A Concrete Presentation of Game Semantics

W. Blum[*] C.-H. L. Ong[†]

Abstract

We briefly present a new representation theory for game semantics which is very concrete: instead of playing in an arena game in which P plays the innocent strategy given by a term, the same game is played out over (a souped up version of) the abstract syntax tree of the term itself. The plays that are thus traced out are called *traversals*. More abstractly, traversals are the justified sequences that are obtained by performing parallel-composition *less* the hiding. After stating and explaining a number of Path-Traversal Correspondence Theorems, we present a tool for game semantics based on the new representation.

1 Introduction

In game semantics, programs are modelled as strategies (for the player P). Strategies, which are certain sets of *plays* (or *legal positions*), are composed by *parallel composition plus hiding*, in the sense of the process algebra CSP [6]. The starting point of our work is a kind of representation theory of the game semantics of higher-type programs (such as recursion schemes, PCF and Idealized Algol) that is very concrete, involving combinatorics over infinite structures defined by the abstract syntax trees of the programs being modelled. Take a program M which may be open. In this approach the strategy-denotation of M, written $[\![M]\!]$, is represented by a set $\mathcal{T}rv(M)$ of *traversals* over a possibly infinite tree – called the *computation tree* of M – which is generated from (a souped up version of) the abstract syntax tree of M. (Formally a traversal over a tree is a sequence of nodes starting from the root; quite unlike a path in the tree, a traversal can "jump" all over the tree, and may visit certain nodes infinitely often.) A traversal over the computation tree of M does not correspond to a play in $[\![M]\!]$, but rather to an *interaction sequence* that is obtained by *uncovering* [7] a play in $[\![M]\!]$ in a hereditary fashion; and a suitable projection of the traversals given by M – corresponding to the operation of hiding – gives the strategy-denotation $[\![M]\!]$. We call such a result a *Path-Traversal Correspondence Theorem.* (Denoting programs by sets of interaction sequences obtained by hereditary uncovering was first considered by Greenland in his DPhil thesis [4], which he has called *revealed semantics*.) The set $\mathcal{T}rv(M)$ is defined by recursion over the syntax of M and by rule induction. Intuitively these formation rules define what amounts to the composition algorithm of innocent strategies (less the hiding) but expressed in a setting in which moves (of the innocent game) are mapped to nodes of the computation tree. As a consequence of the representation, instead of the arena game in which P participates by playing the innocent strategy given by a term, the same game can be played out very concretely over (a souped up version of) the abstract syntax tree of the term itself (which we call the computation tree).

This view of game semantics and the theory of traversals were first developed to prove a theorem in the verification of infinite structures (namely, ranked trees generated by higher-order recursion schemes have decidable MSO theories, in [8]). The theory has at least two other applications. First it justifies the game characterization of the Higher-Order Matching Problem (but NOT the algorithm) developed by Colin Stirling. Secondly it underpins the (as yet unpublished) result that for each $n \geq 0$, the following are equi-expressive for generating ranked trees:

[*]Oxford University Computing Laboratory (OUCL), Wolfson Building, Parks Road, Oxford OX1 3QD, UK. william.blum@comlab.ox.ac.uk

[†]OUCL. Luke.Ong@comlab.ox.ac.uk

1. order-n recursion schemes (= order-n PCF terms generated from uninterpreted order-1 symbols)
2. order-n collapsible pushdown automata (CPDA)
3. order-n pure innocent strategies.

In this paper, after an introduction to traversals, we describe a tool that illustrates the representation theory, briefly demonstrating its capabilities. The tool can perform the inter-translations between CPDA and recursion schemes; it demonstrates the path-traversal correspondence in the form of a 2-player game in which the user can play against a given term interactively, over the computation tree of the term. Finally the tool is an aid to the type-setting of justified sequences.

2 Computation tree, Traversals and the Correspondence Theorem

In [8], one of us introduced the notion of computation tree and traversals over a computation tree for the purpose of studying trees generated by higher-order recursion scheme. Here we extend these concepts to the pure (*i.e.* without constants) simply-typed lambda calculus. Our setting allows the presence of free variable of any order. Moreover the term studied is not necessarily of ground type. (This contrasts with [8]'s setting where the term is of ground type and contains only *uninterpreted constant*[1] of order 1 at most and no free variables.) Note we automatically account for the presence of uninterpreted constants since they can be regarded as free variables. We will then state the *Correspondence Theorem* (Theorem 2.1).

In the following we fix a simply-typed term $\Gamma \vdash M : T$.

2.1 Computation tree

In [8] the computation tree of a grammar is defined as the unravelling of a finite graph representing the long transform of a grammar. Similarly we define the computation tree of a λ-term as an abstract syntax tree of its η-long normal form. We write $l\langle t_1, \ldots, t_n\rangle$ with $n \geq 0$ to denote the tree with a root labelled l with n children subtrees $t_1, \ldots, t_n$.

Definition 2.1. The ***computation tree*** $\tau(M)$ of a simply-typed term $\Gamma \vdash M : T$ with variable names in a countable set $\mathcal{V}$ is a tree with labels in $\{@\} \cup \mathcal{V} \cup \{\lambda x_1 \ldots x_n \mid x_1, \ldots, x_n \in \mathcal{V}, n \in \mathbb{N}\}$ defined from its η-long form as follows. Let $\overline{x}$ denote some list of variables $x_1 \ldots x_n$ for some $n \geq 0$.

$$\begin{array}{rcl} \text{For } m \geq 0,\ z \in \mathcal{V}\text{: } \tau(\lambda\overline{x}.zs_1 \ldots s_m) & = & \lambda\overline{x}\langle z\langle \tau(s_1), \ldots, \tau(s_m)\rangle\rangle \\ \text{For } m \geq 1\text{: } \tau(\lambda\overline{x}.(\lambda y.t)s_1 \ldots s_m) & = & \lambda\overline{x}\langle @\langle \tau(\lambda y.t), \tau(s_1), \ldots, \tau(s_m)\rangle\rangle\ . \end{array}$$

Even-level nodes are λ-nodes (the root is on level 0). A single λ-node can represent several consecutive variable abstractions or it can just be a *dummy lambda* if the corresponding subterm is of ground type. Odd-level nodes are variable or application nodes.

