \,\mathsf{op}_{D,E,F}\, N : \mathsf{exp}F \rangle\!\rangle$.

Let $u = q \cdot q_1 \cdot s_1 \cdot d_1 \cdot q_2 \cdot s_2 \cdot e_2 \cdot f$ be a deterministic play in $\langle\!\langle \Gamma \vdash M \,\mathsf{op}_{D,E,F} N : \mathsf{exp}F \rangle\!\rangle$, where $f = \mathsf{op}_{D,E,F}(d,e)$ (i.e. $\forall v_1 \in d, v_2 \in e,\ v_1 \,\mathsf{op}\, v_2 \in f$). From the induction hypothesis and the safety of $\mathsf{op}_{D,E,F}$, there must be partitions $d' \subseteq d$, $e' \subseteq e$ and $f' \subseteq f$ of type D',E' and F' respectively, and plays s'_1, s'_2, such that $q \cdot s'_1 \cdot d' \in \langle\!\langle \Gamma' \vdash M' \rangle\!\rangle$, $q \cdot s'_2 \cdot e' \in \langle\!\langle \Gamma' \vdash N' \rangle\!\rangle$, $f' \in \mathsf{op}_{D',E',F'}(d', e')$, and $s_1 = \overline{s'_1}$, $s_2 = \overline{s'_2}$. Then, $t = q \cdot q_1 \cdot s'_1 \cdot d'_1 \cdot q_2 \cdot s'_2 \cdot e'_2 \cdot f' \in \langle\!\langle \Gamma' \vdash M' \,\mathsf{op}_{D'}\, N' : \mathsf{exp}D' \rangle\!\rangle$ and $u = \bar{t}$.

Consider the case of assignment. Let $t \in \langle\!\langle \Gamma' \vdash M' := N' : \mathsf{com} \rangle\!\rangle$ be of the form $run \cdot q_2 \cdot s_2 \cdot e'_2 \cdot write(d')_1 \cdot s_1 \cdot ok_1 \cdot done$, where d', e' are of type D', E' respectively and $e' \cap d' \neq \emptyset$. From the induction hypothesis, there exist partitions e of type E and d of type D such that $q \cdot \overline{s_2} \cdot e \in \langle\!\langle \Gamma \vdash N : \mathsf{exp}E \rangle\!\rangle$, $write(d) \cdot \overline{s_1} \cdot ok \in \langle\!\langle \Gamma \vdash M : \mathsf{var}D \rangle\!\rangle$, and $d' \subseteq d$, $e' \subseteq e$. Then, $\bar{t} = run \cdot q_2 \cdot \overline{s_2} \cdot e_2 \cdot write(d)_1 \cdot \overline{s_1} \cdot ok_1 \cdot done \in \langle\!\langle \Gamma \vdash M := N : \mathsf{com} \rangle\!\rangle$ and $d \cap e \neq \emptyset$.

Let $u = run \cdot q_2 \cdot s_2 \cdot e_2 \cdot write(d)_1 \cdot s_1 \cdot ok_1 \cdot done$ be a deterministic play in $\langle\!\langle \Gamma \vdash M := N : \mathsf{com} \rangle\!\rangle$, where $e \subseteq d$. From the induction hypothesis, there must be partitions $d' \subseteq d$ and $e' \subseteq e$ of type D' and E' respectively, and plays s'_1, s'_2, such that $q \cdot s'_2 \cdot e' \in \langle\!\langle \Gamma' \vdash N' \rangle\!\rangle$, $write(d') \cdot s'_1 \cdot ok \in \langle\!\langle \Gamma' \vdash M' \rangle\!\rangle$, $d' \cap e' \neq \emptyset$, and $s_1 = \overline{s'_1}$, $s_2 = \overline{s'_2}$. Then, $t = run \cdot q_2 \cdot s'_2 \cdot e'_2 \cdot write(d')_1 \cdot s'_1 \cdot ok_1 \cdot done \in \langle\!\langle \Gamma' \vdash M' := N' : \mathsf{com} \rangle\!\rangle$ and $u = \bar{t}$. $\square$

Note that this can be strengthened to apply to interaction plays which are deterministic with respect to the abstractions in $\Gamma' \vdash M' : T'$. The latter notion allows non-deterministic conversions of, and operator applications to, abstracted values which are not split by the corresponding abstractions in $\Gamma' \vdash M' : T'$.

The following consequence of Theorem 2 and the correspondence between standard and interaction game semantics (Eqn. 2) will justify the correctness of the abstraction refinement procedure.

Corollary 1. *Suppose $\Gamma' \vdash M' : T'$ refines $\Gamma \vdash M : T$.*

(i) If $[\![\Gamma \vdash M : T]\!]$ is safe, then $\Gamma' \vdash M' : T'$ is safe.
(ii) If $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$ has a deterministic unsafe interaction play, then $\Gamma' \vdash M' : T'$ is unsafe.

