D. MacQueen

Webs

Robin Milner, 1 Sept 85

A way of approaching the static semantics of sharing in modules, using congruences to represent sharing of components.

A skeletal language is given, with its operational semantics.

[Stimulated by Bob Harper's draft for the static semantics of modules.].

There is a debt to Pebble in the relationship between expressions and declarations.

1. The LANGUAGE

Functor identifiers *funid* : A denumerable set of identifiers

Labels *lab* : A denumerable set of identifiers.

Expressions *exp* : [In each case below, $n \geq 0$]

exp ::=

$\{lab_1 \rightarrow exp_1, \ldots, lab_n \rightarrow exp_n\}$ — construction (lab_i distinct)

path — selection

funid exp — application

path ::=

$lab_1 . \ldots . lab_{n+1}$

Specifications *spec* :

spec ::=

tree $[equ_1, \ldots, equ_n]$

tree ::=

$\{lab_1 \rightarrow tree_1, \ldots, lab_n \rightarrow tree_n\}$ (lab_i distinct)

equ ::=

path = path′

Definitions *def* :

def ::=

def funid spec = exp

Declarations *dec* :

dec ::=

dec exp

> A Program is a sequence of definitions and declarations

2. Trees, Congruences and Webs

We use $a, b, \ldots$ to range over Lab, the set of labels, and $p, q, \ldots$ to range over Lab^*. ε is the empty sequence in Lab^*. For $S \subseteq Lab^*$, $Pref(S)$ is the prefix closure of S.

A <u>tree</u> is a non-empty prefix-closed subset of Lab^*. We use T to range over trees. We define:

- $Inits(T) = \{a \mid \exists p.\ ap \in T\}$
- $pT = Pref(p) \cup \{pq \mid q \in T\}$
- $T/p = \{q \mid pq \in T\} \qquad (p \in T)$

Clearly pT and T/p are trees.

A <u>congruence</u> C on T is an equivalence over T such that

(i) If $(p,q) \in C$ and $pa \in T$, then $qa \in T$ and $(pa, qa) \in C$ (Closure under right concatenation)

(ii) If $(p, pq) \in C$ then $q = \varepsilon$ (No cycles).

We define:

- $pC = Id_{Pref(p)} \cup \{(pq_1, pq_2) \mid (q_1, q_2) \in C\}$
- $C/p = \{(q_1, q_2) \mid (pq_1, pq_2) \in C\} \qquad ((p,p) \in C)$

If C is a congruence over T, then clearly pC (resp. C/p) is a congruence over pT (resp. T/p).

A <u>web</u> $W = (T, C)$ is a tree T with a congruence C over T. We define (T_2, C_2) to be a <u>subweb</u> of (T_1, C_1), and write $(T_2, C_2) \leq (T_1, C_1)$, in case $T_2 \subseteq T_1$ and $C_2 \subseteq C_1$. (Fewer paths, less "sharing").

If (T, C) is a wet, then (T', C') is a <u>wet</u> <u>on</u> (T, C) if C' is a congruence on $T \uplus T'$ (the disjoint union of T and T') and moreover $C' \upharpoonright T = C$. More exactly, we introduce two special labels <u>ext</u> and <u>int</u>, and define $T \uplus T' = \underline{\text{ext}}\, T \cup \underline{\text{int}}\, T'$; note that this is a tree (containing ε) and our requirement may be written $C'/\underline{\text{ext}} = C$.

We shall also use

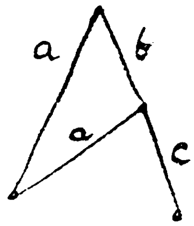
- $C'/(\underline{\text{ext}}, \underline{\text{int}}) = \{(p, p') \mid (\underline{\text{ext}}\, p, \underline{\text{int}}\, p') \in C'\}$
  (a relation between T and T')
- $(\underline{\text{ext}}, \underline{\text{int}})R = \{(\underline{\text{ext}}\, p, \underline{\text{int}}\, p') \mid (p, p') \in R\} \subseteq T \uplus T'$
  (where R is a relation between T and T')

