A Grainless Semantics for HARPO/L

Theodore S. Norvell
Computer Engineering Research Labs
Memorial University

December 21, 2010

DRAFT. Typeset December 21, 2010

Abstract

A grainless semantics follows the following three principles quoted from John Reynolds [[citation]]:

— Operations have duration and can overlap each other during execution.

— If two overlapping operations touch the same location, the meaning of the program execution is "wrong".

— If, from a given starting state, execution of a program can give "wrong", then no other possibilities need be considered.

In HARPO/L, we modify the second principle slightly by allowing parallel operations to simultaneously read locations.

Following the work of Reynolds [[citation]] we provide a semantics in three steps. The first is a denotational semantics of the language's expressions. This semantics gives not only the possible values of each expression in each state, but also gives the portion of the state that needs to be read to compute the value. The second gives a denotational semantics of commands by mapping each command into a set of sequences of actions. Each action is considered to take no time. Operations that would normally take a period of time, such as reading and writing memory, are split into two actions: one indicates the start of the operation and one indicates the end of the operation. For the duration of such an operation, memory locations are either read-locked or write-locked. The third step is to give an operational semantics for traces.

0 List of symbols

Names $n, b, c \in N$. And sans serif is used for name constants in examples.

Locations $l \in L$
Sets and bags of locations $R, W \in \mathcal{P}(L)$ or $R \in \mathcal{B}(L)$
Objects: $o, k \in O$, $K \subseteq O$
Arrays $a \in A$
Methods $m \in M$
Numbers $i, j \in \mathbb{N}$
Boolean values: $ok \in \mathbb{B}$
Values $v \in V$
States $\sigma \in \Sigma$
Expressions E
Commands C
Actions $p, q, r \in P$
Traces $s, t, u \in P^\infty$
Trace sets $S, T, U \subseteq P^\infty$
Configurations $x, y, z \in Z$, $X, Y \subseteq Z$

1 Context

We assume that the source program has already gone through static checking, generic specialization, and object instantiation phases. In other words that the program has been compiled to an object graph, an initial state, and a single command. The context for a HARPO/L command consists of a static part, which is the object graph, and a dynamic part which is the state.

1.0 Object Graph

For each program, the object graph is described by 5 sets and a number of functions. The sets are:

- N, a set of identifiers.
- L, a set of locations.
- O a set of objects. Note that this set is fixed as there is no allocation or deallocation.
- A is a set of arrays. Each array has a fixed length length(a).
- M is a set of methods.

We will also use a set V, which is the set of primitive values for a particular implementation.

The special value $\bigstar$ indicates an expression computation "gone wrong" and we'll assume that it is outside any of the above sets.

Each object has a fixed set of field names $\text{fields}(o) \subseteq_{\text{fin}} N$ and for $n \in \text{fields}(o)$, $\text{field}(o, n) \in O \cup L \cup A$. We'll extend the field function to wrong values by defining $\text{field}(\bigstar, n) = \bigstar$, for all $n \in N$. We assume that all local variables have been replaced by object fields, so local variables are also accessed as fields.

Each object has a fixed set of method names $\text{methods}(o) \subseteq_{\text{fin}} N$ and for $n \in \text{methods}(o)$, $\text{method}(o, n) \in M$. We have $\text{method}(\bigstar, n) = \bigstar$, for all $n \in N$.

Similarly, for each $a \in A$, and $v \in \{0, .. \text{length}(a)\}$, $\text{index}(a, v) \in O \cup L \cup A$. (We assume $\{0, .. \text{length}(a)\} \subseteq V$, for all arrays a.) For all other $v \in V$, define $\text{index}(a, v) = \bigstar$. We'll also define $\text{index}(\bigstar, v) = \text{index}(a, \bigstar) = \bigstar$.

The set of global entities is represented by a finite partial function $\text{global} \in N \rightsquigarrow O \cup L \cup A$.

Because of the static nature of HARPO/L, the global, fields, field, methods, method, length, and index functions do not depend on the state.

1.1 States

As mentioned above V is a set of primitive values.

A state is a partial function $\sigma \in \Sigma = (L \rightsquigarrow V)$. States may be considered as sets of pairs, as the source (L) and the target (V) are always the same. Thus I'll deliberately ignore any distinction between a state and its graph (i.e. its set of pairs).

The domain of σ is written $\delta(\sigma)$.

Two partial functions are compatible if they agree on the intersection of their domains:

$$\sigma_0 \smile \sigma_1 = \forall l \in \delta(\sigma_0) \cap \delta(\sigma_1) \cdot \sigma_0(l) = \sigma_1(l) \quad .$$

If $\sigma_0 \smile \sigma_1$, then, using the pun between states and their graphs, $\sigma_0 \cup \sigma_1$ is a state too.