4 Abstraction Refinement

An abstraction D is *finitary* if it has finitely many partitions, and a term is *finitely abstracted* if it contains only finitary abstractions. A set of abstractions is *effective* if their equivalence classes have finite representations, and if conversions of abstract values between abstractions,

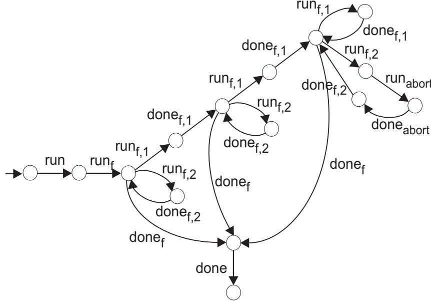


Fig. 4. A strategy as a finite automaton

and all arithmetic-logic operators over abstract values, are computable.

The following is immediate, given the regular-language representation of strategies:

Proposition 1. *For any finitely abstracted term $\Gamma \vdash M : T$ with abstractions from an effective set, the set $[\![\Gamma \vdash M : T]\!]$ is a regular language. Moreover, an automaton which recognises it is effectively constructible. The same is true for the $\langle\!\langle - \rangle\!\rangle$ semantics.*

Let $A[\![\Gamma \vdash M : T]\!]$ and $A\langle\!\langle \Gamma \vdash M : T\rangle\!\rangle$ denote the automata which recognise $[\![\Gamma \vdash M : T]\!]$ and $\langle\!\langle \Gamma \vdash M : T\rangle\!\rangle$ respectively. Since there is no hiding in the construction of $A\langle\!\langle \Gamma \vdash M : T\rangle\!\rangle$, this automaton is deterministic.

Given a finite word u and a deterministic automaton A which accepts u, we call u *cycle-free* if the accepting run visits any state of A at most once. From here on we work only with the abstractions $[]$ and $[n, m]$, where $n \leq 0 \leq m$. Observe that these abstractions are finitary and form an effective set.

Example 4. Consider the term

$$\mathsf{abort} : \mathsf{com},\ f : \mathsf{com}, \mathsf{com} \to \mathsf{com} \vdash$$
$$\mathsf{new}[0,1]\, x := 0 \,\mathsf{in}\, f(x := !x + 1,\ \mathsf{if}\ (!x > 1)\ \mathsf{then\ abort})$$

The strategy for this term represented as a finite automaton is shown in Fig. 4. The model illustrates only the possible behaviors of this term: if the non-local procedure f calls its first argument, two or more times, and afterwards its second argument then the term is abort-unsafe; otherwise the term terminates successfully. The model does not assume that f uses its arguments, or how many times or in what order. Notice that no references to the variable x appear in the model because it is locally defined and so not visible from the outside of the term. An unsafe play is:

$$run\, run_f\ run_{f,1}\ done_{f,1}\ run_{f,1}\ done_{f,1}$$
$$run_{f,2}\ run_{\mathsf{abort}}\ done_{\mathsf{abort}}\ done_{f,2}\ done_f\ done.$$

So, the term is unsafe.

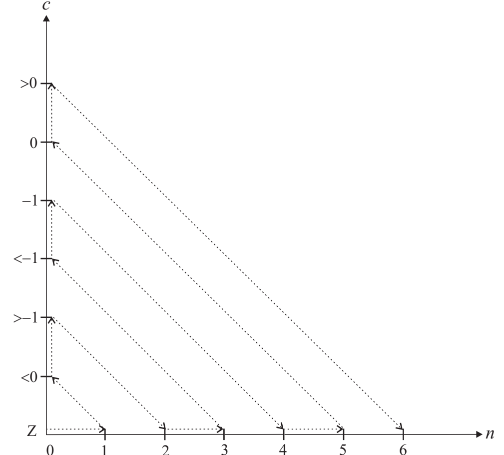


Fig. 5. A possible definition of $\sqsubseteq$

Let $\prec$ denote the following computable linear ordering between abstracted values:

$$\mathbb{Z} \prec (<0) \prec (>-1) \prec (<-1) \prec -1 \prec 0 \prec (>0) \prec \cdots$$
$$(<(-n-1)) \prec -n-1 \prec n \prec (>n) \prec \cdots$$

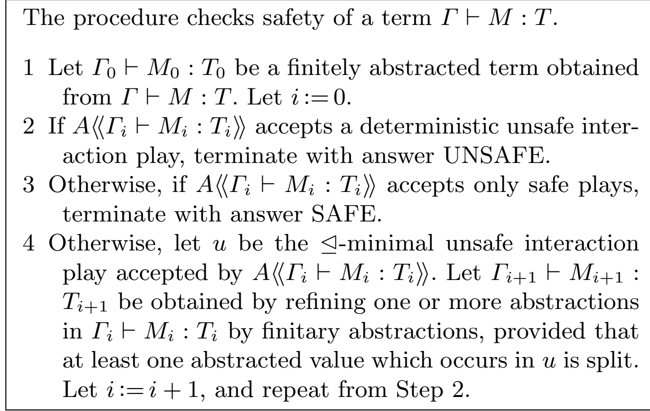
This ordering has the property that $c \prec c'$ whenever $c' \subset c$. For two moves (possibly tagged with sub-term coordinates) r and r' which are equal except for containing different abstracted integer values c and c', let $r \prec r'$ if $c \prec c'$. Now, we extend this ordering to a computable linear ordering on all moves (in an arbitrary but fixed way), and denote it by $\prec$. Let $\prec$ also denote the linear orderings on plays obtained by lifting the linear ordering on moves lexicographically.