We say that a variable node n labelled x is ***bound*** by a node m, and m is called the ***binder*** of n, if m is the closest node in the path from n to the root such that m is labelled $\lambda\overline{\xi}$ with $x \in \overline{\xi}$.

We write $\circledast$ to denote the root of the computation tree $\tau(M)$. The set of nodes of the computation tree is denoted by N. The sets $N_@$, N_λ and N_{var} are the subset of N consisting of the @-nodes, λ-nodes and variable nodes respectively.

[1]A constant f is *uninterpreted* if the small-step semantics of the language does not contain any rule of the form $f \cdots \rightarrow e$. f can be regarded as a data constructor.

2.2 Justified sequences of nodes

We define the ***enabling relation*** on the set of nodes of the computation tree as follows: m enables n, written $m \vdash n$, if and only if n is bound by m (and we sometimes write $m \vdash_i n$ to precise that n is the i^{th} variable bound by m); or m is the root $\circledast$ and n is a free variable; or n is a λ-node and m is its parent node.

We say that a node n_0 of the computation tree is ***hereditarily enabled*** by $n_p \in N$ if there are nodes $n_1, \ldots, n_{p-1} \in N$ such that n_{i+1} enables n_i for all $i \in 0..p-1$.

For any set of nodes $S, H \subseteq N$ we write $S^{H\vdash}$ for $S \cap \vdash^* (H) = \{n \in S \mid \exists n_0 \in H \text{ s.t. } n_0 \vdash^* n\}$ – the subset of S constituted of nodes hereditarily enabled by some node in H. We will abbreviate $S^{\{n_0\}\vdash}$ into $S^{n_0\vdash}$.

We call ***input-variables nodes*** the elements of $V_{\mathsf{var}}^{\circledast\vdash}$ *i.e.* variables that are hereditarily enabled by the root of $\tau(M)$. Thus we have $V_{\mathsf{var}}^{\circledast\vdash} = V \setminus (V_{\mathsf{var}}^{N_@\vdash} \cup V_{\mathsf{var}}^{N_\Sigma\vdash})$.

A ***justified sequence of nodes*** is a sequence of nodes with pointers such that each occurrence of a variable or λ-node n different from the root has a pointer to some preceding occurrence m verifying $m \vdash n$. In particular, occurrences of @-nodes do not have pointer. We represent the pointer in the sequence as follows $\overset{\curvearrowleft^{i}}{m \ldots n}$. where the label indicates that either n is labelled with the ith variable abstracted by the λ-node m or that n is the i^{th} child of m. Children nodes are numbered from 1 onward except for @-nodes where it starts from 0. Abstracted variables are numbered from 1 onward. The i^{th} child of n is denoted by $n.i$.

We say that a node n_0 of a justified sequence is ***hereditarily justified*** by n_p if there are occurrences $n_1, \ldots, n_{p-1}$ in the sequence such that n_i points to n_{i+1} for all $i \in 0..p-1$. Let n be an occurrence in a justified sequence s. We write $s \upharpoonright r$ to denote the subsequence of s consisting of the occurrences hereditarily justified by n.

The notion of ***P-view*** $\ulcorner t \urcorner$ of a justified sequence of nodes t is defined the same way as the P-view of a justified sequences of moves in Game Semantics:[2]

$$\begin{array}{rcl@{\qquad}rcl}
\ulcorner \epsilon \urcorner & = & \epsilon & \ulcorner s \cdot \overset{\curvearrowleft}{m \cdot \ldots \cdot \lambda\overline{\xi}} \urcorner & = & \ulcorner s \urcorner \cdot \overset{\curvearrowleft}{m \cdot \lambda\overline{\xi}} \\
\text{for } n \notin N_\lambda,\ \ulcorner s \cdot n \urcorner & = & \ulcorner s \urcorner \cdot n & \ulcorner s \cdot \circledast \urcorner & = & \circledast
\end{array}$$

The O-view of s, written $\llcorner s \lrcorner$, is defined dually. We will borrow the game semantic terminology: A justified sequences of nodes satisfies ***alternation*** if for any two consecutive nodes, one is a λ-node and the other is not, and ***P-visibility*** if every variable node points to a node occurring in the P-view a that point.

2.3 Adding value-leaves to the computation tree

We now add another ingredient to the computation tree that was not originally used in [8]. Let $\mathcal{D}$ denote the set of values of the base type o. We add ***value-leaves*** to $\tau(M)$ as follows: For each value $v \in \mathcal{D}$ and for each node $n \in N$ we attach the child leaf v_n to n. We write V for the set of vertices of the resulting tree (*i.e.* inner nodes and leaf nodes). For \$ ranging in $\{@, \lambda, var\}$, we write $V_\$$ to denote the set of inner nodes from $N_\$$ plus the leaf-nodes with parent in $N_\$$ *i.e.* $V_\$ = N_\$ \cup \{v_n \mid n \in N_\$, v \in \mathcal{D}\}$.

Everything that we have defined can be lifted to this new version of computation tree. The enabling relation $\vdash$ is extended so that every leaf is enabled by its parent node. A link going from a value-leaf v_n to a node n is labelled by v: $\overset{\curvearrowleft^{v}}{n \ldots v_n}$. For the definition of P-view and visibility, value-leaves are treated as λ-nodes if they are at an odd level in the computation tree, and as variable nodes if they are at an even level.