If l is a location and v is a value, then $l \mapsto v$ is the state such that $\delta(l \mapsto v) = \{l\}$ and $(l \mapsto v)(l) = v$. Using the pun between states and their graphs we have $(l \mapsto v) = \{(l, v)\}$.

If σ_0 and σ_1 are states, then $\sigma_0 \triangleright \sigma_1$ is a state such that

$$\begin{array}{rcll} \delta(\sigma_0 \triangleright \sigma_1) & = & \delta(\sigma_0) \cup \delta(\sigma_1) & \\ (\sigma_0 \triangleright \sigma_1)(l) & = & \sigma_0(l) & \text{if } l \in \delta(\sigma_0) \\ (\sigma_0 \triangleright \sigma_1)(l) & = & \sigma_1(l) & \text{if } l \notin \delta(\sigma_0) \text{ and } l \in \delta(\sigma_1) \quad . \end{array}$$

Note that

$$\sigma_0 \triangleright \sigma_1 = \sigma_1 \triangleright \sigma_0 = \sigma_0 \cup \sigma_1, \text{ when } \sigma_0 \smile \sigma_1 \quad .$$

I'll generally use $\sigma_0 \cup \sigma_1$ in this case to emphasize the symmetry of the operation and the compatibility of the states.

2 Expressions

The semantics of each expression E is a set of pairs $[E]_\$$ each in $\Sigma \times (V \cup \{\bigstar\})$. For example:

- Literals

  $$[E]_\$ = \{(\emptyset, v)\}\,, \text{ where } E \text{ is a literal and } v \text{ is the corresponding value.}$$

- Addition[0]

  $$[E_0 + E_1]_\$ = \left\{(\sigma_0, v_0) \in [E_0]_\$\,, (\sigma_1, v_1) \in [E_1]_\$ \mid \sigma_0 \smile \sigma_1 \cdot \left(\sigma_0 \cup \sigma_1, v_0 \hat{+} v_1\right)\right\}$$

  where , $v_0 \hat{+} v_1 = v_0 + v_1$ when $v_{0,} v_1 \in V$ and $v_0 \hat{+} v_1 = \bigstar$ otherwise. The meaning of $+$ here is actually determined by the rules based on the static types of E_0 and E_1.

- Other expressions are similar

- Tuples. Although tuples are not part of the language, we will need this when we treat the matter of guards in rendezvous:

  $$[(E_0, E_1)]_\$ = \left\{(\sigma_0, v_0) \in [E_0]_\$\,, (\sigma_1, v_1) \in [E_1]_\$ \mid \sigma_0 \smile \sigma_1 \cdot (\sigma_0 \cup \sigma_1, (v_0, v_1))\right\}$$

2.0 LValue expressions and fetches

An lvalue expression is one that refers to a location, array, or object.

We assume that all lvalue expressions have been rewritten during the initialization phase to fully qualified form. I.e., each lvalue expression is of the form

$$n q_0 q_1 ... q_m \quad ,$$

where n is the name of a global and each qualifier q_i is either of the form $.n$ where n is a field name or $[E]$ where E is an expression of integer type. Recall that we are assuming that local variables have been converted to fields and so they are treated the same as fields.

The semantics of each lvalue expression E is a set of pairs $[E]_@$ each in $\Sigma \times (O \cup L \cup A \cup \{\bigstar\})$ as follows:

[0]The set $\{v \in S \mid P \cdot E\}$ contains exactly those things that equal E^v_a for some value $a \in S$ of variable(s) v satisfying boolean expression P^v_a. For example $\left\{n \in \mathbb{N} \mid n \text{ is prime} \cdot n^2\right\}$ is the set of squares of prime natural numbers: $\{4, 9, 25, 49, \cdots\}$.

If the "$\mid P$" is omitted, it defaults to "$\mid$ *true*" .

The generalization to more than one variable is obvious. E.g. $\{m \in M, n \in \mathbb{N} \mid m \text{ is prime} \wedge n \text{ is prime} \cdot m \times n\}$ is $\{4, 6, 9.10, 14, \cdots\}$.

- A name by itself must refer to a global. Static checks and the nature of fully qualified names will ensure $n \in \delta(\text{global})$.

$$[n]_{@} = \{(\emptyset, \text{global}(n))\}$$

- Field lookup.

$$[E.n]_{@} = \{(\sigma, o) \in [E]_{@} \cdot (\sigma, \text{field}(o, n))\}$$

  Static checks will ensure that $o \in O \cup \{\bigstar\}$ and that $n \in \text{fields}(o)$ if $o \in O$. Recall that if $o = \bigstar$, then $\text{field}(o, n) = \bigstar$ too.