Let $(n, c) \sqsubseteq (n', c')$ be any computable linear ordering between pairs of non-negative integers and abstracted integer values which is obtained by extending the partial ordering defined by $n \leq n'$ and $c \preceq c'$, and which admits no infinite strictly decreasing sequences, and no infinite strictly increasing sequences bounded above. An example of such an ordering $\sqsubseteq$ is given in Fig. 5. The arrows show how the ordering increases. So, we have that $(3, \mathbb{Z}) \sqsubseteq (2, <0)$, but $(1, <0) \sqsubseteq (3, \mathbb{Z})$. For any play u, let $|u|$ denote its length, and $\max(u)$ denote the $\prec$-maximal abstracted integer value in u (or $\mathbb{Z}$ if there is no such value). Let $u \sqsubseteq u'$ mean $(|u|, \max(u)) \sqsubseteq (|u'|, \max(u'))$. Now, let $\trianglelefteq$ be the linear ordering between plays such that $u \trianglelefteq u'$ if and only if either $u \sqsubset u'$, or $|u| = |u'|$, $\max(u) = \max(u')$ and $u \preceq u'$.

Lemma 1. *In the linear order of all plays with respect to $\trianglelefteq$:*

(i) there is no infinite strictly decreasing sequence;
(ii) there is no infinite strictly increasing sequence which is bounded above.

Proof. This is due to the following two facts. Firstly, the $\sqsubseteq$ ordering between pairs of non-negative integers


The procedure checks safety of a term $\Gamma \vdash M : T$.

1 Let $\Gamma_0 \vdash M_0 : T_0$ be a finitely abstracted term obtained from $\Gamma \vdash M : T$. Let $i := 0$.
2 If $A\langle\!\langle \Gamma_i \vdash M_i : T_i \rangle\!\rangle$ accepts a deterministic unsafe interaction play, terminate with answer UNSAFE.
3 Otherwise, if $A\langle\!\langle \Gamma_i \vdash M_i : T_i \rangle\!\rangle$ accepts only safe plays, terminate with answer SAFE.
4 Otherwise, let u be the $\trianglelefteq$-minimal unsafe interaction play accepted by $A\langle\!\langle \Gamma_i \vdash M_i : T_i \rangle\!\rangle$. Let $\Gamma_{i+1} \vdash M_{i+1} : T_{i+1}$ be obtained by refining one or more abstractions in $\Gamma_i \vdash M_i : T_i$ by finitary abstractions, provided that at least one abstracted value which occurs in u is split. Let $i := i + 1$, and repeat from Step 2.

Fig. 6. Abstraction refinement procedure

and abstracted integer values has the properties (i) and (ii). Secondly, for any such pair (n, c), there are only finitely many plays u such that $|u| = n$ and $\max(u) = c$. □

The abstraction refinement procedure (ARP) is given in Fig. 6. Note that, in Step 1, the initial abstractions can be chosen arbitrarily; and in Step 4, arbitrary abstractions can be refined in arbitrary ways, as long as that splits at least one abstracted value in u. These do not affect correctness and semi-termination, but they allow experimentation with different heuristics in concrete implementations.

Theorem 3. *ARP is well-defined and effective. If it terminates with SAFE (UNSAFE, respectively), then $\Gamma \vdash M : T$ is safe (unsafe, respectively).*

Proof. For well-defined-ness, Lemma 1 (i) ensures that the $\trianglelefteq$-minimal unsafe interaction play u accepted by $A\langle\!\langle \Gamma_i \vdash M_i : T_i \rangle\!\rangle$ always exists. Since the condition in Step 2 was not satisfied, u is not deterministic. Therefore, u cannot contain only singleton abstracted values, so there is at least one abstracted value in u which can be split.

Effectiveness follows from the fact that it suffices to consider cycle-free plays in Step 4, and from computability of $\trianglelefteq$.

If ARP terminates with SAFE (UNSAFE, respectively), then $\Gamma \vdash M : T$ is safe (unsafe, respectively) by Corollary 1. □

Theorem 4. *If $\Gamma \vdash M : T$ is unsafe then ARP will terminate with UNSAFE.*

Proof. By the correspondence in Eqn. 2, there exists an unsafe interaction play $t \in \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$.

For each i, let U_i be the set of all unsafe $u \in \langle\!\langle \Gamma_i \vdash M_i : T_i \rangle\!\rangle$, and let $u_i^\dagger$ be the $\trianglelefteq$-minimal element of U_i.