Thus

$$
\begin{aligned}
C' = \; & \underline{\text{ext}}(C'/\underline{\text{ext}}) && (\subseteq \underline{\text{ext}}\, T \times \underline{\text{ext}}\, T) \\
\cup \; & (\underline{\text{ext}}, \underline{\text{int}})(C'/(\underline{\text{ext}}, \underline{\text{int}})) && (\subseteq \underline{\text{ext}}\, T \times \underline{\text{int}}\, T') \\
\cup \; & (\underline{\text{int}}, \underline{\text{ext}})(C'/(\underline{\text{int}}, \underline{\text{ext}})) && (\subseteq \underline{\text{int}}\, T' \times \underline{\text{ext}}\, T) \\
\cup \; & \underline{\text{int}}(C'/\underline{\text{int}}) && (\subseteq \underline{\text{int}}\, T' \times \underline{\text{int}}\, T')
\end{aligned}
$$

The idea is that an expression exp, evaluated on a wet (T, C), yields (T', C') which is a wet on (T, C). Intuitively, the result tree T' has paths which may "share" with each other and with paths of T (the "sharing" being represented by C'), but the evaluation induces no further sharing among paths of T, since $C' \upharpoonright T = C$.

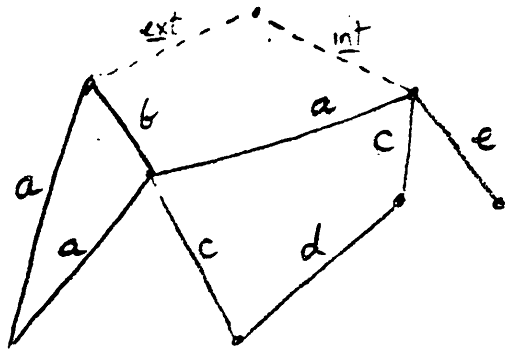
As an example, we may draw a wet (T, C) as a dag with labelled arcs, as follows:

(T, C) 

$T = \{\varepsilon, a, b, ba, bc\}$

$C = Id_T \cup \{(a, ba)\}$

and evaluation may produce (T', C') on (T, C), shown thus:



$T' = \{\varepsilon, a, aa, ac, c, cd, e\}$

$C' = \{(\varepsilon, \varepsilon)\} \cup \underline{ext}\, C$
$\quad \cup\ \underline{int}\,(Id_{T'} \cup \{ac, cd\})$
$\quad \cup\ (\underline{ext}, \underline{int})\{(b, a), (ba, aa), (bc, ac)\}$
$\quad \cup\ (\underline{int}, \underline{ext})\{(a, b), (aa, ba), (ac, bc)\}$

Clearly we can represent congruences more briefly if we define the closure*:

- $Cl\ R$ = the smallest congruence containing R

For example, in the above we have $C = Cl\{(a, ba)\}$, and

$C' = Cl\left(\underline{ext}\, C \cup \underline{int}\{(ac, cd)\} \cup (\underline{ext}, \underline{int})\{(b, a)\}\right)$

We shall call any web of the form $(T \dot\cup T', C')$ a double web; it is clearly a web (T', C') on (T, C) where $C = C'/\underline{ext}$.

---

* Strictly, the closure is relative to a particular tree T. For $R \subseteq T \times T$, one should write $Cl_T\, R$; in our uses T will be clear from context.

3. Composition of double webs

Clearly if $W' = (T', C')$ is a web on W, then $(T', C'/\underline{int})$ is a web, which we denote by $W'/\underline{int}$.

Given a web W' on W, and a web W'' on $W'/\underline{int}$, there is a natural way of composing them to form a web W''' on W.

$$((T'', C'') \text{ on } (T', C'/\underline{int})) \ \underline{\text{with}} \ ((T', C') \text{ on } (T, C))$$
$$\underline{\text{yields}} \ ((T''', C''') \text{ on } (T, C))$$

In fact $T''' = T''$, and C''' (a congruence on $T \cup T''$) is the result of extending C'' and C' to $T \cup T' \cup T''$, taking the closure congruence, and restricting it to $T \cup T''$. We shall denote the resulting $W''' = (T''', C''')$ by

$$\text{Compose}\,(W'', W', W)$$

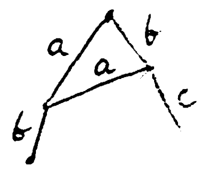
— the formal definition is a little tedious, but not difficult.

