A tool for constructing structures generated by higher-order recursion schemes and collapsible pushdown automata

William Blum

Oxford University Computing Laboratory

December 5, 2007

Abstract

This is a brief documentation for HOG(temporary name), a tool that allows one to explore the infinite tree generated by higher-order recursion schemes and collapsible pushdown automata. The executable files and sources in OCaml/F# can be downloaded from `http://web.comlab.ox.ac.uk/oucl/work/william.blum/`.

1 Background

The reader is referred to De Miranda's thesis [1] for an introduction to higher-order recursion schemes. Higher-order collapsible pushdown automata (CPDA for short) were introduced in [2]. The equivalence between the two devices is shown in the paper. The algorithm that is used by HOG to convert a recursion scheme to a CPDA is also described in it.

2 Features

HOG allows you to load a recursion scheme and observe the infinite tree generated by it. It can also convert it to an equivalent CPDA of the same order whose infinite tree can in turn be explored. You also have the possibility to convert the recursion scheme to an order-n PDA. The latter conversion is correct (in the sense that the same tree is generated) provided that the original recursion scheme satisfies the "safety restriction" [3]. (The proof of correctness will be written down soon).

The current version does not allow you to load your own CPDA as the parser has not been implemented yet. Consequently you can only create CPDAs by converting an equivalent recursion scheme.

3 Defining a recursion scheme

A recursion scheme is defined in a file with the .rs extension. The syntax is self-explanatory. As an example, consider the higher-order recursion scheme formally defined by $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where $\Sigma = \{f : o \to o \to o, a : o\}$, $\mathcal{N} = \{S : o, F : ((o \to o \to o) \to o \to o) \to o, H : ((o \to o \to o) \to o \to o) \to o \to o \to o, I : o \to o \to o \to o, G : (o \to o \to o) \to o \to o\}$ and

$$\begin{aligned} S &\to F\,G \\ F\,\psi &\to \psi\,(H\,\psi)\,a \\ H\,\psi\,x\,y &\to \psi\,(I\,x)\,y \\ I\,x\,u\,v &\to x \\ G\,\phi\,z &\to f\,(\phi\,(\phi\,z\,a)\,a)\,a \end{aligned}$$

The syntax for defining this recursion scheme in a .rs file is as follows:

```
// Commentary
name { "This is an example of tree-generating recursion scheme." }
validator { none } // ignore this
terminals {
    f:o->o->o;
    a:o;
}
nonterminals {
    S: o ;
    F: ((o -> o -> o)->o->o) -> o;
    H: ((o -> o -> o)->o->o) -> o -> o -> o;
    I: o -> o -> o -> o;
    G: (o -> o -> o)->o->o;
}
rules {
    S = F G;
    F psi = psi (H psi) a;
    H psi x y = psi (I x) y;
    I x u v = x;
    G phi z = f (phi (phi z a)a)a;
}
```

Note that the first rule must be the one corresponding to the start symbol S.

4 Usage

If you start the program with arguments, it will try to open the file passed as a parameter. Otherwise you can open a recursion scheme from the File\Open menu.

4.1 Recursion scheme

Once a recursion scheme is opened, a window appears containing a description of the recursion scheme on the right and the value tree on the left. Of course, since the value tree is potentially infinite, it cannot be represented entirely, therefore it is presented to the user in a lazy way: at the beginning, only the root of the tree is shown containing the start symbol S. You can expand a branch of the tree by double-clicking on the corresponding node. When doing this, the expression contained in the node is evaluated using the OI-strategy (the head-redex is reduced by performing substitution) and new nodes are created for each terminal occurring in head position in the new expression.

4.1.1 Computation graph

If you click on the button "Computation graph" it will open up a new window containing the computation graph of the recursion scheme (obtained by computing the η-long normal form, see [4]). Terminals are represented by rectangular nodes. Non-terminals are shown in yellow, variables in dark green. In white are the abstraction and application nodes. An example is given in Figure 1.

4.1.2 Conversion to CPDA/PDA

You can create the equivalent n-CPDA or n-PDA (provided that the recursion scheme is safe) by clicking on the appropriate button. This will open up a new window which allows you to explore the value-tree generated by the CPDA (see section 4.2).

4.2 CPDA

The CPDA window is designed similarly to the recursion scheme window. The value tree is shown on the left and the description of the CPDA on the right.

4.3 CPDA representation

The transition function of the CPDA is given by a list of instructions. A configuration of the CPDA is given by an instruction number together with an order n stack.