It follows by Theorem 2 that, for any $u \in \langle\!\langle \Gamma_{i+1} \vdash M_{i+1} : T_{i+1} \rangle\!\rangle$, $\overline{u} \in \langle\!\langle \Gamma_i \vdash M_i : T_i \rangle\!\rangle$. Also, we have $\overline{u} \trianglelefteq u$, since they have the same length and $\overline{c} \preceq c$ for any c. Now, Step 4 ensures that, for any i, $u_i^\dagger \notin \langle\!\langle \Gamma_{i+1} \vdash M_{i+1} : T_{i+1} \rangle\!\rangle$.

Therefore, $u_0^\dagger \lhd u_1^\dagger \lhd \cdots u_i^\dagger \lhd \cdots$. But, for each i, $u_i^\dagger \trianglelefteq \overline{t}^i \trianglelefteq t$. By Lemma 1 ($ii$), ARP must terminate for $\Gamma \vdash M : T$. □

ARP may diverge for safe terms. This is generally the case with abstraction refinement methods since the underlying problem is undecidable. A simple example is the term

$$e : \mathsf{expint}, \mathsf{abort} : \mathsf{com} \vdash \\ \mathsf{newint}\, x := e \,\mathsf{in}\, \mathsf{if}\, (!x = !x + 1)\, \mathsf{then}\, \mathsf{abort} : \mathsf{com}$$

This term is safe, but any finitely abstracted term (anti-refinement) of it is unsafe.

5 Implementation

We have developed a series of game-based verification tools. Originally we developed a direct implementation of the CEGAR procedure outlined in Fig. 7, leading to a tool called GameChecker. However, this suffered some severe inefficiencies.

In this section we explain in several stages how our approach can be reformulated to achieve much more competitive efficiency while retaining the novelty of compositionality. We present the key techniques we use to gain time efficiency (Section 5.1), space efficiency (Section 5.2), and iteration-count efficiency (Section 5.3). These techniques are combined in our newer tool called Mage. A larger case study and tool comparison is presented in Section 5.4.

5.1 Symbolic Game Automata

Game models are built by constructing the transition relation of an automaton, with transitions labelled by game moves, as suggested by Fig. 4. For constants the automata can be defined directly. For composite terms (applications) the automaton is derived from the sub-automata. For binders like λ, the game model is a retagging so the automaton is a relabelling of the body automaton.

Naive implementation of this plan can easily result in asymptotic inefficiency. First let us explain the problem by giving two simple but bad implementation techniques for composite automata.

Let $\triangle \in \mathit{State} ::= n \mid (\triangle_1, \triangle_2)$. Let Move be the set M_A of moves in the alphabet of the type of the term under analysis.

5.1.1 Composition by transition matching

The game automaton of term MN can be built by:

- representing automata as functions of type $State \to 2^{(State \times Move)}$ — i.e., from a state, what are all the possible outgoing transitions (moves and next-states);
- building the automata of M and N;
- forming transitions from the composite state $(\triangle_1, \triangle_2)$ by taking transitions from $\triangle_1$ to $(\triangle_3, m)$ in the M automaton and from $\triangle_2$ to $(\triangle_4, m)$ in the N automaton; then add a composite transition to $((\triangle_3, \triangle_2), m)$ or $((\triangle_1, \triangle_4), m)$ as apt if m is a non-interaction move, or to $((\triangle_3, \triangle_4), m)$ if m is interaction (recall that our type system ensures that the move alphabets are equal at composition points).

The problem with this is that the entire behaviour of MN is constructed out of a representation of the entire behaviour of M and the entire behaviour of N. This gets terribly inefficient as refinements grow the data abstractions.

Typically, only a few of the behaviours of M are enabled by N — consider $(\lambda x.x+1)k$ for a trivial example where the composite model remains constant size however large the abstraction for x and the model of the function. Similarly, M often uses only a few of the behaviours of N as in $(\lambda x.x := k)y$ or $\mathsf{newint}\, x := k \,\mathsf{in}\, (\lambda y.y * y)(x := !x + 1; x)$.

Moreover, the transitions of N should not even be considered until M makes an interaction move, and after that transitions of M should not be considered until N makes an interaction move.

5.1.2 Composition by transition testing

The reachable states of game automaton of term MN can be found by:

- representing automata as functions of type $State \times Move \to 2^{State}$ — i.e., from a state, what are the next-states reached via transitions with a specific move-label;
- for a reachable composite state $(\triangle_1, \triangle_2)$, for every move m, if the automaton of M has a transition from $(\triangle_1, m)$ to $\triangle_3$ then if m is non-interaction then $(\triangle_3, \triangle_2)$ is a composite next-state (similarly for N); and if m is interaction then $(\triangle_3, \triangle_4)$ is a composite next-state if the automaton of N has a transition from $(\triangle_2, m)$ to $\triangle_4$.