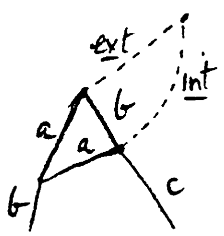
4. Operations on webs and double webs

Each form of expression (construction, selection, application) will be understood semantically as an operation (on webs or double webs) yielding a double web.

(1) Selection. Let $W = (T, C)$ be a web, and $p \in T$. Then Select(p, W), the selection of p from W, is the web (T', C') on W where $T' = T/p$

and $C' = Cl\left(\underline{ext}\, C \cup \underline{int}(C/p) \cup \{(\underline{ext}\, pq, \underline{int}\, q) \mid pq \in T\}\right)$

Example 1 $p = b$, and $(T, C) =$ . Then

$(T \cup T', C') =$ 

It is simple (but necessary) to prove that the selection of p from W is indeed a web on W.

(2) Construction. Let $W = (T, C)$ be a web, let $W_i = (T_i', C_i')$ be webs on (T, C), $1 \le i \le n$, and let a_i $(1 \le i \le n)$ be distinct labels. Write $\tilde{a} = (a_1, \ldots, a_n)$ and $\tilde{W} = (W_1, \ldots, W_n)$. Then Construct$(\tilde{W}, \tilde{a}, W)$, the construction of $\tilde{W}$ by $\tilde{a}$ from W, is the web (T', C') on W,

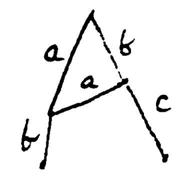
where $T' = a_1 T_1' \cup \ldots \cup a_n T_n'$

and

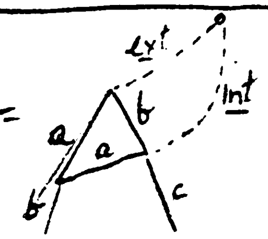
$$C' = Cl\left(\underline{ext}\,C \cup \underline{int}\left(\bigcup_i a_i(C_i'/\underline{int})\right) \cup \{(\underline{ext}\,p, \underline{int}\,a_i q) \mid (\underline{ext}\,p, \underline{int}\,q) \in C_i,\ 1 \le i \le n\}\right)$$

Again, it is necessary to prove that a construction from W is indeed a web on W.

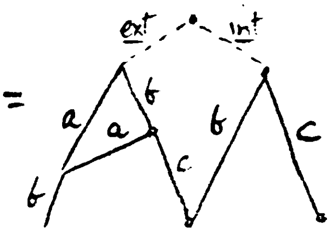
<u>Example 2</u> $W = (T, C) =$



$(T \cup T_1', C_1') =$

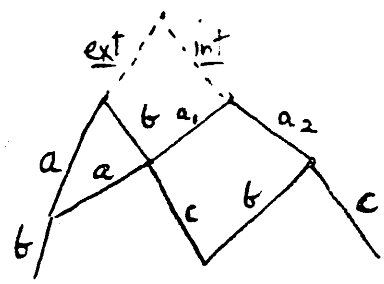


$(T \cup T_2', C_2') =$



Then $\text{Construct}((W_1, W_2), (a_1, a_2), W) = (T', C')$, where

$(T \cup T', C') =$



Before considering the meaning of application, it may help to see how the two examples may be declared in our language.

