Assignment 2:
Cost Semantics and Type Inference

15-819: Foundations of Quantitative Program Analysis (Fall 2019)

Out: Wednesday, September 18, 2019
Due: Tuesday, October 1, 2019 11:59pm EDT

1 High-Water Mark

Recall the two structural cost semantics that we defined in lecture. The structural semantics with resource effects is defined by the judgment

$$M \vdash e \longmapsto^* e' \mid q \, .$$

The intended meaning is that e steps to e' with cost q under metric M. The structural semantics with resource safety is defined by the judgment

$$M \vdash \langle e \mid q \rangle \longmapsto^* \langle e' \mid q' \rangle \, .$$

It states that, with $q \in \mathbb{Q}_{\geq 0}$ available resources, expression e evaluates to expression e' and $q' \in \mathbb{Q}_{\geq 0}$ available resources.

Both judgments can be used to define a notion of high-water mark resource usage.

Definition 1. *Let $e : \tau$ be a closed expression. The high-water mark resource usage with resource effects $\mathcal{E}_M(e)$ of expression e under metric M is* $\max\{q \mid M \vdash e \longmapsto^* e' \mid q\}$ *if the maximum exists and ∞ otherwise.*

Definition 2. *Let $e : \tau$ be a closed expression. The high-water mark resource usage with resource safety $\mathcal{S}_M(e)$ of e under metric M is defined as the smallest q such that the following holds (or ∞ if no such q exists). For all e', q', if $\langle e \mid q \rangle \longmapsto^* \langle e' \mid q' \rangle$ and not e'* val *then $\langle e' \mid q' \rangle \longmapsto \langle e'' \mid q'' \rangle$ for some e'', q''*

The two definitions are equivalent.

Theorem 1. *Let $e : \tau$ be a closed expression and M a metric. Then $\mathcal{S}_M(e) = \mathcal{E}_M(e)$*

Task 1.1 (30 pts). Prove Theorem 1.

2 Let-Normal Form

In this problem, you will convert expressions into *let-normal form* without altering the resource consumption as defined by the cost semantics.

Recall our simple functional language from the cost semantics lecture. For this problem, we extend it with let binding and ticks.

$$
\begin{array}{lcll}
e & ::= & x & x \\
& & \mathrm{app}(e_1; e_2) & e_1(e_2) \\
& & \mathrm{lam}\{\tau\}(x.e) & \lambda(x:\tau)e \\
& & \mathrm{fix}\{\tau\}(x.e) & \mathsf{fix}\ x\ \mathsf{as}\ e \\
& & \mathrm{triv} & \langle\rangle \\
& & \mathrm{tick}\{q\} & \mathsf{tick}\,(q) \\
& & \mathrm{let}(e_1; x.e_2) & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2
\end{array}
$$

The static semantics of the added syntactic forms is defined by the following rules.

$$
\frac{}{\Gamma \vdash \mathrm{tick}\{q\} : \mathrm{unit}}\ (\text{T:Tick})
\qquad
\frac{\Gamma \vdash^m e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash^m e_2 : \tau}{\Gamma \vdash^m \mathrm{let}(e_1; x.e_2) : \tau}\ (\text{T:Let})
$$

Here, we consider the evaluation dynamics with resource safety, that is, the judgment $M \vdash e \Downarrow^{q}_{q'} v$. We extend it as follows.

$$
\frac{q' = q - M(\mathsf{tick}_q) \geq 0}{M \vdash \mathrm{tick}\{q\} \Downarrow^{q}_{q'} \mathrm{triv}}\ (\text{Es:Tick})
$$

$$
\frac{M \vdash e_1 \Downarrow^{p}_{r} v_1 \qquad M \vdash [v_1/x]e_2 \Downarrow^{r}_{q'} v \qquad p = q - M(\mathsf{let}) \geq 0}{M \vdash \mathrm{let}(e_1; x.e_2) \Downarrow^{q}_{q'} v}\ (\text{Es:Let})
$$

Metrics are mappings

$$
M : (\{\mathsf{var}, \mathsf{app}, \mathsf{lam}, \mathsf{fix}, \mathsf{trv}, \mathsf{let}\} \cup \{\mathsf{tick}_q \mid q \in Q\}) \to \mathbb{Q}
$$

For the remainder of the problem, we consider a fixed but arbitrary metric M.

