# Resource Analysis: Problem Set 4

Jan Hoffmann
Carnegie Mellon University

February 15, 2016

Due before 1:30pm on Monday, February 22

## 4.1 (12 Points) Solved Forms and MGUs

Recall the following definitions from the lecture.

**Definition 1.** *A type substitution $\sigma$ is a most general unifier (MGU) of a set of type constraints $C$ if $\sigma \in \mathcal{U}(C)$ and for every $\rho \in \mathcal{U}(C)$ there exists a type substitution $\tau$ such that $\rho = \tau \circ \sigma$.*

**Definition 2.** *An equation $\langle X, T \rangle \in C$ is in* solved form *in a constraints set $C$ if $X$ is a variable that does not occur anywhere else in $C$; in particular $X \notin Var(T)$. A set of constraints $C$ is in solved form if every equation in $C$ is in solved form in $C$.*

Let $C = \{\langle X_1, T_1, \rangle \ldots, \langle X_n, T_n, \rangle\}$ be a set of type constraints in solved form and let $\sigma_C = \{X_1 \mapsto T_1, \ldots, X_n \mapsto T_n\}$. Prove the following statements.

a) $\sigma_C$ is a MGU of $C$.

b) For every unifier $\rho \in \mathcal{U}(C)$ we have $\widehat{\rho} \circ \sigma_C = \rho$.

## 4.2 (18 Points) Let Normal Form

Consider the subset of expressions that is given by the following subset of syntactic forms.

$$\begin{array}{lcll} e & ::= & x & x \\ & & \mathrm{app}(e_1, e_2) & e_1\ e_2 \\ & & \mathrm{let}(e_1, x.e_2) & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ & & \mathrm{true} & \mathsf{true} \\ & & \mathrm{false} & \mathsf{false} \\ & & \mathrm{if}(e, e_1, e_2) & \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

Recall the corresponding rules from the resource-safety-based cost semantics.

$$\frac{V(x)=\ell \qquad q=q'+M^{\mathsf{var}}}{V;H\,{}_M\!\vdash x \Downarrow^{q}_{q'} (\ell,H)}\;(\textsc{Es:Var}) \qquad\qquad \frac{H'=H,\ell\mapsto(\lambda x.e,V) \qquad q=q'+M^{\mathsf{abs}}}{V;H\,{}_M\!\vdash \mathrm{lam}(x.e) \Downarrow^{q}_{q'} (\ell,H')}\;(\textsc{Es:Abs})$$

$$\frac{\begin{array}{c} q=q_0+M^{\mathsf{app}} \qquad V;H\,{}_M\!\vdash e_1 \Downarrow^{q_0}_{q_1} (\ell_1,H_1) \\ H(\ell_1)=(\lambda x.e,V') \qquad V;H_1\,{}_M\!\vdash e_2 \Downarrow^{q_1}_{q_2} (\ell_2,H_2) \qquad V'[x\mapsto\ell_2];H_2\,{}_M\!\vdash e \Downarrow^{q_2}_{q'} (\ell,H') \end{array}}{V;H\,{}_M\!\vdash \mathrm{app}(e_1,e_2) \Downarrow^{q}_{q'} (\ell,H')}\;(\textsc{Es:App})$$

$$\frac{q=q_0+M^{\mathsf{let}} \qquad V,H\,{}_M\!\vdash e_1 \Downarrow^{q_0}_{q_1} (\ell_1,H_1) \qquad V[x\mapsto\ell_1],H_1\,{}_M\!\vdash e_2 \Downarrow^{q_1}_{q'} (\ell,H')}{V;H\,{}_M\!\vdash \mathrm{let}(e_1,x.e_2) \Downarrow^{q}_{q'} (\ell,H')}\;(\textsc{Es:Let})$$

$$\frac{b\in\{\mathrm{true},\mathrm{false}\} \qquad q=q'+M^{\mathsf{cbool}} \qquad H'=H,\ell\mapsto b}{V;H\,{}_M\!\vdash b \Downarrow^{q}_{q'} (H',\ell)}\;(\textsc{Es:Bconst})$$

$$\frac{q=p+M_1^{\mathsf{cond}} \qquad V;H\,{}_M\!\vdash e \Downarrow^{p}_{p'} (\ell,H') \qquad H'(\ell)\text{=true} \qquad V;H'\,{}_M\!\vdash e_1 \Downarrow^{p'}_{q'} (\ell_1,H_1)}{V;H\,{}_M\!\vdash \mathrm{if}(e,e_1,e_2) \Downarrow^{q}_{q'} (\ell_1,H_1)}\;(\textsc{Es:Cond1})$$

$$\frac{q=p+M_2^{\mathsf{cond}} \qquad V;H\,{}_M\!\vdash e \Downarrow^{p}_{p'} (\ell,H') \qquad H'(\ell)\text{=false} \qquad V;H'\,{}_M\!\vdash e_2 \Downarrow^{p'}_{q} (\ell_2,H_2)}{V;H\,{}_M\!\vdash \mathrm{if}(e,e_1,e_2) \Downarrow^{q}_{q'} (\ell_2,H_2)}\;(\textsc{Es:Cond2})$$