- Array lookup.

$$[E[F]]_{@} = \{(\sigma_0, a) \in [E]_{@}, (\sigma_1, i) \in [F]_{\$} \mid \sigma_0 \smile \sigma_1 \cdot (\sigma_0 \cup \sigma_1, \text{index}(a, i))\}$$

  Static checks will ensure that $a \in A \cup \{\bigstar\}$ and that i is at least of a reasonable type, if it is not $\bigstar$. Static checks do not ensure that i is in range, but the way we defined the index function, is such that $\text{index}(a, i) = \bigstar$ if i is not in range. Recall that if $a = \bigstar$ or $i = \bigstar$, then $\text{index}(a, i) = \bigstar$ too.

Now we can define the semantics of fetches.

- Where E is an lvalue expression, let V_E be the subset of values of E's type, then define

$$\begin{aligned} [E]_{\$} \;&=\; \{(\sigma, l) \in [E]_{@}, v \in V_E \mid l \in L \wedge \sigma \smile (l \mapsto v) \cdot (\sigma \cup (l \mapsto v), v)\} \\ &\quad \cup \{(\sigma, \bigstar) \in [E]_{@} \cdot (\sigma, \bigstar)\} \quad . \end{aligned}$$

  The static checks will have ensured that, for any lvalue expression E for which $[E]_{\$}$ is relevant, $[E]_{@} \subseteq \Sigma \times (L \cup \{\bigstar\})$, so the two cases covered here suffice. I.e. there is no need to define $[E]_{\$}$ where $[E]_{@}$ may result in an array or an object, as no valid program will need to fetch an array or an object.

2.1 Discussion

Traditional denotational semantics uses total states (i.e. states that are total functions). By using partial functions, we can capture additional information in the semantics of states. This is the set of locations that needs to be read in order to compute the value of the expression. This set is the footprint of the expression. If $(\sigma, v) \in [E]_{\$}$, then $\delta(\sigma)$ is a footprint that can be used to compute the value v for the expression E.

Here are some examples. Consider a program with three global int variables called i, j and k and one global array called a. We'll assume

$$\begin{array}{rcl}
\text{global}(\mathsf{a}) & = & a0 \in A \\
\text{length}(a) & = & 3 \\
\text{index}(a, 0) & = & l0 \in L \\
\text{index}(a, 1) & = & l1 \in L \\
\text{index}(a, 2) & = & l2 \in L \\
\text{global}(\mathsf{i}) & = & l3 \in L \\
\text{global}(\mathsf{j}) & = & l4 \in L \\
\text{global}(\mathsf{k}) & = & l5 \in L \quad .
\end{array}$$

The lvalue semantics $[\mathsf{i}]_@$ of the lvalue expression i equals

$$\{(\emptyset, l3)\} \quad ;$$

the domain of state $\emptyset$ is set $\emptyset$ corresponding to the fact that no fetches are done to compute the location.

The expression semantics $[\mathsf{i}]_\$$ of the expression i includes the following

$$(l3 \mapsto 0, 0)\,, (l3 \mapsto 42, 42)\,, \text{etc}$$

but not

$$(l3 \mapsto 42 \cup l0 \mapsto 0, 42) \quad ,$$

for example, as the value of location $l0$ can not be accessed in the computation. The expression semantics $[\mathsf{i}]_\$$ also does not include

$$(l3 \mapsto 0, 1)$$

as, in a state where i is 0, the fetch will not result in 1.

The lvalue semantics $[\mathsf{a}[\mathsf{i}]]_@$ of the lvalue expression $\mathsf{a}[\mathsf{i}]$ includes

$$(l3 \mapsto 0, l0),\ (l3 \mapsto 1, l1), \text{ and } (l3 \mapsto -1, \bigstar) \quad .$$

The expression semantics $[\mathsf{a}[\mathsf{i}]]_@$ of $\mathsf{a}[\mathsf{i}]$ includes

$$(l0 \mapsto 23 \cup l3 \mapsto 0, 23),\ (l1 \mapsto 42 \cup l3 \mapsto 1, 42)\,, \text{ and } (l3 \mapsto -1, \bigstar) \quad .$$

The examples give deterministic semantics in the sense that $[E]_\$$ is the graph of a partial function in $\Sigma \rightsquigarrow (V \cup \{\bigstar\})$. There is no requirement that expression evaluation be deterministic. For example we could define an choice expression

$$E_0 \Box E_1$$

such that

$$[E_0 \Box E_1]_\$ = \left\{(\sigma_0, v_0) \in [E_0]_\$, (\sigma_1, v_1) \in [E_1]_\$, v \in \{v_0, v_1\} \mid \sigma_0 \smile \sigma_1 \cdot (\sigma_0 \cup \sigma_1, v)\right\}$$

(The choice is made between values, rather than which expression to evaluate.) Nondeterminism could be quite useful in dealing with floating point numbers.