Definition 3. *The* tick metric M_T *is the metric with* $M_T(\mathsf{tick}_q) = q$ *and* $M_T(K) = 0$ *for* $K \neq \mathsf{tick}_q$.

Definition 4 (Let-Normal Form)**.** *We say that an expression e is in* let-normal form *if it is derived using the following grammar.*

$$
\begin{array}{lcl}
\bar{e} & ::= & x \\
& & \mathrm{app}(x_1; x_2) \\
& & \mathrm{lam}\{\tau\}(x.\bar{e}) \\
& & \mathrm{fix}\{\tau\}(x.\bar{e}) \\
& & \mathrm{triv} \\
& & \mathrm{tick}\{q\} \\
& & \mathrm{let}(\bar{e}_1; x.\bar{e}_2)
\end{array}
$$

The idea of let-normal form is apply syntactic forms to variables only instead of expression whenever we can do so without limiting the expressivity of the language. Let-normal form simplifies resource analysis because it limits sequential composition to the let rule and makes the evaluation order of subexpression explicit. For our simple language, the only change is that function applications $\mathrm{app}(x_1; x_2)$ are restricted to variables.

We now Define a translation judgment $\Gamma \vdash e \rightsquigarrow \bar{e} : \tau$ that relates expressions e such that $\Gamma \vdash e : \tau$ to expressions $\bar{e}$ in let-normal form such that the following theorem holds.

Theorem 2. *If* $\Gamma \vdash e \rightsquigarrow \bar{e} : \tau$ *and* $\Gamma \vdash e : \tau$ *then*

- $\bar{e}$ *is in let-normal form,*
- $\Gamma \vdash \bar{e} : \tau$*, and*
- *for all* q *and* q'*,* $M \vdash e \Downarrow^{q}_{q'} v$ *if and only if* $M_T \vdash \bar{e} \Downarrow^{q}_{q'} v$

The third part of the theorem only applies to closed expressions, that is, when $\Gamma = \cdot$ is empty. In this case, the theorem states that $\bar{e}$ evaluates to the same value as e and the cost of $\bar{e}$ under the tick metric M_T is identical to the cost of e under the fixed metric M.

Below are the first two translation rules.

$$\frac{t \text{ fresh} \qquad q = M(\mathsf{var})}{\Gamma, x : \tau \vdash x \rightsquigarrow \mathrm{let}(\mathrm{tick}\{q\}; t.x) : \tau}\ (\text{LN:Var})$$

$$\frac{t \text{ fresh} \qquad q = M(\mathsf{lam}) \qquad \Gamma, x : \tau \vdash e \rightsquigarrow \bar{e} : \tau'}{\Gamma \vdash \mathrm{lam}\{\tau\}(x.e) \rightsquigarrow \mathrm{let}(\mathrm{tick}\{q\}; t.\mathrm{lam}\{\tau\}(x.\bar{e})) : \tau \to \tau'}\ (\text{LN:Abs})$$

Task 2.1 (15 pts). Complete the definition of $\Gamma \vdash e \rightsquigarrow \bar{e} : \tau$ by giving translation rules for the other syntactic forms (5 rules).

You do not have to write down the complete proof of Theorem 2. However, let us at least think about how we would approach the proof. A good idea is to prove the theorem by rule induction on the typing $\Gamma \vdash e : \tau$. The interesting case arises if e is a closed expression. Then we proceed with an inner rule induction on $M \vdash e \Downarrow^{q}_{q'} v$ to show

$$M \vdash e \Downarrow^{q}_{q'} v \ \text{ iff } \ M_T \vdash \bar{e} \Downarrow^{q}_{q'} v$$

However, in the case of the rule Es:Let we run into a problem.