As expressed simply above this approach also has the problem that both M and N are queried about every move. But the more serious problem with this approach is that as the move alphabets grow and the terms become more specialized the overwhelming majority of state-move queries will return $\emptyset$.

However, this approach makes two important changes which we can use to gain efficiency. Firstly, for interaction moves, the composite automata asks N if it can match the specific moves that M can make rather than searching all interaction moves of N for a match. Secondly, the automata description does not involve *constructing* any transitions; the reachability of error states can be decided by collecting reachable states and only the moves along the path to the error need recording. This is the idea of symbolic automata: to encode the transitions as functions rather than as a graph.

5.1.3 Efficient symbolic composition

By combining the better aspects of the above approaches we derive a definition of symbolic game automata that avoids all the inefficiency problems. It works by adding a controlling direction to every composite state: $\triangle_1 \blacktriangleleft \triangle_2$ (function-term in control) or $\triangle_1 \blacktriangleright \triangle_2$ (argument-term in control). Possible next-states are found by asking the controlling sub-automaton for all its next transitions and asking the other sub-automaton if it can match any that are labelled by interaction moves. Therefore we need to encode every game automata as *two* functions: one of type $State \to 2^{(State \times Move)}$ for when it is in control and one of type $State \times Move \to 2^{State}$ for when it is not.

Example 5. Reconsider $(\lambda x.x+1)k$. To begin, $(\lambda x.x+1)$ is in control so first we follow its 'run' move. Then its body makes the interaction move 'run x' so we ask the k automaton if it can make a move from its initial state labelled by 'run' (because the λ un-tags 'x'). Then k is in control and we ask what transition it can make, there is only one (k) and it is an interaction move so we ask $(\lambda x.x + 1)$ if it has a transition labelled k, which it has, and so regains control. Then finally there is only one transition possible which returns the move 'k+1'.

Example 6. $\mathsf{newint}\, x := k \,\mathsf{in}\, (\lambda y.y * y)(x := !x + 1; x)$. The interesting application in this term is of $\mathsf{newint}\, x := k$ to the rest (recall from Section 2 that a new-expression is an application when expressed in the underlying functional notation). The automaton of $\mathsf{newint}\, x := k$ starts in a state that records that the value of x is k then runs the body. Control stays in the body except where x is read or written. Therefore the interaction moves in the composite automata are these: (1) a read of x returning the move k; (2) a write of $k + 1$ to x; (3) a read of x for the LHS of $*$; (4) a read of x; (5) a write of $k + 2$ to x; (6) a read of x for the RHS of $*$. Then the body return move is $(k+1)*(k+2)$; this is also an interaction move which the $\mathsf{newint}\, x := k$ automaton copies out as its return move and the final move in the model.

5.2 Laziness

By laziness in model checking we mean tracing parts of the model as they are demanded by the state space exploration algorithm. This reduces the space overhead for large models because only visited states, and the moves needed for counterexample certification or for abstraction refinement, are stored. When the program has errors which the abstraction is detailed enough to reveal, laziness also has the benefit that counterexamples can

be certified as they are met so there can also be a big time saving.

The efficient symbolic automata developed in the previous section are almost ready to support laziness. The important consideration for implementation is that the next-transitions automaton encoding must produce its result lazily — i.e. transitions can be produced one at a time, without performing any computation pertinent only to transitions other than the one produced. This is important because game models often have very high branching degrees, so using an eager next-transition function lazily (i.e. only considering one state at a time but considering all its transitions together) would often perform much worse than true laziness.

To make this explicit in the definition, we number the outgoing transitions of each state from 0 up to the number of transitions minus one. The states of lazy automata for constants are the symbolic automata states paired with transition numbers and the states of composite lazy automata are the pairs of the states of the sub-(lazy)-automata, paired with transition numbers. Thus the lazy automaton next-transition function for constants is derived from the symbolic version according to the following equation. The transition order in the lazy version is fixed but arbitrary.

$$\begin{aligned} &\mathit{next}(\triangle, i) = \begin{cases} (\triangle_i, \mathit{move}_i), \text{if } i < n \\ \bot, \text{otherwise} \end{cases} \\ &\mathit{iff}\ \ \mathit{next\text{-}transitions}(\triangle) = \{(\triangle_i, \mathit{move}_i)\}_{i=0}^{n-1} \end{aligned}$$

For composite terms it is built from the lazy *next* functions of the sub-terms in the obvious way.