That expressions do not have side effects, on the other hand, is built in to the semantic formalism.

3 Commands

3.0 Traces

Given an alphabet of atoms P, P^* is the set of all finite traces over P, i.e.

$$P^* = \bigcup_{n \in \mathbb{N}} (\{0, ..n\} \to P) \quad ,$$

while P^ω is the set of infinite traces, i.e. $\mathbb{N} \to P$. Let $P^\infty = P^* \cup P^\omega$. Finite traces may be written as $\langle p, q, r \rangle$.

The length of a trace $\#s$ is $|\delta(s)|$. So that

$$(\#s = \aleph_0) = (s \in P^\omega) \text{ and } (\#s \in \mathbb{N}) = (s \in P^*) \quad ,$$

for any $s \in P^\infty$.

The catenation $s;t$ of traces s and t in P^∞ is

$$\begin{aligned} \#(s;t) &= \aleph_0 \text{ if } s \in P^\omega \text{ or } t \in P^\omega \quad , \\ \#(s;t) &= \#s + \#t \text{ otherwise} \quad , \\ (s;t)\,(i) &= s(i), \text{ for all } i \in \mathbb{N} \text{ such that } i < |s| \quad \text{, and} \\ (s;t)(i) &= t(i - \#s), \text{ for all } i \in \mathbb{N} \text{ such that } i \geq \#s \quad . \end{aligned}$$

In particular, $s;t = s$ if $s \in P^\omega$.

A trace set S is any subset of P^∞.

We can define, for trace sets S and T and $i \in \mathbb{N}$,

$$\begin{aligned} S;T &= \{s, t \mid s \in S, t \in T \cdot s;t\} \\ S^i &= \underbrace{S; S; ...; S}_{i \text{ times}} \\ S^* &= \bigcup_{i \in \mathbb{N}} S^i \\ S^\omega &= S; S; ... \\ S^\infty &= S^* \cup S^\omega \quad . \end{aligned}$$

For two traces s and t, we define the fair interleaving $s \shortparallel t$ of the traces as the trace set

$$\{s_0, s_1, \cdots, t_0, t_1, \ldots \in P^* \mid s = s_0; s_1; \ldots \wedge t = t_0; t_1; \ldots \cdot s_0; t_0; s_1; t_1; \cdots\} \quad ,$$

where any of the pieces could be empty, but all are finite. As this definition involves an unbounded (and possibly infinite) number of variables, we make it more precise. For given s and t, suppose $f : \delta(s;t) \to \{0,1\}$ and f doesn't get infinitely stuck at one value: $\forall i \in \mathbb{N} \cdot \neg\forall j \in \mathbb{N} \cdot i + j \in \delta(f) \wedge f(i+j) \neq f(i)$. Define

$$\begin{array}{rcl} \mathrm{merge}_f(s,t) & : & \delta(s;t) \to P \\ \mathrm{merge}_f(s,t)(i) & = & \left\{ \begin{array}{ll} s(i - \sum_{j \in \{0,..i\}} f(j)) & \text{if } f(i) = 0 \\ t(\sum_{j \in \{0,..i\}} f(j)) & \text{if } f(i) = 1 \end{array} \right. \end{array}$$

The fair interleaving of two traces is

$$s \shortparallel t = \big\{f \mid f \text{ is as above} \cdot \mathrm{merge}_f(s,t)\big\}$$

The fair interleaving of two trace sets S and T is

$$S \shortparallel T = \bigcup_{s \in S, t \in T} s \shortparallel t$$

As usual in formal language theory, I will pun between an atom p, trace $\langle p \rangle$, and trace set $\{\langle p \rangle\}$, when there is no ambiguity; for example $p; q = \langle p \rangle ; \langle q \rangle = \langle p, q \rangle$ and $p^* = \{\langle p \rangle\}^*$.

