COMP 523: Language-based security
Assignment 5 (100 points total)

Prof. B. Pientka
McGill University

Due: 30 Nov 2017 at 2:35pm

1 (Co)Pattern Matching using Stacks [35 points]

We revisit here the discussion regarding (co)pattern matching using (co)recursive types. In class we described an abstact machine that modelled the call stack using continuations for describing the operational semantics for a language supporting (co)pattern matching. The development was based on two research papers [APTS13] and [TCP16].

Here we revisit this idea and give a short summary of some of the core ideas from these papers that were discussed in class below. Your task is to extend the abstract machine semantics to handle pairs and terms build using constructors c_i.

$$
\begin{array}{lcl}
\text{Types } T, S & := & \ldots \mid T \to S \mid T \times S \mid \mu X.D \mid \nu X.R \\
\text{Variants } D & := & \langle c_1 : T_1; \ldots, c_k : T_k \rangle \\
\text{Records } R & := & \{d_1 : T_1; \ldots; d_n : T_n\}
\end{array}
$$

$$
\begin{array}{lcl}
\text{Terms } t, s & := & \ldots \mid x \mid \mathsf{fn}\ p \to t \mid t\ s \mid (t, s) \mid c_i\ t \mid d_i\ t \\
\text{Patterns } p & := & x \mid (p_1, p_2) \mid c_i\ p \\
\text{Copattern } q & := & \cdot \mid p\ q \mid .d_i\ p
\end{array}
$$

$$
\text{Stack } E \quad := \quad \cdot \mid (s\ \bullet)\ E \mid \nu E \mid .d\ E \mid \ldots
$$

Recall that we can define finite data such as lists or numbers using recursive types ($\mu X.D$) while we define infinite data such as streams using corecursive types ($\nu X.R$).

$$
\begin{array}{lcl}
\mathsf{Nat} & = & \mu X.\langle \mathsf{z} : \mathsf{unit}\ ;\ \mathsf{suc} : X \rangle \\
\mathsf{List} & = & \mu X.\langle \mathsf{nil} : \mathsf{unit}\ ;\ \mathsf{cons} : \mathsf{Nat} \times X \rangle \\
\mathsf{Stream} & = & \nu X.\{\mathsf{hd} : \mathsf{Nat}\ ;\ \mathsf{tl}\ :\ X\}
\end{array}
$$

We can then define a stream $0,\ 1,\ 2,\ 0,\ 1,\ 2,\ 0,\ 1, \ldots$ that cycles through 0, 1, and 2 as follows.

$$
\begin{array}{lllll}
\mathsf{rec\ cycle.\ fn} & \mathsf{x}\ .\mathsf{hd} & \to & \mathsf{x} \\
\mid & \mathsf{z}\ .\mathsf{tl} & \to & \mathsf{cycle}\ (\mathsf{suc}(\mathsf{suc}\ \mathsf{z})) \\
\mid & (\mathsf{suc}\ \mathsf{x})\ .\mathsf{tl} & \to & \mathsf{cycle}\ \mathsf{x}
\end{array}
$$

In particular, we describe the rules for the continuation-based abstact machine below. To illustrate the idea, consider applications $s\ t$. To evaluate an application $s\ t$ together with a stack E, we evaluate the term t and save on the stack $(s\ \bullet)$. When t is a value, we pop $(s\ \bullet)$ off the stack. To evaluate $s\ v$, we push the value v on the stack E and continue evaluating s. When making an observation about a term t, we push the observation onto the stack E.

Configuration $t\ ;\ E$ steps to configuration $t'\ ;\ E'$

$$\begin{array}{lcll}
(s\ t)\ ;\ E & \longrightarrow & t & ;\ (s\ \bullet)\ E \\
v\ ;\ (s\ \bullet)\ E & \longrightarrow & (s\ v)\ ; & E \\
(s\ v)\ ;\ E & \longrightarrow & s & ;\ v\ E \\
t.d\ ;\ E & \longrightarrow & t & ;\ .d\ E
\end{array}$$

When we evaluate a function $\mathsf{fn}\ q \longrightarrow t$, we pop check the call stack E and pop off the right number of observations and values, i.e. there exists a pattern instantiation σ which when applied to q gives us a sequence of observations and values. This sequence must match what is on top of the call stack E.