$\underline{dec}\ a = \{b \to \{\,\}\}$

$\underline{dec}\ b = \{a \to a, c \to \{\,\}\}$ — A selection, referring to the "a" previously declared

Now the "environment" is the web $W = (T, C)$ of Example 1. Then, the expressions

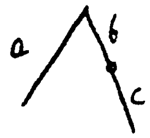
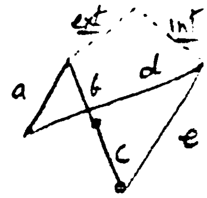
$$b \qquad \text{and} \qquad \{b \to b.c,\ c \to \{\,\}\}$$

evaluate to the webs W_1 and W_2 on W, of Example 2, while the expression

$$\{a_1 \to b,\ a_2 \to \{b \to b.c,\ c \to \{\,\}\}\}$$

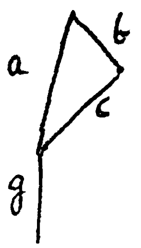
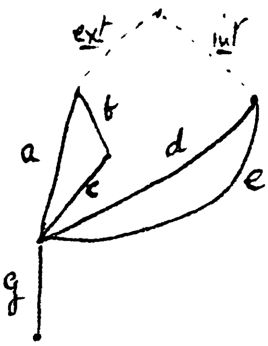
evaluates to the web (T', C') on W of Example 2.

(3) <u>Application</u> The meaning of a functor identifier F defined to be a double web, namely W_F'' on W_F', where W_F' is the web denoted by its specification and W_F'' the result of evaluating its expression on W_F'. Consider an example:

W_F'  W_F'' on W_F' 

Using this example, we show what the result should be of applying this double web to an argument web $W' \geq W_F'$:

<u>Argument</u> <u>Result</u>

W'  W'' on W' 

We can see that, because W' has more paths and more sharing than W_F', so W'' has more paths and more sharing than W_F''.

<u>Proposition</u> Given a web W_F'' on W_F' and a web $W' \geq W_F'$, there is a least web W'' on W' such that $W'' \geq W_F''$.

We define Extend(W_F'', W_F', W') to be this web W'' on W'.

Now in general we want, roughly speaking, to apply the (meaning of the) functor identifier F to a web W' on W (i.e. the result of evaluating the argument expression on W). The full result should be a web W'' on W.

Finally, therefore, we define the application of a web W_F'' on W_F' to a web W' on W, provided that $W_F' \leq W'/\underline{\text{int}}$, as follows:

$$Apply(W_F'', W_F', W', W) =$$
$$Compose\left(Extend(W_F'', W_F', W'/\underline{\text{int}}), W', W\right)$$

(4) <u>Absorption</u>. The final operation is to do with declarations, not expressions. The declared expression yields a web $W' = (T', C')$ on the "environment" web $W = (T, C)$, and we wish to produce a new "environment" $W_{new} = (T_{new}, C_{new})$ in which T' overrides T (in a precise sense) to form T_{new}.

If $A \subseteq Lab$, define in general

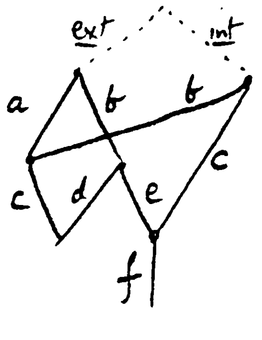
- $T \upharpoonright A = \{\varepsilon\} \cup \{ap \in T \mid a \in A\}$
- $C \upharpoonright A = \{(\varepsilon, \varepsilon)\} \cup \{(ap, bq) \in C \mid a, b \in A\}$

Then we define

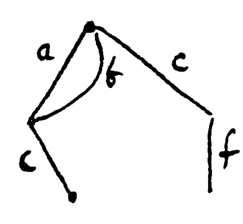
$$Absorb((T, C), (T', C')) = (T_{new}, C_{new}) \text{ where}$$
$$T_{new} = T \upharpoonright (Lab - Inits(T')) \cup T'$$
$$C_{new} = Cl\Big(C \upharpoonright A \cup C'/\underline{\text{int}}$$
$$\cup \{(ap, q) \in C'/(\underline{\text{ext}}, \underline{\text{int}}) \mid a \notin Inits(T')\}\Big)$$

<u>Example</u>

(T', C') on (T, C):



$Absorb((T,C),(T',C'))$:



5. Semantics of the Language

We imagine that a "program" in our language is a sequence of definitions mixed with declarations. Each declaration modifies the environment web, and thus affects the meaning of subsequent declarations, since they can refer to it using selection. However the environment web has no effect upon the meaning of a definition, since all selections within a definition are taken to refer to the (virtual) environment web denoted by the definition's specification.