3.1 Command semantics

For our purposes the set of atoms P is a set of actions, which will be defined below, bit by bit, as needed. The semantic function for commands maps each command C to a trace set

$$[C]_! \subseteq P^\infty \quad .$$

3.2 Assignment

$$\begin{array}{rcl} [E_0 := E_1]_! & = & \left\{ \begin{array}{l} (\sigma_0, l) \in [E_0]_@, (\sigma_1, v) \in [E_1]_\$ \mid \sigma_0 \smile \sigma_1 \wedge l \neq \bigstar \wedge v \neq \bigstar \cdot \\ \mathbf{start}(\sigma_0 \cup \sigma_1, \{l\}); \mathbf{fin}(l \mapsto v, \delta(\sigma_0) \cup \delta(\sigma_1)) \end{array} \right\} \\ & \cup & \left\{ \begin{array}{l} (\sigma_0, l) \in [E_0]_@, (\sigma_1, v) \in [E_1]_\$ \mid \sigma_0 \smile \sigma_1 \wedge (l = \bigstar \vee v = \bigstar) \cdot \\ \mathbf{chaos}(\sigma_0 \cup \sigma_1) \end{array} \right\} \end{array}$$

3.2.0 Example

For example, $[\mathsf{i} := \mathsf{j}]_!$ includes

$$\mathbf{start}(l4 \mapsto 99, \{l3\}); \mathbf{fin}(l3 \mapsto 99, \{l4\}) \quad .$$

While $[\mathsf{i} := \mathsf{i} + \mathsf{j}]_!$ includes

$$\mathbf{start}(l3 \mapsto 34 \cup l4 \mapsto 99, \{l3\}); \mathbf{fin}(l3 \mapsto 133, \{l3, l4\}) \quad .$$

3.3 Configurations

We understand the meaning of commands in two steps. The first step is the semantic function $[]_!$ which maps each command to a set of traces. The second step explains each trace by its effect on an abstract machine.

Before going any further, we'll try to understand the meaning of the trace sets generated by command assignment commands.

A configuration is a 5-tuple $(\sigma, R, W, K, ok) \in Z$ consisting of

- σ, a total state, i.e. a total function from L to V,
- R, a bag (multiset) of locations that may currently be being read,
- W, a set of locations that may currently be being written,
- K, a set of locks that are locked, and
- ok, a boolean indicating that the computation has not gone wrong.

3.4 Actions

A relation $x \overset{p}{\rightarrowtail} y$ on configurations shows how the machine can evolve in one step under the influence of an action p.

For a given action p and configuration x, there are three possibilities:

- $x \overset{p}{\rightarrowtail} y$ for no y. In this case we have met a dead end. This happens when the action a is not applicable to x. For example the action $\mathbf{start}(l4 \mapsto 99, \{l3\})$ that arises from the assignment $\mathsf{i} := \mathsf{j}$, is not applicable in a state where j is anything but 99.

- $x \overset{p}{\rightarrowtail} y$ for all y. In this case the computation is undefined from this point on. Anything could happen. This is what the **chaos** action is for. It indicates a computation gone wrong. Computations also blow up if they need to read a location that is being written, need to write a location that is being read, or need to write a location that is being written. When x is not ok, this will always be the case.

- $x \overset{p}{\rightarrowtail} y$ for one or more y, but not for all. Typically only one y will do. In this case the computation proceeds to the next configuration. As mentioned above, this case only arises when x is ok; by convention y will also be ok.

$$\begin{array}{lcl}
(\sigma_0, R_0, W_0, K_0, \text{true}) & \overset{\mathbf{start}(\sigma,W)}{\rightarrowtail} & \begin{cases} (\sigma_0, \delta(\sigma) \uplus R_0, W_0 \cup W, K_0, \text{true}) \\ \qquad \text{if } \sigma \smile \sigma_0 \text{ and } \delta(\sigma) \cap W_0 = \emptyset \text{ and } W \cap (W_0 \cup R_0) = \emptyset \\ (\sigma_1, R_1, W_1, K_1, ok_1) \\ \qquad \text{if } \sigma \smile \sigma_0 \text{ and (} \delta(\sigma) \cap W_0 \neq \emptyset \text{ or } W \cap (W_0 \cup R_0) \neq \emptyset \text{)} \end{cases} \\
(\sigma_0, R_0, W_0, K_0, \text{true}) & \overset{\mathbf{fin}(\sigma,R)}{\rightarrowtail} & (\sigma \rhd \sigma_0, R_0 - R, K_0, W_0 - \delta(\sigma), \text{true}) \\
(\sigma_0, R_0, W_0, K_0, \text{true}) & \overset{\mathbf{chaos}(\sigma)}{\rightarrowtail} & (\sigma_1, R_1, W_1, K_1, ok_1) \qquad \sigma \smile \sigma_0 \\
(\sigma_0, R_0, W_0, K_0, \text{false}) & \overset{p}{\rightarrowtail} & (\sigma_1, R_1, W_1, K_1, ok_1)
\end{array}$$

The additive union $\delta(\sigma) \uplus R_0$ yields a bag which contains every element of the set $\delta(\sigma)$ one more time than does the bag R_0, and contains everything else the same number of times as does R_0. Conversely the bag subtraction $R_0 - R$ yields a bag that contains each element of set R one time less than does R_0 (down to a minimum of zero times) and contains everything else the same number of times as does R_0.