We say that an occurrence of an inner node $n \in N$ is ***answered*** by an occurrence v_n if v_n in the sequence that points to n, otherwise we say that n is ***unanswered***. The last unanswered node is called the ***pending node***. A justified sequence of nodes is ***well-bracketed*** if each value-leaf occurring in it is justified by the pending node at that point.

[2] The equalities in the definition determine pointers implicitly. For instance in the second clause, if in the left-hand side, n points to some node in s that is also present in $\ulcorner s \urcorner$ then in the right-hand side, n points to that occurrence of the node in $\ulcorner s \urcorner$.

2.4 Traversals of the computation tree

A *traversal* is a justified sequence of nodes of the computation tree where each node indicates a step that is taken during the evaluation of the term.

Definition 2.2 (Traversals for simply-typed λ-terms)**.** The set $\mathcal{T}rv(M)$ of ***traversals*** over $\tau(M)$ is defined by induction over the rules of Table 1. A traversal that cannot be extended by any rule is said to be *maximal*.

Initialization rules

(Empty) $\epsilon \in \mathcal{T}rv(M)$.

(Root) The sequence constituted of a single occurrence of $\tau(M)$'s root is a traversal.

Structural rules

(Lam) If $t \cdot \lambda\overline{\xi}$ is a traversal then so is $t \cdot \lambda\overline{\xi} \cdot n$ where n denotes $\lambda\overline{\xi}$'s child and:

- If $n \in N_@ \cup N_\Sigma$ then it has no justifier;
- if $n \in N_{\mathsf{var}} \setminus N_{\mathsf{fv}}$ then it points to the only occurrence[a] of its enabler in $\ulcorner t \cdot \lambda\overline{\xi} \urcorner$;
- if $n \in N_{\mathsf{fv}}$ then it points to the only occurrence of the root $\circledast$ in $\ulcorner t \cdot \lambda\overline{\xi} \urcorner$.

(App) If $t \cdot @$ is a traversal then so is $$\begin{tikzcd}[column sep=small] t \cdot @ & \cdot\, n \arrow[l, bend right, "0"'] \end{tikzcd}.$$

Input-variable rules

(InputVar) If t is a traversal where $t^\omega \in N^{\circledast\vdash}_{\mathsf{var}} \cup L^{\circledast\vdash}_{\lambda}$ and x is an occurrence of a variable node in $\llcorner t \lrcorner$ then so is $t \cdot n$ for any child λ-node n of x, n pointing to x.

(InputValue) If $t_1 \cdot x \cdot t_2$ is a traversal with pending node $x \in N^{\circledast\vdash}_{\mathsf{var}}$ then so is $$\begin{tikzcd}[column sep=small] t_1 \cdot x & \cdot\, t_2 & \cdot\, v_x \arrow[ll, bend right, "v"'] \end{tikzcd}$$ for all $v \in \mathcal{D}$.

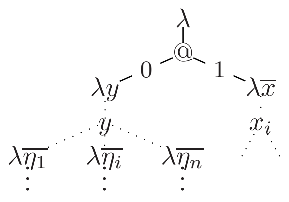
Copy-cat rules

(Var) If $$\begin{tikzcd}[column sep=small] t \cdot n \cdot \lambda\overline{x} & \ldots x_i \arrow[l, bend right, "i"'] \end{tikzcd}$$ is a traversal where $x_i \in N^{@\vdash}_{\mathsf{var}}$ then so is $$\begin{tikzcd}[column sep=small] t \cdot n & \cdot\, \lambda\overline{x} & \ldots x_i \arrow[l, bend right, "i"'] & \cdot\, \lambda\overline{\eta_i} \arrow[lll, bend right, "i"'] \end{tikzcd}.$$

(Value) If $$\begin{tikzcd}[column sep=small] t \cdot m \cdot n & \ldots v_n \arrow[l, bend right, "v"'] \end{tikzcd}$$ is a traversal where $n \in N$ then so is $$\begin{tikzcd}[column sep=small] t \cdot m & \cdot\, n & \ldots v_n \arrow[l, bend right, "v"'] & \cdot\, v_m \arrow[lll, bend right, "v"'] \end{tikzcd}.$$

Table 1: Traversal rules for the simply-typed λ-calculus.

[a]Prop. 2.1 will show that P-views are paths in the tree thus n's enabler occurs exactly once in the P-view.


λ
$@$
0
1
λy
$\lambda\overline{x}$
y
x_i
$\lambda\overline{\eta_1}$
$\lambda\overline{\eta_i}$
$\lambda\overline{\eta_n}$

A traversal always starts by visiting the root. Then it mainly follows the structure of the tree. The (Var) rule permits to jump across the computation tree. The idea is that after visiting a variable node x, a jump is allowed to the node corresponding to the subterm that would be substituted for x if all the β-redexes occurring in the term were reduced. The sequence $$\begin{tikzcd}[column sep=small] \lambda \cdot & @ & \cdot\, \lambda y \ldots & y \arrow[l, bend right, "1"'] & \cdot\, \lambda\overline{x} \arrow[lll, bend right, "1"'] & \ldots x_i \arrow[l, bend right, "i"'] & \cdot\, \lambda\overline{\eta_i} \ldots \arrow[lll, bend right, "i"'] \end{tikzcd}$$ is an example of traversal of the computation tree shown on the right.

Proposition 2.1 (counterpart of proposition 6 from [9])**.** *Let t be a traversal. Then:*

(i) t is a well-defined and well-bracketed justified sequence;

(ii) t is a well-defined justified sequence verifying alternation, P-visibility and O-visibility;

(iii) If t's last node is not a value-leaf, then $\ulcorner t \urcorner$ is the path in the computation tree going from the root to t's last node.

The ***reduction*** of a traversal t is the subsequence of t obtained by keeping only occurrences of nodes that are hereditarily enabled by the root $\circledast$. This has the effect of eliminating the "internal nodes" of the computation. If t is a non-empty traversal then the root $\circledast$ occurs exactly once in t and the reduction of t is equal to $t \upharpoonright r$. We write $\mathcal{T}rv(M)^{\upharpoonright\circledast}$ for the set or reduction of traversals of M.