Now we present a search procedure for lazy symbolic models. Fig. 7 defines a search procedure for lazy symbolic models. The idea is that the user defines a program abstraction, builds the corresponding lazy *next* function and starts the search with $\mathit{lazy\text{-}search}(\mathit{next}, \{(\epsilon, (\triangle_0, 0))\}, \emptyset)$. That is, initially no states have been visited and the frontier contains the empty trace paired with the initial state. The initial state specifies that the first transition is to be searched first. After this, the Fig. 7 presentation is non-deterministic in that both the next state after the first transition and the state at the start of the second transition are added to the frontier. A DFS can be forced by prioritizing the former and a BFS by prioritizing the latter. When the search find an *abort* move it returns its entire state (the counterexample trace, frontier and visited sets). The user program can then analyze the counterexample and generate refinements if appropriate. The user can then choose whether to continue searching in the current model, or begin searching one of the refined models.

$$\begin{aligned}
&\mathit{lazy\text{-}search}(\mathit{next} : \mathit{State} \to (\mathit{State} \times \mathit{Move})_\bot, \\
&\qquad\qquad \mathit{frontier} : 2^{(\mathit{Move}^* \times \mathit{State})}, \mathit{visited} : 2^{\mathit{State}}) = \\
&f := \mathit{frontier} \\
&v := \mathit{visited} \\
&\text{while } \exists f', \mathit{trace}, \triangle, n\ .\ (f' = f - \{(\mathit{trace}, (\triangle, n))\}) \\
&\ \ \text{if } n = 0 \text{ then } v := v \cup \{(\triangle, n)\} \\
&\ \ \text{if } \mathit{next}(\triangle, n) = ((\triangle', n'), \mathit{move}) \\
&\ \ \text{then if } \mathit{move} = \mathit{abort} \text{ then return } (\mathit{trace}, f, v) \\
&\ \ f := f' \cup \{(\mathit{trace}, (\triangle, n+1))\} \\
&\qquad \cup \{(\mathit{trace} \cdot \mathit{move}, (\triangle', 0)) \mid \triangle' \notin v\} \\
&\text{return } \bot
\end{aligned}$$

Fig. 7. Lazy symbolic model search

5.3 Reducing Refinement Iterations

A trace leading to an error move can be accepted as genuine proof of unsafety when all the values in its moves are exact (not ranges) because then it is also a trace in the precise model. However, using this as the criterion can often result in many iterations.

MAGE uses a test that accepts traces with approximate symbols when the approximation does not cause non-determinism in the approximated automaton for transitions that are deterministic in the precise version. This test is a check of whether the trace is *deterministic* in the sense of Definition 4. The aim of this check is to spare the need for an expensive SAT test, commonly used in other verification systems. Note that the determinism check is only valid for verification based on data approximation — in particular, it does not apply to the popular predicate abstraction based systems in general.

Example 7. The model of the term

$$\begin{aligned} &e : \mathsf{exp\,int} \vdash \mathsf{newint}\, x := e\, \mathsf{in} \\ &\quad \mathsf{if}\ (x < 1000000\, \mathsf{or}\, x > 1000)\, \mathsf{then\ skip\ else\ abort} \end{aligned}$$

has no paths to abort as soon as the abstraction of x is precise enough to contain no values that are both less than 1000 and greater than 1000000. When there are such values the model does contain traces to abort but they are all non-deterministic because there must also be traces that follow the skip branch.

Another way to reduce the iteration count is by a careful choice of refinement heuristics. In general, our research suggests that it is better to only increase the size of type abstractions by one at each refinement iteration. Introducing more divisions in the hope of homing in on errors more quickly tends to make search slower; pruning would help but is difficult to achieve in the general case without causing refinement iteration cycles.

Therefore our basic refinement heuristic works by refining an abstract value (i, j) to (i, k) and $(k + 1, j)$. It is better to have k close to i when low values are more likely to reveal errors and close to j for high value errors. We do no analysis to choose a best value for k in such splits. But we do use the heuristic that when i and j are both finite then k should be the mid-point so error values

can be found in a logarithmic number of iterations and a minimal expansion of the abstract type size. For unbounded abstract values, introducing a small bounded range often reveals errors quickly where they are data independent (i.e. when *any* concrete value will suffice). When particular concrete values are needed to reveal errors the heuristic is that smaller values are more likely. Therefore we use $k = 0$ if $i \leq 0$ and $j > 0$. And we use $k = i + 2^n - 1$ if $i > 0$ and $j = \infty$, using a suitable modest (but not too small!) natural n. And similarly, $k = j - (2^n - 1)$ if $i = -\infty$ and $j \leq 0$.

Example 8. With our heuristic, the term

$$e : \mathsf{exp\,int} \vdash \mathsf{newint}\, x := e\, \mathsf{in}$$
$$\mathsf{if}\, (x < 1000000\, \mathsf{or}\, x > 1000)\, \mathsf{then\ skip\ else\ abort}$$

is provably safe after only six refinement iterations.

5.4 Results

Now we compare our game-based verification tools and a non-compositional tool on a larger, more realistic, problem.

Mage [BG08] is our most recent tool and incorporates all the concepts explained in previous sections. It is a stand-alone tool implemented in Haskell.

GameChecker [DGL06] is our previous tool. It was the first game-based verification system with data abstraction and CEGAR. But it does not feature the concepts explained in the preceding parts of this section. It is implemented in Java. It works by building the game model for a given abstraction, compiling it into a CSP process, checking safety with FDR[Ros98][2] and then analyzing the FDR output to produce an answer (safe or unsafe) or an abstraction-refinement.