The last rule says that a computation that is not ok can lead to any configuration at all. This ensures that once a computation has gone wrong it can not be recovered from, unequivocally.

3.4.0 Examples

Let's look at some examples. In these examples, I'll only show the relevant parts of the state.

As we saw earlier $[\mathsf{i} := \mathsf{j}]_!$ includes

$$\langle \mathbf{start}(l4 \mapsto 99, \{l3\}), \mathbf{fin}(\{(l3, 99)\}, \{l4\}) \rangle \quad .$$

In a state $l3 \mapsto 10 \cup l4 \mapsto 99$ with no initial read or write locks what happens?

$$\begin{array}{rl}
& ((l3 \mapsto 10 \cup l4 \mapsto 99), \emptyset, \emptyset, \emptyset, \text{true}) \\
\overset{\mathbf{start}(l4 \mapsto 99, \{l3\})}{\rightarrowtail} & ((l3 \mapsto 10 \cup l4 \mapsto 99), \{l4\}, \{l3\}, \emptyset, \text{true}) \\
\overset{\mathbf{fin}(l3 \mapsto 99, \{l4\})}{\rightarrowtail} & ((l3 \mapsto 99 \cup l4 \mapsto 99), \emptyset, \emptyset, \emptyset, \text{true}) \quad .
\end{array}$$

In the same state, $l3 \mapsto 10 \cup l4 \mapsto 99$, but a different trace from $[\mathsf{i} := \mathsf{j}]_!$

$$\langle \mathbf{start}(l4 \mapsto 23, \{l3\}), \mathbf{fin}(l3 \mapsto 23, \{l4\}) \rangle \quad ,$$

no transition is possible. Finally, consider if there is initially a write lock on $l3$:

$$((l3 \mapsto 10 \cup l4 \mapsto 99)\,, \emptyset, \{l3\}\,, \emptyset, \mathrm{true}) \overset{\mathbf{start}(l4 \mapsto 99, \{l3\})}{\rightarrowtail} y \qquad ,$$

for all configurations y. In a sense the computation blows up.

3.5 Control constructs

Sequential composition means catenate the possible traces

$$[C_0\ C_1]_! = [C_0]_!\,; [C_1]_! \quad .$$

For boolean expression E, define

$$\begin{array}{rcl} \mathrm{filter}(E) & \subseteq & P^\infty \\ \mathrm{filter}(E) & = & \left\{\sigma \mid (\sigma, \mathsf{true}) \in [E]_\$ \cdot \langle \mathbf{start}(\sigma, \emptyset), \mathbf{fin}(\emptyset, \delta(\sigma))\rangle\right\} \\ & \cup & \left\{\sigma \mid (\sigma, \bigstar) \in [E]_\$ \cdot \langle \mathbf{chaos}(\sigma)\rangle\right\} \quad . \end{array}$$

For a boolean expression E, static checks and the semantics of expressions will ensure that $[E]_\$ \subseteq \Sigma \times \{\mathsf{false}, \mathsf{true}, \bigstar\}$ and that, for any σ,

$$\text{if } (\sigma, \mathsf{true}) \in [E]_\$ \text{ then } (\sigma, \mathsf{false}) \in [\neg E]_\$$$

and

$$\text{if } (\sigma, \mathsf{false}) \in [E]_\$ \text{ then } (\sigma, \mathsf{true}) \in [\neg E]_\$ \quad .$$

Now

$$\begin{array}{rcl} [(\mathbf{if}\ E\ C_0\ \mathbf{else}\ C_1\ \mathbf{if})]_! = & & \mathrm{filter}(E); [C_0]_! \\ & \cup & \mathrm{filter}(\neg E); [C_1]_! \quad . \end{array}$$

While loops

$$[(\mathbf{wh}\ E\ C\ \mathbf{wh})]_! = ((\mathrm{filter}(E); [C]_!)^*\,; \mathrm{filter}(\neg E)) \cup (\mathrm{filter}(E); [C]_!)^\omega \quad .$$

Parallelism

$$[(\mathbf{co}\ C_0 \shortparallel C_1\ \mathbf{co})]_! = [C_0]_! \shortparallel [C_1]_! \quad .$$

3.6 Mutual exclusion

To deal with statements, we need some new actions

$$\begin{array}{rcl} [(\mathbf{with}\ E_0\ C\ \mathbf{with})]_! = & & \displaystyle\bigcup_{(\sigma,k)\in[E_0]_@ \mid k \neq \bigstar} \mathbf{start}(\sigma, \emptyset); \mathbf{fin}(\emptyset, \delta(\sigma)); \mathrm{enter}(k); [C]_!\,; \mathbf{rel}(k) \\ & \cup & \{(\sigma, k) \in [E_0]_@ \mid k = \bigstar \cdot \langle \mathbf{chaos}(\sigma)\rangle\} \end{array}$$

where

$$\text{enter}(k) = \mathbf{try}(\{k\})^\infty; \mathbf{acq}(k) \quad .$$

The static semantics will ensure that, if $(\sigma, k) \in [E_0]_@$, then $k \in O \cup \{\bigstar\}$ and that if $k \in O$ then the object implements the Lock interface.

