Stack Macro Tree Trasnducers

Kazuhiro Inaba

May 8, 2013

1 StackMTT

A stack macro tree transducer [EV85] is similar to a macro tree transducer, but its rules are in the following form.

$$F\ (\sigma\ x_1\ \ldots\ x_n)\ y_1\ \ldots\ y_m\ ys \to rhs$$

where

$$\begin{aligned} rhs ::= {} & y_i && (1 \le i \le m) \\ | {} & \delta\ rhs\ \ldots\ rhs \\ | {} & G\ x_i\ rhs\ \ldots\ rhs\ ys && (1 \le i \le n). \end{aligned}$$

Intuitively, the rule pops m (can be 0) trees from the stack, output them at y_i, and pushes some (can be 0) values when it inducively goes down to a subtree. The trailing parameter ys denotes the "untouched" part of the stack.

2 Towards Decomposing Unsafe HTTs

2.1 Overview

Claim 2.1. $n-\mathrm{HTT} \subseteq (\mathrm{StackMTT})^n$.

The original purpose of this note is to show the above claim, where $n-$HTT is the class of *unsafe* [KNU01, BO09] order-n tree transducers and the superscript ^{n} means the n-fold composition. This is the easy consequence of the following claim, which turned out to be hard to prove for $n > 2$ case. Thus, this note is currently devoted to show the original idea that still goes well with $n \le 2$ case, and records what is problematic in more higher-order cases.

Claim 2.2 (Main Lemma)**.** $n-\mathrm{HTT} \subseteq (n-1)-\mathrm{HTT}\ ;\ \mathrm{StackMTT}$.

More specifically, it claims that an $n-$HTT can be decomposed to an $(n-1)-$HTT followed by a particular translation called *Eval*, which performs first-order substitution symbolically represented as a tree. For people who are familiar with tree transducers, this can be thought as a generalization of the

`TOP`; `YIELD` decomposition of macro tree transducers (the generalization is needed to handle unsafe terms). For those from functional programming, this is a kind of *defunctionalization*, limited to only first order functions (the limitation is need to perform the evaluation with a single-input machine).

2.2 Eval

$\mathrm{Eval}_{X,Y}$ is a stack macro tree transducer running on input trees over the alphabet $\Delta \cup \{\mathtt{s}, \mathtt{z}, @\} \cup \{\mathtt{K_k D_d} \mid 0 \le k \le X, 0 \le d \le Y\}$. The set of rules of Eval is as follows.

$$
\begin{aligned}
\mathit{Eval}\ (\delta\ x_1 \ldots x_n)\ ys &\to \delta\ (\mathit{Eval}\ x_1, ys) \ldots (\mathit{Eval}\ x_n, ys) && \\
\mathit{Eval}\ (@\ x_1 x_2)\ ys &\to \mathit{Eval}\ x_1\ (\mathit{Eval}\ x_2\ ys)\ ys && \text{(push)} \\
\mathit{Eval}\ (\mathtt{z})\ y_1\ ys &\to y_1 && \text{(output-top)} \\
\mathit{Eval}\ (\mathtt{s}\ x_1)\ y_1\ ys &\to \mathit{Eval}\ x_1\ ys && \text{(pop)} \\
\mathit{Eval}\ (\mathtt{K_k D_d}\ x_1)\ y_1\ \ldots\ y_k\ y_{k+1}\ \ldots\ y_{k+d}\ ys &\to \mathit{Eval}\ x_1\ y_1\ \ldots\ y_k\ ys && \text{(drop slice)}
\end{aligned}
$$

Basically, it is meant to interprent a symbolic substitution tree whose variables are represented by de-Bruijn index in unary notation (e.g., $4 = \mathtt{ssssz}$). For example, a variable 4 will be substitued by the right child of 4th nearest binding node @. The tricky thing is the $\mathtt{K_k D_d}$ symbols which should be read (keep k and drop d). It is used to cleverly manage variable environments complicated by higher-order unsafe terms.

As a shorthand we will use the notation $\langle n \rangle = \overbrace{\mathtt{s}(\mathtt{s}(\cdots(\mathtt{s}}^{n}\,\mathtt{z})\cdots))$ for unary numerals.

2.3 Order Reduction

Types in unsafe HTT are represented by the following expression.

$$t ::= o \mid t \to t$$

where o denotes the order-0 type, i.e., output trees. The order-1 arity of a type t is:

$$
\begin{aligned}
\mathit{ary1}\,(t) &= p && \text{if } t = \overbrace{o \to o \to \cdots \to o}^{p} \to o \\
\mathit{ary1}\,(t) &= 0 && \text{otherwise.}
\end{aligned}
$$

Note, that for the particular purpose of this section, we care the arity only for order-1 types. For the higher-order types, *ary1* is always defined to be zero.

Now, each rule of a $n-$HTT

$$F\ v_1\ \ldots\ v_a\ y_0\ \ldots\ y_{m-1} \to \mathit{rhs}$$

where y_0 is the leftmost parameter such that all parameters to the right (including y_0 itself is order-0, is transformed to another rule of a $(n-1)-$HTT.

$$F\ v_1\ \ldots\ v_a\ \to [\![rhs]\!]_m$$

by $[\![\ \cdot\ ]\!]$, all order-1 entities in the right hand side is transformed to order-0 (i.e., trees). That effectively reduces the order of the whole transducer by 1. The subscript $_m$ intuively means that "from this context, we have to pop m values from the stack to reach the *caller* (of F) environment.