Application nodes are used to connect the operator and the operand of an application in the computation tree but since they do not play any role in the computation of the term, we can remove them from the traversals. We write $t - @$ for the sequence of nodes-with-pointers obtained by removing from t all @-nodes and value-leaves of @-nodes; any link pointing to an @-node is replaced by a link pointing to the immediate predecessor of @ in t. We write $\mathcal{T}rv(M)^{-@}$ for the set $\{t - @ \mid t \in \mathcal{T}rv(M)\}$.

Remark 2.1. If M is β-normal then τ does not contain any @-node therefore all nodes are hereditarily justified by r and we have $\mathcal{T}rv(M)^{-@} = \mathcal{T}rv(M) = \mathcal{T}rv(M)^{\upharpoonright\circledast}$.

2.5 Revealed Game Semantics

In game semantics, strategy composition is achieved by performing a CSP-like "composition + hiding". If the internal moves are not hidden then we obtain an alternative semantics called *revealed semantics* in [4] and *interaction* semantics in [3]. Here we will refer to this semantics as the ***fully revealed game semantics***. The fully revealed game denotation of a term $\Gamma \vdash M : T$, written $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle$, is obtained by uncovering[3] all the internal moves from $[\![\Gamma \vdash M : T]\!]$ generated during strategy composition. We introduce a variation of the fully-revealed game denotation called the ***syntactically-revealed game denotation***, written $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle_{\mathsf{s}}$. In this denotation, as opposed to the fully-revealed denotation, only *certain* internal moves from $[\![\Gamma \vdash M : T]\!]$ are uncovered. More precisely, this denotation uncovers the internal moves that are generated by the composition with the evaluation map ev at @-nodes of the computation tree, but for term of the form $yN_1 \ldots N_p$ for some $p \geq 1$, the denotation hides the internal moves played by the copy-cat strategy denotating y. The formal definition can be found in [1].

2.6 Computation trees and arenas

We consider the well-bracketed game model of the simply-typed lambda calculus. We choose to represent strategies using "prefix-closed set of plays".[4] We fix a term $\Gamma \vdash M : T$ and write $[\![\Gamma \vdash M : T]\!]$ to denote its standard strategy denotation. The answer moves of a question q are written v_q where v ranges in $\mathcal{D}$.

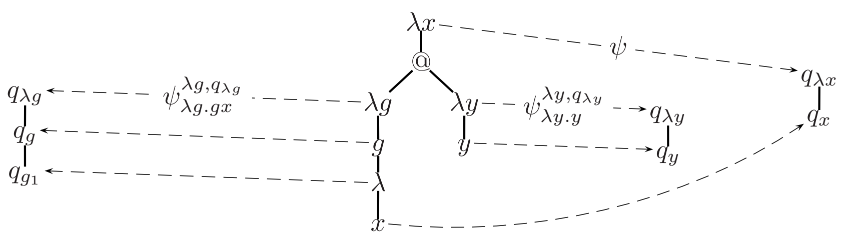
Proposition 2.2. *There exists a function φ_M, constructible from $\tau(M)$, that maps nodes from $V \setminus (V_@ \cup V_\Sigma)$ to moves of the interaction arena underlying the revealed strategy $\langle\!\langle \Gamma \vdash M : T \rangle\!\rangle_{\mathsf{s}}$ and such that φ maps λ-nodes to O-questions, variable nodes to P-questions, value-leaves of λ-nodes to P-answers and value-leaves of variable nodes to O-answers.*

Example 2.1. Take $\lambda x.(\lambda g.gx)(\lambda y.y)$ with $x, y : o$ and $g : (o, o)$. The diagram below represents the computation tree (middle), the arenas $[\![(o, o), o]\!]$ (left), $[\![o, o]\!]$ (right), $[\![o \to o]\!]$ (rightmost) and

[3] An algorithm that uniquely recovers hidden moves is given in Part II of [7].

[4] In the literature, a strategy is usually defined as a set of plays closed by *even*-length prefix. For the purpose of showing the Correspondence Theorem, however, the "prefix-closed"-based definition is more adequate.

$\varphi = \psi \cup \psi^{\lambda g, q_{\lambda g}}_{\lambda g.gx} \cup \psi^{\lambda y, q_{\lambda y}}_{\lambda y.y}$ (dashed-lines).



We extend the function φ to justified sequences of nodes as follows: If $t = t_0 t_1 \dots$ is a justified sequence of nodes in $V_\lambda \cup V_{\mathsf{var}}$ then $\varphi(t)$ is defined to be the sequence of moves $\varphi(t_0)\ \varphi(t_1)\dots$ equipped with the pointers of t.

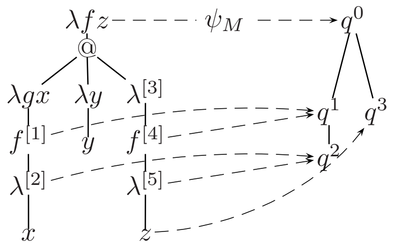
2.7 The Correspondence Theorem

In the simply-typed lambda calculus, the set $\mathcal{T}rv(M)$ of traversals of the computation tree is isomorphic to the set of plays of the syntactically-revealed denotation. Moreover the set of traversal reductions is isomorphic to the standard strategy denotation:

Theorem 2.1 (The Correspondence Theorem)**.** *The function φ_M gives us the following two isomorphisms:*

$$(i)\ \varphi_M \quad : \mathcal{T}rv(M)^{-@} \xrightarrow{\cong} \langle\!\langle \Gamma \vdash M : T \rangle\!\rangle_{\mathsf{s}}$$
$$(ii)\ \varphi_M \quad : \mathcal{T}rv(M)^{\upharpoonright \circledast} \xrightarrow{\cong} [\![\Gamma \vdash M : T]\!] \ .$$