Blast [HJM05] is a very powerful software verification tool for C programs, developed out of work on predicate abstraction. We use it to show how the game-based tools contrast with what we consider the leading contemporary non-compositional approach. Blast also performs a CEGAR procedure and it uses an external theorem prover.

The reader should bear in mind that this example compares our data abstraction tools with a predicate abstraction tool (because this is where the current emphasis in software model checking is placed). The differing capabilities of data abstraction and predicate abstraction should, and do, mean that depending on the example one will usually outperform the other. Therefore our practical result is a demonstration that our compositional data-abstraction tools are not at a consistent disadvantage in comparison to other approaches.

[2] FDR `http://www.fsel.com` is a commercial model checker for CSP, free for academic use.

Example 9. For an example with a very large model we consider the following simple stack ADT.

```
oflo : com,        // called on pop empty
uflo : com,        // called on push full
e : expint,        // data
check(expint, com) : com       // arbitrary user
⊢
newint buffer[k] in        // fixed-size stack of numbers
newint top := 0 in        // first free index
let pop =
  if (top = 0) then uflo
    else {top := top − 1; buffer[top + 1]} in
let push(varint x) =
  if (top = k) then oflo
    else {buffer[top] := x; top := top + 1} in
check(pop, push(e))
```

The stack is implemented as a fixed-size array of k elements (which we can vary). The push in this ADT calls *oflo* if the stack is full and the pop calls *uflo* if the stack is empty.

Model checking the stack exercises all possible combinations of push and pop, revealing the contexts that lead to oflo and uflo. Table 2 and Table 1 show the time taken by the three tools to find such situations.

Mage. Calls to uflo are found almost immediately for any k. This demonstrates the benefit of laziness: as one pop on the starting empty stack causes uflo, the fault is revealed by a short deterministic error trace in the refined model, and this trace is found early in the lazy-search.

Calls to *oflo* are found in $O(log(k))$ iterations. This is because it is necessary to refine the top variable enough to be deterministic for all buffer array indices. But it is not really necessary to refine the input supplying the pushed values to reveal the fault. This demonstrates the benefit of the refinement heuristics. Because the counterexample trace is a series of $k + 1$ pushes the laziness does not help, but because the sufficiently-refined model is quite large the symbolic technique is key to achieving these verification times.

GameChecker. The search for *uflo* takes few iterations but increasing and long times. As for Mage, few iterations because the shortest counterexample 'pop empty' is independent of both buffer size and stacked data. But the long times are simply a result of no laziness: the whole stack model gets very large as k grows and it is all built before the search begins.

The call to *oflo* is found in a linear number of iterations. This happens because the refinement heuristic replaces an abstract value (i, j) by (i, i) and $(i + 1, j)$

stack	Mage		GameChecker		Blast	
size	time	(iters)	time	(iters)	time	(iters)
2	0.03	(2)	5.31	(2)	0.07	(1)
4	0.03	(2)	8.21	(2)	0.07	(1)
8	0.03	(2)	20.29	(2)	0.07	(1)
16	0.03	(2)	78.26	(2)	0.07	(1)
32	0.03	(2)	494.20	(2)	0.07	(1)
64	0.03	(2)	8,982.13	(2)	0.07	(1)
128	0.03	(2)	>7 hrs	-	0.07	(1)

Table 1. Stack underflow detection (times in seconds).

stack	Mage		GameChecker		Blast	
size	time	(iters)	time	(iters)	time	(iters)
2	0.1	(2)	10.1	(4)	1.6	(2)
4	0.1	(3)	27.6	(6)	3.3	(4)
8	0.2	(4)	112.6	(10)	4.6	(8)
16	0.4	(5)	780.7	(18)	7.8	(16)
32	1.2	(6)	12,268.1	(36)	17.3	(32)
64	3.9	(7)	>7 hrs	-	43.7	(64)
128	13.9	(8)	-	-	145.3	(128)
224	19.1	(9)	-	-	506.4	(224)
225	19.3	(9)	-	-	-	-
256	54.8	(9)	-	-	-	-
512	215.3	(10)	-	-	-	-
1024	864.7	(11)	-	-	-	-

Table 2. Stack overflow detection (times in seconds).

so to fully eliminate the nondeterminism for all array indexes takes a linear number of iterations. The slower times here are due more to the lack of symbolic modelling.

Blast. The call to *uflo* is found quickly for any k, similar to Mage. This shows a situation where lazy data abstraction and lazy predicate abstraction have similar performance (building a few paths in a model vs testing the reachability of a few paths).