When there is a guard we have

$$\begin{aligned}[(\mathbf{with}\ E_0\ \mathbf{when}\ E_1\ C\ \mathbf{with})]_! = & \bigcup_{(\sigma,k)\in[E_0]_@ \mid k\neq\bigstar} \mathbf{start}(\sigma,\emptyset); \mathbf{fin}(\emptyset,\delta(\sigma)); \text{enter}(k,E_1); [C]_!\,; \mathbf{rel}(k) \\ & \cup\ \{(\sigma,k)\in[E_0]_@ \mid k=\bigstar \cdot \langle\mathbf{chaos}(\sigma)\rangle\} \quad ,\end{aligned}$$

where

$$\begin{aligned}\text{enter}(k,E) &\subseteq P^\infty \\ \text{enter}(k,E) &= (\mathbf{try}(\{k\})^\infty; \mathbf{acq}(k); \text{filter}(\neg E); \mathbf{rel}(k))^\infty\,; \mathbf{try}(\{k\})^\infty; \mathbf{acq}(k); \text{filter}(E) \quad .\end{aligned}$$

3.7 Actions for locks

We need a meaning for the three new actions.

$$\begin{aligned}(\sigma_0, R_0, W_0, K_0, \text{true}) &\overset{\mathbf{try}(K)}{\rightarrowtail} (\sigma_0, R_0, W_0, K_0, \text{true}) \qquad \text{if } K \subseteq K_0 \quad , \\ (\sigma_0, R_0, W_0, K_0, \text{true}) &\overset{\mathbf{acq}(k)}{\rightarrowtail} (\sigma_0, R_0, W_0, K_0 \cup \{k\}\,, \text{true}) \qquad \text{if } k \notin K_0 \quad , \\ (\sigma_0, R_0, W_0, K_0, \text{true}) &\overset{\mathbf{rel}(k)}{\rightarrowtail} (\sigma_0, R_0, W_0, K_0 - \{k\}\,, \text{true}) \quad .\end{aligned}$$

3.8 Rendezvous

3.8.0 Simple rendezvous

Accept statements and calls are rather complicated. We start with the simplest case of

$$(\mathbf{accept}\ n()\ C\ \mathbf{accept}) \quad .$$

For each object o belonging to the class in which this statement appears we need two locks $\mathfrak{a}(o,n)$ and $\mathfrak{d}(o,n)$. In the initial configuration, all these locks are in the lock set, meaning they are locked.

Now a call by a client is

$$\begin{aligned}[E.n()]_! = & \bigcup_{(\sigma,o)\in[E]_@ \mid o\neq\bigstar} \mathbf{start}(\sigma,\emptyset); \mathbf{fin}(\emptyset,\delta(\sigma)); \mathbf{rel}(\mathfrak{a}(o,n)); \text{enter}(\mathfrak{d}(o,n)) \\ & \cup\ \{(\sigma,k)\in[E_0]_@ \mid k=\bigstar \cdot \langle\mathbf{chaos}(\sigma)\rangle\} \quad .\end{aligned}$$

The server's semantics, where o is the object containing the server thread,

$$[(\mathbf{accept}\ n()\ C\ \mathbf{accept})]_! = \mathrm{enter}(\mathfrak{a}(o,n)); [C]_!; \mathbf{rel}(\mathfrak{d}(o,n))$$

The locks here are being used as binary semaphores, with $\mathbf{rel}(k)$ being a V operation and $\mathrm{enter}(k)$ being a P operation. As the computation proceeds, the lock set evolves as

$$\begin{array}{l} \{\mathfrak{a}(o,n),\mathfrak{d}(o,n)\} \overset{\text{client releases } \mathfrak{a}(o,n)}{\longrightarrow} \{\mathfrak{d}(o,n)\} \overset{\text{server acquires } \mathfrak{a}(o,n)}{\longrightarrow} \\ \{\mathfrak{a}(o,n),\mathfrak{d}(o,n)\} \overset{\text{the body executes}}{\longrightarrow} \{\mathfrak{a}(o,n),\mathfrak{d}(o,n)\} \overset{\text{server releases } \mathfrak{d}(o,n)}{\longrightarrow} \\ \{\mathfrak{a}(o,n)\} \overset{\text{client acquires } \mathfrak{d}(o,n)}{\longrightarrow} \{\mathfrak{a}(o,n),\mathfrak{d}(o,n)\} \end{array}$$