Example 2.2. Take $M = \lambda f z.(\lambda g x.f x)(\lambda y.y)(f z) : ((o,o),o,o)$. The figure below represents the computation tree (left tree), the arena $[\![((o,o),o,o)]\!]$ (right tree) and ψ_M (dashed line). (Only question moves are shown for clarity.) The justified sequence of nodes t defined hereunder is an example of traversal:



$$t = \lambda f z\ @\ \lambda g x\ f^{[1]}\ \lambda^{[2]}\ x\ \lambda^{[3]}\ f^{[4]}\ \lambda^{[5]}\ z$$

$$t \upharpoonright r = \lambda f z\ f^{[1]}\ \lambda^{[2]}\ f^{[4]}\ \lambda^{[5]}\ z$$

$$\varphi_M(t \upharpoonright r) = q^0\ q^1\ q^2\ q^1\ q^2\ q^3 \in [\![M]\!] \ .$$

3 Presentation of the tool

We have developed a tool called HOG that permits one to visualize and explore the Traversal-Game Semantics Correspondence presented in the previous section. The binary files and sources in OCaml/F# can be downloaded from `http://web.comlab.ox.ac.uk/oucl/work/william.blum/`.

3.1 Generation of the computation graph/tree

HOG accepts two kind of objects as an input: simply-typed terms or higher-order recursion schemes. A recursion scheme is a special kind of higher-order grammar that can be used to generated an infinite tree called the value-tree (not to be confused with the computation tree). A recursion scheme can be thought of as a simply-typed term of ground type extended with recursion and containing uninterpreted constants of order at most 1 (corresponding to constructors for nodes of the value tree). Since a recursion scheme can make use of recursion, the computation tree becomes a computation *graph*. Traversals are however defined in the same way and the

Correspondence Theorem still holds. This setting was originally treated in [8] where the concepts of computation tree and traversal were first introduced.

HOG allows you to generate the computation tree of a higher-order simply-typed term as well as the computation graph of a higher-order recursion scheme.

3.2 Example

Take for instance the Urzyczin recursion scheme (see [2]). It is formally given by the tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where the set of terminals is $\Sigma = \{[: o \to o,] : o \to o, * : o \to o, 3 : o \to o \to o \to o, e : o, r : o\}$, the set of non-terminals is $\mathcal{N} = \{S : o, D : (o \to o \to o) \to o \to o \to o \to o, F : o \to o, E : o, G : o \to o \to o\}$ and the set of rules $\mathcal{R}$ is given by

$$\begin{array}{rcl} S & \to & [\,(D\,G\,E\,E\,E\,) \\ D\,\varphi x y z & \to & 3([(D\,(D\,\varphi\,x)\,z\,(F\,y)(F\,y)))(](\varphi\,y\,x))(*z) \\ F\,x & \to & *\,x \\ E & \to & e \\ G\,uv & \to & r \end{array}$$

Using the HOG syntax this is expressed as follows:

```
name { "Urzyczin tree" } validator { demiranda_urzyczyn } terminals{
    [:o-> o;
    ]:o -> o;
    *:o -> o;
    3:o -> o -> o -> o;
    e:o;
    r:o; }
nonterminals {
    S:o ;
    D:(o -> o -> o) -> o -> o -> o -> o ;
    F:o -> o ;
    E:o;
    G:o -> o -> o; }
rules {
    S = [ (D G E E E) ;
    D \varphi x y z = 3 ([ (D (D \varphi x) z (F y) (F y))) (] (\
        varphi y x)) (* z);
    F x = * x ;
    E = e ;
    G u v = r ; }
```

Figure 1 represents the computation graph. (The TEX source code for this graph is automatically generated by HOG.) The framed nodes correspond to the constant nodes.

3.3 Playing the traversal game

Once a computation tree/graph is loaded, HOG allows you to play a traversal game over it. The user plays for the Opponent while HOG plays for the Proponent. Figure 2 shows a screenshot of the tool when a traversal game is played.

The right-hand side of the window contains a representation of the computation graph and the left part of the window contains a list of justified sequences created by the user. The first justified sequence in the screenshot corresponds to a traversal that is currently being played by the user. Opponent moves and Proponent moves are represented by circles and rectangular nodes respectively. The user makes a move by selecting a node among the valid nodes highlighted in the computation graph. If there is a unique possible justifier in the traversal then the move is automatically accepted, otherwise the user has to specify a move by selecting a valid justifier in