4.3.1 The CPDA code (the transition function)

As mentioned before, the transition function of the CPDA is encoded with a list of instructions also called the CPDA code. This code is shown on the right in the CPDA window. Each line of the CPDA code 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. Except for branching instructions, the CPDA executes the code in a linear way: after performing an instruction it moves on to
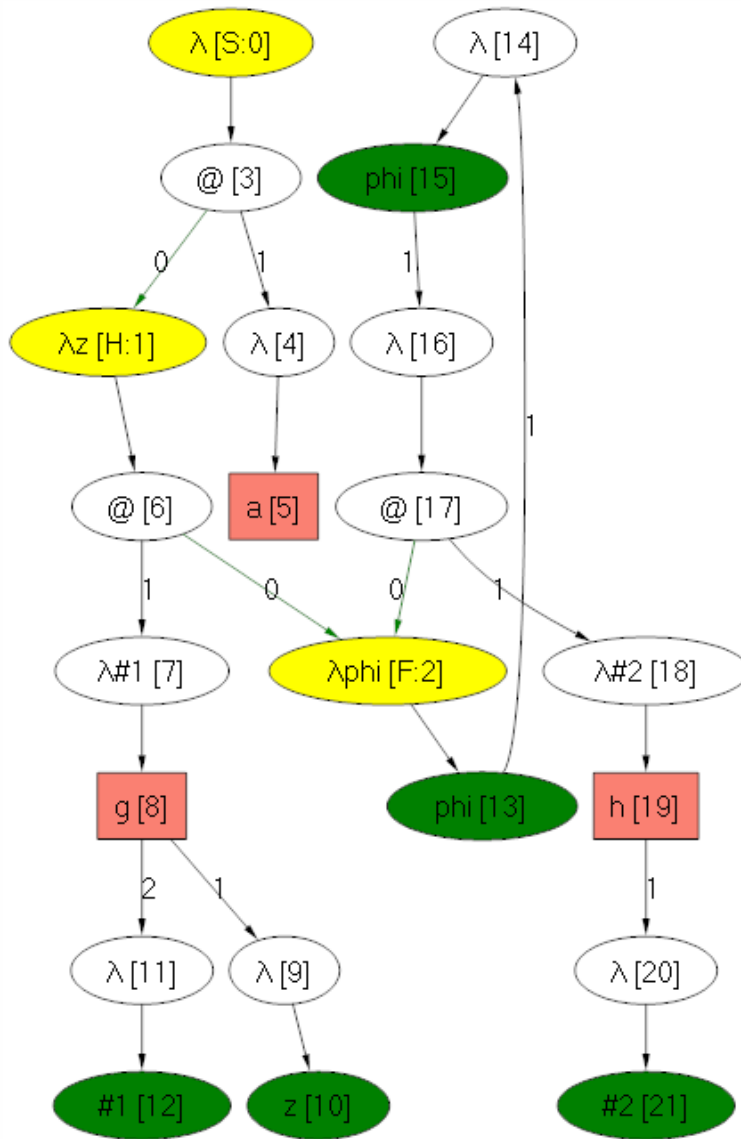
Figure 1: Computation graph generated from an order 2-recursion scheme

the following one. In the initial configuration, the CPDA is positioned on the first line of the CPDA code.

The stack instructions are:

PUSH1 a (j,k) push the element `a` on the top 1-stack an 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 (duplicates the top $n - 1$-stack).

POPn performs an order n pop on the stack (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.

For the purpose of creating trees, there is an instruction that permits the CPDA to create nodes:

EMIT f LAB$_1$ ... LAB$_k$ emits the terminal f of type $o^k \to o$. 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 LAB$_i$ for $i \in \{1..k\}$.

The branching instructions are:

GOTO lab jumps to the label `lab`.

CASTOPO e$_0$− > lab$_0$...e$_k$− > lab$_k$ performs a test case on the element at the top of the stack. If the top element is equal to e$_i$ for some $i \in \{0..k\}$ then the CPDA jumps to the label 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.

4.3.2 Example

This is an excerpt of the definition of the CPDA obtained by converting the recursion scheme given in section 3:

```
Order: 2

Terminals:
h:o −> o
```

```
g:o -> o -> o
a:o

Stack alphabet: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    17 18 19 20 21

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

 2 NODE0          :      PUSH1 3 (1,1)
 3                        GOTO start
 4 NODE1          :      PUSH1 6 (1,1)
 5                        GOTO start
 6 NODE2          :      PUSH1 13 (1,1)
 7                        GOTO start
 8 NODE3          :      PUSH1 1 (1,1)
 9                        GOTO start
10 NODE4          :      PUSH1 5 (1,1)
11                        GOTO start
12 NODE5          :      EMIT a // after this instruction the
    machine halts.
13 NODE6          :      PUSH1 2 (1,1)
14                        GOTO start
15 NODE7          :      PUSH1 8 (1,1)
16                        GOTO start
17 NODE8          :      EMIT g NODE8_1,NODE8_2
18 NODE8_1        :      PUSH1 9 (0,0)
19                        GOTO start
20 NODE8_2        :      PUSH1 11 (0,0)
21                        GOTO start
22 NODE9          :      PUSH1 10 (1,1)
23                        GOTO start
24 NODE10         :      REPEAT 5 TIMES POP1
25                        COLLAPSE
26                        CASETOP0 3->NODE10_3
27                        FAILWITH "Unexpected top 0-element!"
28 NODE10_3       :      PUSH1 4 (3,1)
29                        GOTO start
...
```

4.4 Exploring the value tree by executing the CPDA code

Each node of the (lazy) value tree corresponds to a configuration of the CPDA. Nodes are labelled either by an instruction number or by a terminal. When selecting a node, the content of the stack in the corresponding configuration is shown in the textbox situated at the bottom of the right-hand side of the window.

At the beginning, the root node is labeled with the instruction number 0 which corresponds to the first instruction of the CPDA code. When double-clicking on a node, the corresponding CPDA instruction is executed. The corresponding stack is updated accordingly and the node label is updated to the next instruction to be executed. This process can be repeated until an `EMIT` instruction is executed.

When a terminal $f : o^k \to o$ is emitted with the instruction `EMIT f LAB`$_1$`...LAB`$_k$, a node labelled f is created in the value tree and k children nodes are attached to it. Each child corresponds to a newly spawn CPDA with the same stack and such that the ith child's starts at the instruction labelled with `LAB`$_i$.

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

References

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

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

[3] T. Knapik, D. Niwiński, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FOSSACS'02*, pages 205–222. Springer, 2002. LNCS Vol. 2303.

[4] 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.