Task 2.2 (5 pts). Describe the problem that arises in the proof for the case Es:Let.

This shows that the let-normal form does not work well in conjunction with dynamic semantics that use substitution. We will later see how we can work around this issue. [1]

[1] A-normal form is a more general version of let-normal form that avoid this issue.

3 Unification in OCaml

In this problem, you will implement an efficient unification algorithm in OCaml. OCaml is very similar to SML. Many of the differences are only syntactic. The following website does a good job listing the key differences:

`http://adam.chlipala.net/mlcomp/`

OCaml is available for all major platforms. You can find installation instructions and other resources at

`https://ocaml.org`

A naive implementation of the declarative unification rules that we discussed in the lecture can be quite inefficient. To make type inference efficient, the developers of the OCaml compiler took the following approach. You can find a skeleton of the implementation in the file `unify.ml`.

First, types are represented by the following mutually recursive type.

```
type type_exp =
  { mutable desc : type_desc; mutable mark : int }

and type_desc =
    Var
  | Arr of type_exp * type_exp
  | Pair of type_exp * type_exp
  | Link of type_exp
```

The record type `type_exp` contains to mutable fields. Mutable fields are a convenient short form and equivalent to fields that contain references. See `https://realworldocaml.org/v1/en/html/records.html`. (In fact, references are derived from mutable records in OCaml.) We have also added pairs that we didn't discuss in lecture. They are treated similar to arrow types.

The mutual recursion in the type is a common pattern in OCaml code. It makes it more convenient to access (and update) data that is shared among all constructors of an inductive type. The type $a * a \to b$ can be represented as follows. Note that type-variable names are implicitly represented by "heap addresses".

```
let type_exp d : type_exp = {desc = d; mark = 0} in

let a = type_exp Var in
let b = type_exp Var in
let s = type_exp (Pair(a, a)) in
type_exp (Arr(s, b))
```

Values of type `type_exp` can be used as a union-find data structure.

```
(* union find operations *)
let rec find (t : type_exp) : type_exp =
    match t.desc with
    | Link l →
        let r = find l
        in t.desc ← Link r;
           r
    | _ → t

let union (t1 : type_exp) (t2 : type_exp) : unit =
  (find t1).desc ← Link (find t2)
```

Recall the rule U:ELIM2:

$$\frac{\sigma = t \mapsto \tau \qquad t \not\in \mathrm{Var}(\tau)}{\langle t,\tau\rangle, C \implies \langle t,\tau\rangle, [\sigma]C}\ \text{(U:ELIM2)}$$

As the first step of OCaml's unification algorithm, the rule E:ELIM2 is basically implemented by a call to the function `union` without checking the side condition $X \not\in \mathrm{Var}(T)$. In this way, a graph structure is created on the heap. The record field `mark` is not used in this first step. In the second, step the graph structure is traversed to identify cycles. If a cycle is found then the types are not unifyable. Otherwise, the unifyied type can be derived by following the pointers on the heap (see function `string_of_type` in `unify.ml`).

Task 3.1 (18 pts). Implement OCaml's efficient unification algorithm by completing the OCaml file `unify.ml` that is available on the web page.

Task 3.2 (5 pts). Argue informally (and briefly) why this approach is equivalent to the declarative rules presented in the lecture.

Task 3.3 (12 pts). In your solution, provide three (different) pairs of (different) values of type `type_exp` that are not unifyable. Similarly, provide three pairs of values of type `type_exp` that are unifyable.

4 Typos in Lecture Notes

Task 4.1 (10 pts). You get 2 bonus points for every typo you find in the lecture notes. You can not get more than 10 bonus points per assignment.