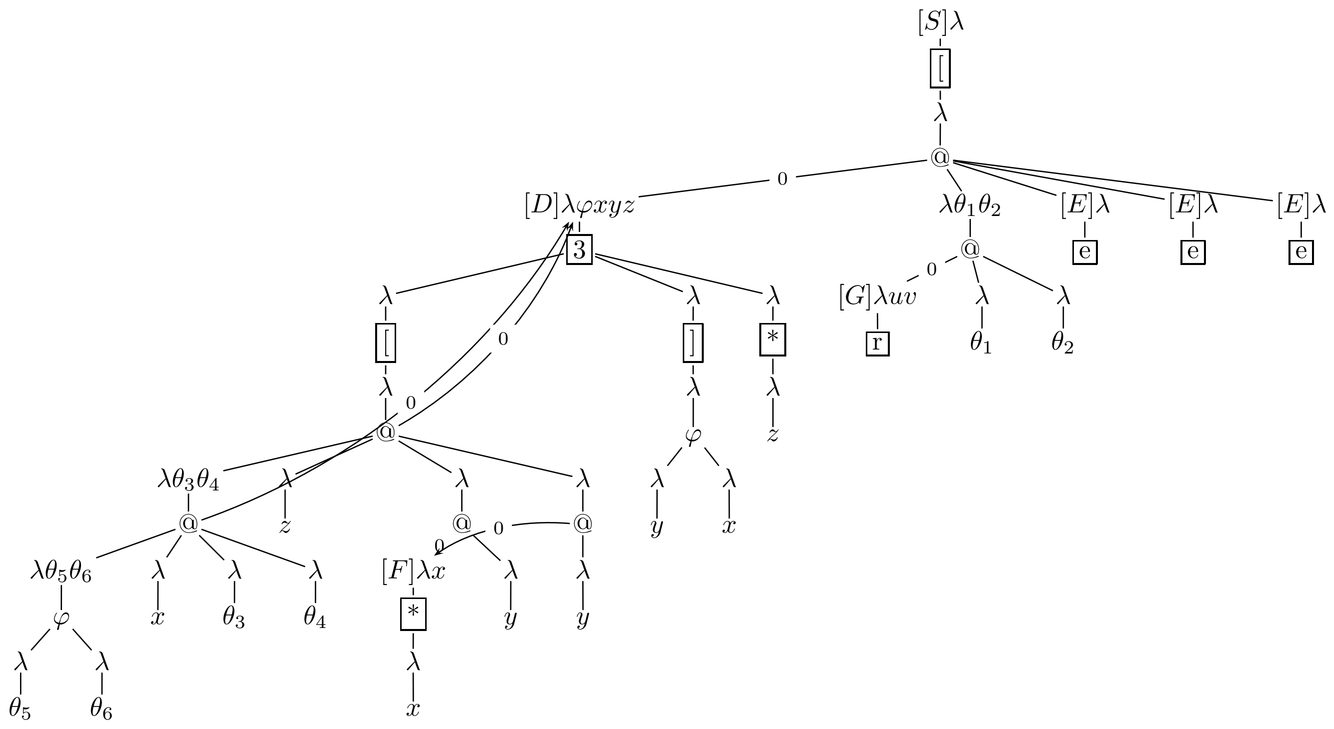
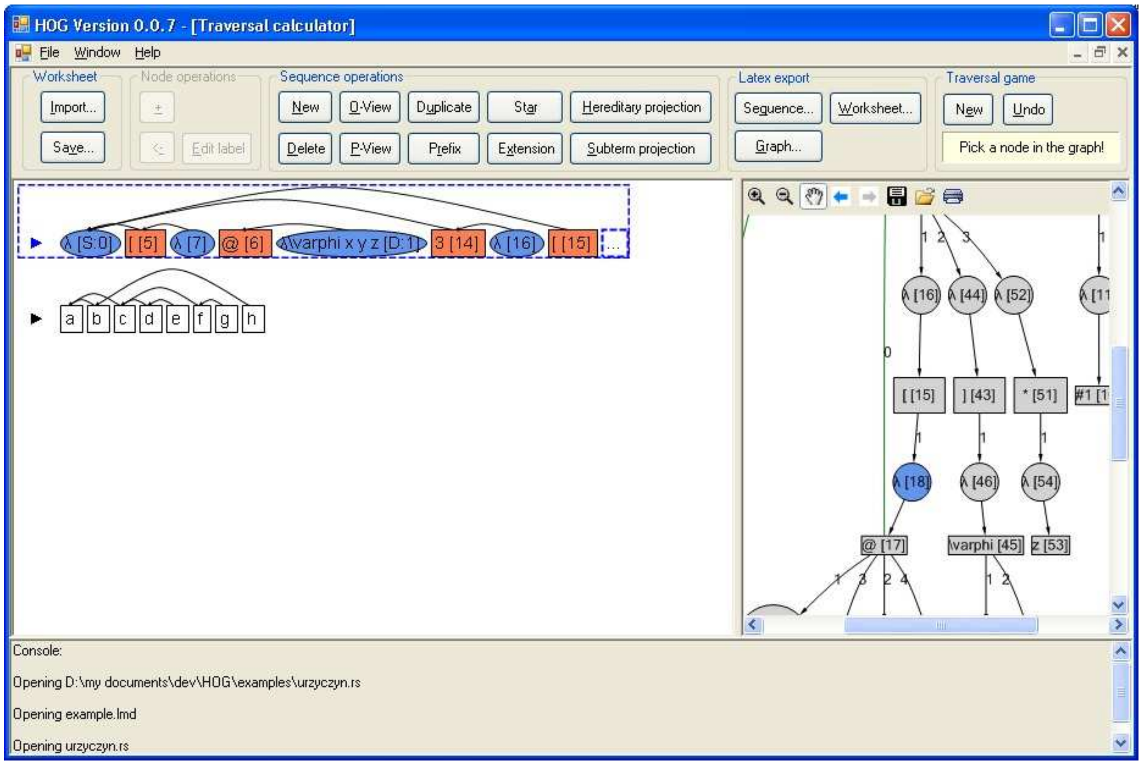


Figure 1: Computation graph of the Urzyczin's recursion scheme.