The call to *oflo* is found in a linear number of iterations. The comparison with GameChecker suggests that the predicate-abstracted models must remain fairly small as the precision is increased whereas the data-abstracted models grow a lot. Comparison with Mage suggests again that the predicate-abstracted models are smaller and quicker to check than symbolic data-abstracted models because the Blast iteration times become lower than the Mage iterations. However, the logarithmic iteration count produced by the Mage refinement heuristics more than compensates for this effect, resulting in much greater verification power on this problem. The overall conclusion is that the Blast approach may benefit from using different refinement heuristics on this type of problem and the Mage approach could benefit if predicate-abstracted models can be used in the game-based style.

6 Conclusions and related work

In this paper, we extended the applicability of game-based software model checking by a data-abstraction refinement procedure which applies to open program fragments which can contain infinite integer types, and which is guaranteed to discover an error if it exists. The procedure is made possible and it was justified by a firm theoretical framework. A prototype tool implementing the procedure was described. It has been tested on a variety of academic examples.

Possibilities for future work include extensions to programs with concurrency [GM06], recursion and higher-order procedures [HO09], as well as to abstractions by arbitrary predicates. The goal is a tool which uses game semantics to achieve compositional verification of practical programs.

The pioneering applications of game models to program analysis were by Hankin and Malacaria [MH98], who also use nondeterminism as a form of abstraction. Their abstraction techniques apply to higher-order constructs rather than just data, by forgetting certain information used in constructing the game models (the *justification pointers*). It is an interesting question whether this style of abstraction can be iteratively refined. The first applications of game-semantic models to model checking were by Ghica and McCusker [GM00]. The latter line of research was further pursued as part of the *Algorithmic Game Semantics* research programme at the University of Oxford [AGMO04], and by Dimovski and Lazić [DL06].

References

[AGMO04] Samson Abramsky, Dan R. Ghica, Andrzej S. Murawski, and C.-H. Luke Ong. Applying game semantics to compositional software modeling and verification. In *TACAS*, pages 421–435, 2004.

[AM96] Samson Abramsky and Guy McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996.

[BG08] Adam Bakewell and Dan R. Ghica. On-the-fly techniques for game-based software model checking. In *TACAS*, pages 78–92, 2008.

[BR01] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *CAV*, pages 260–264, 2001.

[CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *CC*, pages 159–178, 2002.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[DGL05] Aleksandar Dimovski, Dan R. Ghica, and Ranko Lazić. Data-abstraction refinement: A game semantic approach. In *SAS*, pages 102–117, 2005.

[DGL06] Aleksandar Dimovski, Dan R. Ghica, and Ranko Lazić. A counterexample-guided refinement tool for open procedural programs. In *SPIN*, pages 288–292, 2006.

[DL06] Aleksandar Dimovski and Ranko Lazić. Assume-guarantee software verification based on game semantics. In *ICFEM*, pages 529–548, 2006.

[GB09] Dan R. Ghica and Adam Bakewell. Clipping: A semantics-directed syntactic approximation. In *LICS*, 2009. (forthcoming).

[Ghi09] Dan R. Ghica. Applications of game semantics: From software analysis to hardware synthesis. In *LICS*, pages 17–26, 2009.

[GM00] Dan R. Ghica and Guy McCusker. Reasoning about Idealized Algol using regular languages. In *ICALP*, pages 103–115, 2000.

[GM03] Dan R. Ghica and Guy McCusker. The regular-language semantics of second-order Idealized Algol. *Theor. Comput. Sci.*, 309(1-3):469–502, 2003.

[GM06] Dan R. Ghica and Andrzej S. Murawski. Compositional model extraction for higher-order concurrent programs. In *TACAS*, pages 303–317, 2006.

[GM08] Dan R. Ghica and Andrzej Murawski. Angelic semantics of fine-grained concurrency. *Annals of Pure and Applied Logic*, 151(2-3):89–114, 2008.

[GMO06] Dan R. Ghica, Andrzej S. Murawski, and C.-H. Luke Ong. Syntactic control of concurrency. *Theor. Comput. Sci.*, 350(2-3):234–251, 2006.

[HJM05] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The BLAST software verification system. In *SPIN*, pages 25–26, 2005.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *SPIN*, pages 235–239, 2003.

[HO95] J. M. E. Hyland and C.-H. Luke Ong. Pi-calculus, dialogue games and PCF. In *FPCA*, pages 96–107, 1995.

[HO09] David Hopkins and C.-H. Luke Ong. Homer: A higher-order observational equivalence model checker. In *CAV*, pages 654–660, 2009.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Lai97] James Laird. Full abstraction for functional languages with control. In *LICS*, pages 58–67, 1997.

[MH98] Pasquale Malacaria and Chris Hankin. Generalised flowcharts and games. In *ICALP*, pages 363–374, 1998.

[Ong02] C.-H. Luke Ong. Observational equivalence of 3rd-order Idealized Algol is decidable. In *LICS*, pages 245–256, 2002.

[Rey81] John C. Reynolds. The essence of Algol. In *Proceedings of the 1981 International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[Ros98] A. W. Roscoe. *Theory and Practice of Concurrency.* Prentice-Hall, 1998.