$$\frac{E = [\sigma]q@E'}{(\mathsf{fn}\ q \to t)\ ;\ E \longrightarrow [\sigma]t\ ;\ E'}$$

How do we type stacks E? – We use the judgment $T \vdash E : S$ for stack typing. Intuitively, it means that if given a value of type T, we can pass it to the stack; the final result computed by the call stack is of type S.

$$\frac{\vdash s : T_1 \to T_2 \quad T_2 \vdash E : S}{T_1 \vdash (s\ \bullet)\ E : S} \qquad \frac{\vdash v : T_1 \quad T_2 \vdash E : S}{T_1 \to T_2 \vdash v\ E : S} \qquad \frac{R_d[\nu X.R/X] \vdash E : S}{\nu X.R \vdash .d\ E : S}$$

To type a configuation $(t\ ;\ E)$, we require that the term t has type T and the stack expects a value of type T and ultimately returns a value of type S.

$$\frac{\vdash t : T \quad T \vdash E : S}{t\ ;\ E : S}$$

5 pts Exend the definition of stacks to also allow us to store intermediate expressions when we evaluate pairs and finite data.

10 pts Extend the rules for stepping configurations to also step pairs and finite data.

10 pts Extend the typing rules for stacks to handle the additional intermediate states.

10 pts Prove type preservation for your additional configurations.

optional Revisit the case for (co)pattern matching in the type preservation proof; state and prove the necessary lemmas following the sketch given in class.

2 Let-polymorphism (65 points)

We explore two alternatives to allow for let-polymorphism considering the expression $\mathsf{let\ name}\ x = e_1\ \mathsf{in}\ e_2$. In the first approach, we check that expression e_1 is well-typed and then infer the type for $[e_1/x]e_2$.

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash [e_1/x]e_2 : T}{\Gamma \vdash \mathsf{let\ name}\ x = e_1\ \mathsf{in}\ e_2 : T}$$

The disadvantage of this approach is that we are type-checking the expression e_1 multiple times and it is not a realistic basis for an implementation. In the second approach, we rely on *type schemas*.

$$\begin{array}{lrcl} \text{Types} & T & := & \mathsf{nat} \mid \mathsf{bool} \mid \alpha \mid T_1 \to T_2 \mid T_1 * T_2 \\ \text{Type schema} & S & := & T \mid \forall \alpha.S \end{array}$$

Type schemas are related to types through instantiation, written as $S \preceq T$ and defined as follows:

$$\frac{}{T \preceq T}\ \mathtt{inst{-}base} \qquad \frac{[T'/\alpha]S \preceq T}{\forall \alpha.S \preceq T}\ \mathtt{inst{-}all}$$

We then modify the typing rule for let-name as follows:

$$\frac{\Gamma \vdash e_1 \triangleright S \quad \Gamma, x \triangleright S \vdash e_2 : T}{\Gamma \vdash \mathsf{let\ name}\ x = e_1\ \mathsf{in}\ e_2 : T}\ \mathtt{tp{-}letn} \qquad \frac{\Gamma(x) = S \quad S \preceq T}{\Gamma \vdash x : T}\ \mathtt{tp{-}var}$$

$$\text{Typing context} \quad \Gamma \ := \ \cdot \mid \Gamma, x \triangleright S$$

Type schemas can be derived for expressions by quantifying over the free type variables.

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash e \triangleright T}\ \mathtt{tpsc{-}base} \qquad \frac{\Gamma \vdash e : S}{\Gamma \vdash e \triangleright \forall \alpha.S}\ \mathtt{tpsc{-}all}^{\alpha}$$

Note that the premise of the rule $\mathtt{tpsc{-}all}$ is parametric in α, i.e. α is not free in Γ. For the proofs below, we only consider the functions, function application and let-name; all other constructs behave orthogonally.

20 pts Prove type preservation for the new language considering big-step evaluation. Carefully state and prove the necessary substitution lemma.

5 pts State the theorem which asserts the equivalence between the two approaches.

15 pts Prove the easy direction of this equivalence.

25 pts Consider the "harder" direction of the equivalence proof.

References

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *40th ACM Symp. on Principles of Programming Languages (POPL'13)*, pages 27–38. ACM Press, 2013.

[TCP16] David Thibodeau, Andrew Cave, and Brigitte Pientka. Indexed codata. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, pages 351–363. ACM, 2016.