Opening D:\my documents\dev\HOG\examples\urzyczyn.rs

Opening example.lmd

Opening urzyczyn.rs

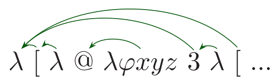
Figure 2: Screenshot of a traversal game in HOG.

```
Order: 2

Stack alphabet: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
     32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59

Code:
Code:

   0                      PUSH1 0 (0,0)
   1 start           :    CASETOP0 0->NODE0 1->NODE1 2->NODE2 3->NODE3 4->NODE4 5->NODE5 6->NODE6 7->
       NODE7 8->NODE8 9->NODE9 10->NODE10 11->NODE11 12->NODE12 13->NODE13 14->NODE14 15->NODE15 16->
       NODE16 17->NODE17 18->NODE18 19->NODE19 20->NODE20 21->NODE21 22->NODE22 23->NODE23 24->NODE24
       25->NODE25 26->NODE26 27->NODE27 28->NODE28 29->NODE29 30->NODE30 31->NODE31 32->NODE32 33->
       NODE33 34->NODE34 35->NODE35 36->NODE36 37->NODE37 38->NODE38 39->NODE39 40->NODE40 41->NODE41
       42->NODE42 43->NODE43 44->NODE44 45->NODE45 46->NODE46 47->NODE47 48->NODE48 49->NODE49 50->
       NODE50 51->NODE51 52->NODE52 53->NODE53 54->NODE54 55->NODE55 56->NODE56 57->NODE57 58->NODE58
       59->NODE59
   2 NODE0           :    PUSH1 5 (1,1)
   3                      GOTO start
   4 NODE1           :    PUSH1 14 (1,1)
   5                      GOTO start
   6 NODE2           :    PUSH1 55 (1,1)
   7                      GOTO start
   8 NODE3           :    PUSH1 58 (1,1)
   9                      GOTO start
  10 NODE4           :    PUSH1 59 (1,1)
  11                      GOTO start
  12 NODE5           :    EMIT [ NODE5_1
  13 NODE5_1         :    PUSH1 7 (0,0)
  14                      GOTO start
  15 NODE6           :    PUSH1 1 (1,1)
  16                      GOTO start
  17 NODE7           :    PUSH1 6 (1,1)
  18                      GOTO start
  19 NODE8           :    PUSH1 4 (1,1)
  20                      GOTO start
  21 NODE9           :    PUSH1 8 (1,1)
  22                      GOTO start
  23 NODE10          :    REPEAT 3 TIMES POP1
  24                      COLLAPSE
  25                      CASETOP0 45->NODE10_45 21->NODE10_21
  26                      FAILWITH "Unexpected top 0-element!"
  27 NODE10_21       :    PUSH1 24 (3,1)
  28                      GOTO start
 ...
```

Figure 3: Excerpt of the listing of the CPDA generated from the Urzyczin recursion scheme.

the traversal. After the move is accepted HOG responds with a P-move. This goes on until the traversal is maximal.

HOG allows the user to perform several operations on the justified sequence such as computing the P-view or the O-view. It is also possible to export the sequence to LaTeX code. For example, typesetting the traversal from Figure 2 with LaTeX produces


λ [λ @ $\lambda\varphi x y z$ 3 λ [...

3.4 Other features

3.4.1 Generating a CPDA from a recursion scheme

HOG can convert any higher-order recursion scheme into an equivalent (*i.e.* generating the same infinite tree) Collapsible Push-down Automaton (CPDA) of the same order. The algorithm used to perform this transformation is taken from [5].

The transition function of a CPDA is given by a list of instructions which we call the "code" of the CPDA. Figure 3.4.1 gives an example of CPDA generated from the Urzyczin recursion scheme.

A line of code of the CPDA is made of three parts: the line number, an optional label and an instruction. There are four kinds of instructions: stack instructions, the node emitting instruction, branching instructions and debugging instructions. The stack instructions are:

`PUSH1 a (j,k)` pushs the element `a` on the top 1-stack and associates the link (j,k) to it. This encoding means that the link points to a prefix stack obtained by performing an order-j pop k consecutive times.

`PUSHn` performs an order n push on the stack (*i.e.* duplicates the top $n-1$-stack).

`POPn` performs an order n pop on the stack (*i.e.* pop the top $n-1$-stack).

`COLLAPSE` collapses the stack to the prefix stack pointed to by the top element in the stack. In other words it executes k times the execution `POPj` where `(j,k)` is the encoding of the link associated to the top element.

`REPEAT n TIMES ins` repeats the instruction `ins` n times where n is a constant. The behaviour is unspecified if `ins` is a branching instruction.

In order to create nodes in the value-tree, the CPDA uses the following special instruction:

`EMIT f` $\mathtt{LAB_1} \ldots \mathtt{LAB_k}$ emits the terminal f of type $o^k \rightarrow o$. The CPDA is then spawn into k other CPDA's, one for each parameter of the terminal f. The $i's$ spawn CPDA will be started at instruction $\mathtt{LAB_i}$ for $i \in \{1..k\}$.

The branching instructions are:

`GOTO lab` jumps to the label `lab`.

`CASTOP0` $\mathtt{e_0} - > \mathtt{lab_0} ... \mathtt{e_k} - > \mathtt{lab_k}$ performs a test case on the element at the top of the stack. If the top element is equal to $\mathtt{e_i}$ for some $i \in \{0..k\}$ then the CPDA jumps to the label $\mathtt{lab_k}$. Otherwise it moves on to the following instruction.

There are also instructions used for debugging the code of a CPDA:

`FAILWITH msg` raises an exception with the message `msg`.

`ASSERT msg` asserts that the condition described by the message `msg` is verified. If the test fails then an exception is raised.

A configuration of the CPDA is given by an instruction number together with an order-n stack. In the initial configuration, the CPDA is positioned on the first instruction and the stack is the empty order-n stack. After executing a transition the CPDA moves to the next instruction (or jumps to another line if it is a branching instruction).

3.4.2 Exploring the value tree by executing the CPDA transitions

When you open a CPDA, HOG displays a lazy value tree (Figure 4). Each node of this tree corresponds to a configuration of the CPDA. Nodes are either labelled by an instruction number or by a terminal. In a separate textbox, HOG shows the content of the stack for the selected configuration.