Please also be noted that variables v_i may still contain a variable whose order was originally 0, because we are dealing with unsafe transformations.

$$\begin{aligned}
[\![y_i]\!]_k &= \langle k-m+i\rangle && \text{(order-0 trailing parameter)}\\
[\![y]\!]_k &= \mathtt{K_0D_k}\ y && \text{(other order-0 parameters)}\\
[\![f]\!]_k &= \mathtt{K}_{ary1(\mathtt{f})}\mathtt{D}_{\mathtt{k}-ary1(\mathtt{f})}\ f && \text{(order-1 parameter)}\\
[\![\varphi]\!]_k &= D_k\ \varphi && \text{(order } \geq 2 \text{ parameters)}\\
[\![\delta]\!]_k &= \delta\ \langle 0\rangle\ \langle 1\rangle\ \ldots\ \langle ary1(\delta)\rangle && \text{(output symbol)}\\
[\![F]\!]_k &= F && \text{(nonterminals)}
\end{aligned}$$

where D_k is an auxiary nonterminal that inserts $\mathtt{K_0D_k}$ before applic $D_k\ \varphi\ params = \mathtt{K_0D_k}\ (v\ params)$. is this correct? no. Function application is coverted as a whole:

$$[\![e\ e'_1\ \ldots\ e'_p\ e_1\ \ldots\ e_n]\!]_k = @\ (@\ (\cdots(@\ (Z)\ [\![e_1]\!]_{k+n-1})\cdots)\ [\![e_{n-1}]\!]_{k+1})\ [\![e_n]\!]_k$$
$$\text{where } Z = [\![e]\!]_{k+n}\ [\![e'_1]\!]_{k+ary1(e'_1)}\ \cdots\ [\![e'_p]\!]_{k+ary1(e'_p)}$$

where e_p is the last non-zero order argument.

Claim 2.3. *By this construction,* $2-$HTT $\subseteq 1-$HTT; Eval.

Proof is by induction on the derivation steps in the $1-$HTT to say there is a exactly "corresponding" step in $2-$HTT.

2.4 Counterexample in the order-3 case

```
S     = T a
T x   = D (B x) b
D p x = F (p (F x)) x  # p :: (o->o)->o->o
F x y = c x y
B y f = f y            # f :: o->o
```

this translates to

```
S     = @ T a
T     = @ (D (B 0)) b
D p   = @ (@ F (!!!p!!! (@ F 1))) 0
F     = c 0 1
B y f = @ f y
```

During the evaluation of `p`, we want to forget about the immediately outer apprication and the parameter of `D` (two occurrences of `xs`) because what is substituted to p comes from the outer environment (it is `B 0 in T`). On the other hand, its argument, `@ F 1` must be evaluated without popping. The `1` refers to the `D`'s paramter `x`.

In other words, we need to apply different number of pop operations to p and its real arguments. *But*, even though we still know the number of necessary pops for p and each argument, we have no way to express the pop counts in the form of a single tree evaluatable by Eval.

2.5 Example in order-2 case.

This is a core of the *U language* written in an order-2 unsafe grammar [AdMO05].

```
D f x y z -> a (f y x) (D (D f x) z (F y) (F y))
```

This will become:

```
D f -> a (@ (@ (K2D3 f) 1+1) 0)
         [D (D f x) z (F y) (F y))]_3
     = a (@ (@ (K2D3 f) 1+1) 0)
         (@ (@ (@ (D [D f x]_5 2+2) (F 1+1)) (F 1))
     = a (@ (@ (K2D3 f) 1+1) 0)
         (@ (@ (@ (D (@ (D [f]_5) 5-3+0) 2+2) (F 1+1)) (F 1))
     = a (@ (@ (K2D3 f) 1+1) 0)
         (@ (@ (@ (D (@ (D (K2D3 f) 5-3+0) 2+2) (F 1+1)) (F 1))
     = a (@ (@ (K2D3 f) 2) 0)
         (@ (@ (@ (D (@ (D (K2D3 f) 2) 4) (F 2)) (F 1))
```

2.6 Comparison to other models.

Panic automata, collapsible pushdown automata, or many other machine models dealing with unsafe grammars, all has either a kind of *thunks* that points to code fragments, or *absolute pointers* to refer some point in stack absolutely.

The direction presented in this note is to use, instead of thunks or absolute links, *relative pointers* (like de-Bruijn index) to identify each variables environment. If it succeeds, it makes it possible to the Eval machine to operate more locally and tractable. (Though I have no luck yet :)).

References

[AdMO05] K. Aehlig, J. G. de Miranda, and C. H. L. Ong. Safety is not a restriction at level 2 for string languages. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 490–504, 2005.

[BO09] William Blum and C.-H. Luke Ong. The safe lambda calculus. *Logical Methods in Computer Science (LMCS)*, 5:1–38, 2009.

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–146, 1985.

[KNU01] Teodor Knapik, Damian Niwiński, and PawełUrzyczyn. Deciding monadic theories of hyperalgebraic trees. In *Typed Lambda Calculi and Applications (TLCA)*, pages 253–267, 2001.