When defining more advanced type systems, it is often handy to work with a simpler set of expressions in which the evaluation order is exclusively defined by let expressions. To this end we use variables instead of expressions in syntactic forms whenever this is possible without restricting the expressivity of the language. We define expressions in *let normal form* as follows.

| | | | |
|---|---|---|---|
| $\bar{e}$ | $::=$ | $x$ | $x$ |
| | | $\mathrm{free}(x)$ | $\mathsf{free}(x)$ |
| | | $\mathrm{lam}(x.\bar{e})$ | $\mathsf{fun}\ x \to \bar{e}$ |
| | | $\mathrm{app}(x_1,x_1)$ | $x_1\ x_2$ |
| | | $\mathrm{let}(\bar{e}_1,x.\bar{e}_2)$ | $\mathsf{let}\ x=\bar{e}_1\ \mathsf{in}\ \bar{e}_2$ |
| | | $\mathrm{flet}(\bar{e}_1,x.\bar{e}_2)$ | $\mathsf{freelet}\ x=\bar{e}_1\ \mathsf{in}\ \bar{e}_2$ |
| | | $\mathrm{true}$ | $\mathsf{true}$ |
| | | $\mathrm{false}$ | $\mathsf{false}$ |
| | | $\mathrm{if}(x,\bar{e}_1,\bar{e}_2)$ | $\mathsf{if}\ x\ \mathsf{then}\ \bar{e}_1\ \mathsf{else}\ \bar{e}_2$ |
| | | $\mathrm{pair}(x_1,x_2)$ | $(x_1,x_2)$ |
| | | $\mathrm{matP}(x,(x_1,x_2).\bar{e})$ | $\mathsf{let}\,(x_1,x_2)=x\ \mathsf{in}\ \bar{e}$ |
| | | $\mathrm{nil}$ | $[]$ |
| | | $\mathrm{cons}(x_1,x_2)$ | $x_1 :: x_2$ |
| | | $\mathrm{matL}(x,\bar{e}_1,(x_1,x_2).\bar{e}_2)$ | $\mathsf{match}\ x\ \mathsf{with}\ \mid [] \to \bar{e}_1 \mid x_1 :: x_2 \to \bar{e}_2$ |
| | | $\mathrm{rec}((f,x).\bar{e}_f,f.\bar{e})$ | $\mathsf{let\ rec}\ f\,x=\bar{e}_f\ \mathsf{in}\ \bar{e}$ |

The expressions in let normal form contain a new syntactic form $\mathrm{flet}(\bar{e}_1,x.\bar{e}_2)$ for let expressions. These *free let expressions* are similar to ordinary let expressions except that they do not influence the resource usage. The corresponding evaluation rule is defined as follows.

$$\frac{V, H\,{}_M\!\vdash e_1 \Downarrow_p^q (\ell_1, H_1) \qquad V[x \mapsto \ell_1], H_1\,{}_M\!\vdash e_2 \Downarrow_{q'}^p (\ell, H')}{V; H\,{}_M\!\vdash \mathrm{flet}(e_1, x.e_2) \Downarrow_{q'}^q (\ell, H')} \text{ (Es:FLet)}$$

$$\frac{V(x) = \ell}{V; H\,{}_M\!\vdash \mathrm{free}(x) \Downarrow_q^q (\ell, H)} \text{ (Es:FVar)}$$

Prove the following theorem.

**Theorem 1.** *Let $\Gamma \vdash^m e : T$. Then there exists an expression $\bar{e}$ in let normal form such that $\Gamma \vdash^m \bar{e} : T$ and*

$$V; H\,{}_M\!\vdash e \Downarrow_{q'}^q (\ell, H') \iff V; H\,{}_M\!\vdash \bar{e} \Downarrow_{q'}^q (\ell, H') .$$

*for every resource metric $M$. Moreover, the expression $\bar{e}$ can be efficiently computed from $e$.*

Note that the expression $e$ in the theorem is defined by the subset of syntactic forms defined in this problem. The theorem does not hold if $e$ contains lambda expressions since the translation to let normal form would affect function closures $(\lambda x.e, V)$ that are stored on the heap. So in general, two evaluation judgements would not result in the same heap $H'$. To prove the theorem for expressions with lambda abstraction we would have to define an equivalence relation for heaps that would be preserved during the evaluation of $e$ and $\bar{e}$ under equivalent heaps.