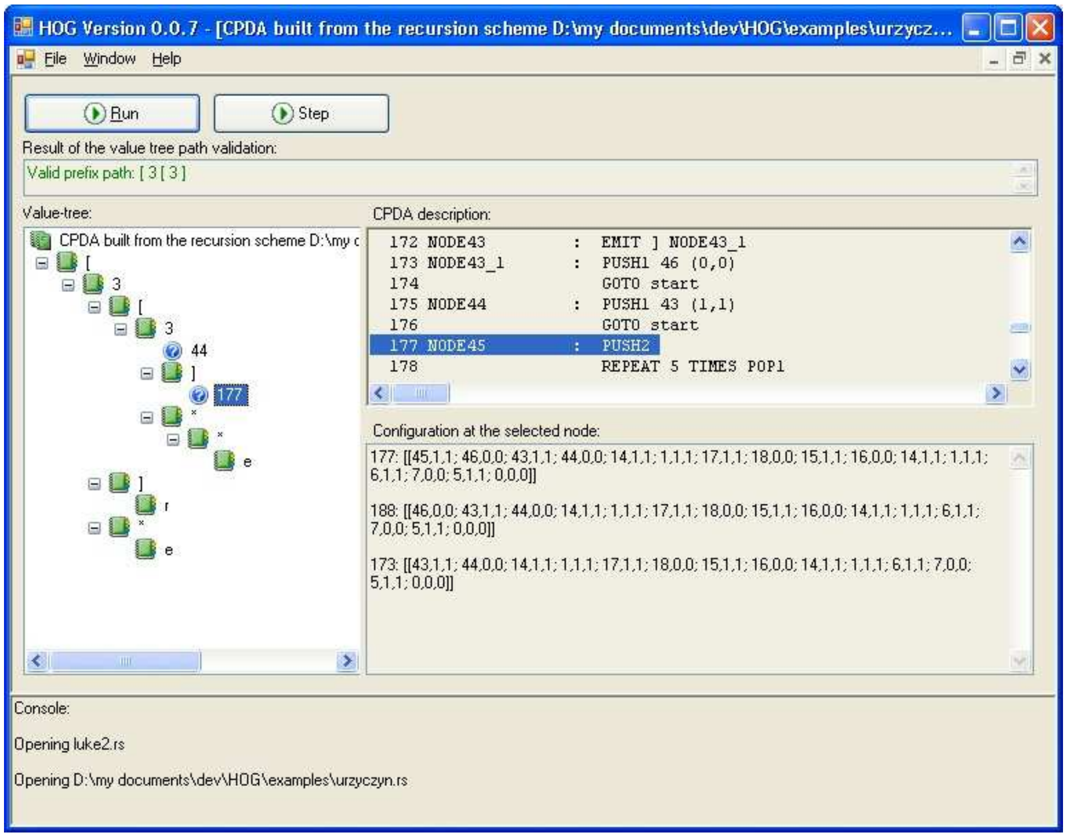
Initially, the root node is labeled with the instruction number 0 which corresponds to the first instruction of the CPDA code. When executing the CPDA transition on a node, the stack is updated accordingly and the node label is updated to the next instruction to be executed. This process is repeated until reaching an `EMIT` instruction.

When an instruction `EMIT f` $\mathtt{LAB_1} \ldots \mathtt{LAB_k}$ is executed (which means that a terminal $f : o^k \rightarrow o$ is emitted), a node labelled f is created in the value tree with k children nodes attached to it. Each child corresponds to a newly spawn CPDA with the same stack as in the original configuration. Finally, the current instruction of the ith child is set to the instruction number with label $\mathtt{LAB_i}$.

Terminal nodes represent the actual nodes of the value-tree and as such cannot be expanded further.

3.4.3 Exploring the tree generated by a recursion scheme

HOG has another feature which allows one to explore the infinite tree generated from a given recursion scheme without having to convert it into a CPDA beforehand.


HOG Version 0.0.7 - [CPDA built from the recursion scheme D:\my documents\dev\HOG\examples\urzycz...
File
Window
Help
Run
Step
Result of the value tree path validation:
Value-tree:
CPDA description:
Configuration at the selected node:
Console:

Valid prefix path: [3 [3]

```
CPDA built from the recursion scheme D:\my c
[
  3
    [
      3
        44
        ]
          177
        *
          *
            e
    ]
      r
    *
      e
```

```
172 NODE43          :  EMIT ] NODE43_1
173 NODE43_1        :  PUSH1 46 (0,0)
174                    GOTO start
175 NODE44          :  PUSH1 43 (1,1)
176                    GOTO start
177 NODE45          :  PUSH2
178                    REPEAT 5 TIMES POP1
```

```
177: [[45,1,1; 46,0,0; 43,1,1; 44,0,0; 14,1,1; 1,1,1; 17,1,1; 18,0,0; 15,1,1; 16,0,0; 14,1,1; 1,1,1;
6,1,1; 7,0,0; 5,1,1; 0,0,0]]

188: [[46,0,0; 43,1,1; 44,0,0; 14,1,1; 1,1,1; 17,1,1; 18,0,0; 15,1,1; 16,0,0; 14,1,1; 1,1,1; 6,1,1;
7,0,0; 5,1,1; 0,0,0]]

173: [[43,1,1; 44,0,0; 14,1,1; 1,1,1; 17,1,1; 18,0,0; 15,1,1; 16,0,0; 14,1,1; 1,1,1; 6,1,1; 7,0,0;
5,1,1; 0,0,0]]
```

```
Opening luke2.rs

Opening D:\my documents\dev\HOG\examples\urzyczyn.rs
```

Figure 4: HOG executing the CPDA generated from the Urzyczin recursion scheme.

References

[1] W. Blum and C.-H. L. Ong. Local computation of beta-reduction. Work In progress, 2008.

[2] Jolie G. de Miranda. *Structures generated by higher-order grammars and the safety constraint.* Dphil thesis, University of Oxford, 2006.

[3] Aleksandar Dimovski, Dan R. Ghica, and Ranko Lazic. Data-abstraction refinement: A game semantic approach. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *LNCS*, pages 102–117. Springer, 2005.

[4] Will Greenland. *Game Semantics for Region Analysis.* PhD thesis, University of Oxford, 2004.

[5] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursive schemes. November 2006. 13 pages, preprint.

[6] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[7] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, December 2000.

[8] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of IEEE Symposium on Logic in Computer Science.* Computer Society Press, 2006. Extended abstract.

[9] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes (technical report). Preprint, 42 pp, 2006.