The set of functor environments is

$$(\mathcal{F} \in)\ \text{Funs} = \text{Funid} \xrightarrow{\text{FIN}} \text{Webs}$$

The semantics is an inference system over sentences of the following forms:

$$\mathcal{F}, W \vdash \text{exp} \Longrightarrow W' \qquad (W' \text{ a web over } W)$$

$$\vdash \text{spec} \Longrightarrow W$$

$$\vdash \text{tree} \Longrightarrow T$$

$$\mathcal{F} \vdash \text{def} \Longrightarrow \mathcal{F}_{new}$$

$$\mathcal{F}, W \vdash \text{dec} \Longrightarrow W_{new}$$

$$\boxed{\mathcal{F}, W \vdash exp \Rightarrow W'}$$

Construction:

$$\frac{\mathcal{F}, W \vdash exp_i \Rightarrow W_i \quad (1 \leq i \leq n)}{\mathcal{F}, W \vdash \{lab_1 \to exp_1, \ldots, lab_n \to exp_n\} \Rightarrow Construct((W_1, \ldots, W_n), (lab_1, \ldots, lab_n), W)} \quad (lab_i \text{ distinct})$$

Selection:

$$\frac{}{\mathcal{F}, W \vdash path \Rightarrow Select(path, W)}$$

$$(W = (T, C),\ path \in T)$$

Application:

$$\frac{\mathcal{F}, W \vdash exp \Rightarrow W'}{\mathcal{F}, W \vdash funid\ exp \Rightarrow Apply(W''_{funid}, W'_{funid}, W', W)}$$

$$(\mathcal{F}(funid) = (W''_{funid}, W'_{funid});\ \ W'_{funid} \leq W/\underline{int})$$

$$\boxed{\vdash tree \Rightarrow T}$$

$$\frac{\vdash tree_i \Rightarrow T_i \quad (1 \leq i \leq n)}{\vdash \{lab_1 \to tree_1, \ldots, lab_n \to tree_n\} \Rightarrow lab_1 T_1 \cup \cdots \cup lab_n T_n} \quad (lab_i \text{ distinct})$$

$$\boxed{\vdash spec \Rightarrow W}$$

$$\frac{\vdash tree \Rightarrow T}{\vdash tree\,[p_1 = p_1', \ldots, p_n = p_n'] \Rightarrow (T, C)} \quad (p_i, p_i' \in T)$$

$$(\text{provided } C = Cl\{(p_1, p_1'), \ldots, (p_n, p_n')\} \text{ exists}^{*})$$

---

* It may fail to exist, due to either of the conditions (i), (ii) on a congruence,

$$\boxed{\mathcal{F} \vdash def \Rightarrow \mathcal{F}_{new}}$$

$$\frac{\vdash spec \Rightarrow W \qquad \mathcal{F}, W \vdash exp \Rightarrow W'}{\mathcal{F} \vdash \underline{def}\ funid\ spec = exp \ \Rightarrow \mathcal{F}[funid \mapsto (W', W)]}$$

$$\boxed{\mathcal{F}, W \vdash dec \Rightarrow W_{new}}$$

$$\frac{\mathcal{F}, W \vdash exp \Rightarrow W'}{\mathcal{F}, W \vdash \underline{dec}\ exp \Rightarrow Absorb(W, W')}$$

6 Remarks

The correspondence between this language and MacQueen modules has not been detailed. But notice that his signatures correspond to our specifications. I find it tidy to allow every "signature" to have a sharing clause, so that the formal parameter to a functor is just a "signature". Also, signature identifiers don't exist here — but they can easily be introduced, and then evaluating a specification from component specifications would entail a little bit of congruence closure.

The main point of this whole exercise is to give a skeletal framework within which questions about sharing — and signature matching — can be clearly studied; also, I wanted to be sure that Bob Harper's Timestamps are not essential in the static semantics of modules. (The question arises whether the machinery of congruences, though mathematically precise, is a reasonable price to pay for dispensing with Timestamps).

One pertinent question to study is the effect of adding explicit "signature" constraints; in particular what is the meaning of

def funid spec = exp : spec′

where the body of the definition must now match spec′?