3.8.1 Choice

Next we consider two methods that are both implemented by the accept statement.The semantics of calls remains the same:

$$(\mathbf{accept}\ n_0()\ C_0 \mid n_1()\ C_1\ \mathbf{accept}) \quad .$$

The server is

$$\begin{array}{ll} & [(\mathbf{accept}\ n_0()\ C_0 \mid n_1()\ C_1\ \mathbf{accept})]_! \\ = & \mathbf{try}(\{\mathfrak{a}(o,n_0),\mathfrak{a}(o,n_1)\})^\infty; \left(\begin{array}{ll} & \mathbf{acq}(\mathfrak{a}(o,n_0)); [C_0]_!; \mathbf{rel}(\mathfrak{d}(o,n_0)) \\ \cup & \mathbf{acq}(\mathfrak{a}(o,n_1)); [C_1]_!; \mathbf{rel}(\mathfrak{d}(o,n_1)) \end{array} \right) \end{array} \quad .$$

The implementation is no longer in terms of standard semaphore operations, as the **try** and **acq** actions are now split up and can not be put back together. Effectively, the server waits on several semaphores at once and then behaves differently depending on which one is successfully acquired.

The extention to more than two branches is straight forward.

3.8.2 Guards

Guarded accepts look like this:

$$(\mathbf{accept}\ n_0()\ \mathbf{when}\ E_0\ C_0 \mid n_1()\ C_1\ \mathbf{when}\ E_1\ \mathbf{accept}) \quad .$$

The semantics of guards is that they are evaluated once at the start of the accept statement and never again. To simplify the semantics, we break the sequence into three phases: evaluation of all the guards, waiting for a suitable call, and everything else.

$$\begin{array}{ll} & [(\mathbf{accept}\ n_0()\ \mathbf{when}\ E_0\ C_0 \mid n_1()\ C_1\ \mathbf{when}\ E_1\ \mathbf{accept})]_! \\ = & \begin{array}{ll} & \bigcup_{v \in \mathbb{B}^2} \mathit{phase0}(v,(E_0,E_1)); \mathit{phase1}(v,n_0,n_1); \mathit{phase2}(v,n_0,n_1,C_0,C_1) \\ \cup & \left\{(\sigma,v) \in [(E_0,E_1)]_\$ \mid v(0) = \bigstar \vee v(1) = \bigstar \cdot \langle \mathbf{chaos}(\sigma) \rangle \right\} \end{array} \end{array} \quad .$$

The first phase is the evaluation of the guards.

$$\mathit{phase0}(v,(E_0,E_1)) = \bigcup_{(\sigma,v)\in[(E_0,E_1)]_\$} \mathbf{start}(\sigma,\emptyset);\mathbf{fin}(\emptyset,\delta(\sigma)) \quad .$$

The second phase is waiting

$$\begin{array}{rcl}
\mathit{phase1}((\text{true},\text{true}),n_0,n_1) & = & \mathbf{try}(\{\mathfrak{a}(o,n_0),\mathfrak{a}(o,n_1)\})^\infty \\
\mathit{phase1}((\text{true},\text{false}),n_0,n_1) & = & \mathbf{try}(\{\mathfrak{a}(o,n_0)\})^\infty \\
\mathit{phase1}((\text{false},\text{true}),n_0,n_1) & = & \mathbf{try}(\{\mathfrak{a}(o,n_1)\})^\infty \\
\mathit{phase1}((\text{false},\text{false}),n_0,n_1) & = & \mathbf{try}(\emptyset)^\infty \quad .
\end{array}$$

The final phase is everything else

$$\begin{array}{rcl}
\mathit{phase2}((\text{true},\text{true}),n_0,n_1,C_0,C_1) & = & \left(\begin{array}{rl} & \mathbf{acq}(\mathfrak{a}(o,n_0));[C_0]_!;\mathbf{rel}(\mathfrak{d}(o,n_0)) \\ \cup & \mathbf{acq}(\mathfrak{a}(o,n_1));[C_1]_!;\mathbf{rel}(\mathfrak{d}(o,n_1)) \end{array}\right) \\
\mathit{phase2}((\text{true},\text{false}),n_0,n_1,C_0,C_1) & = & \mathbf{acq}(\mathfrak{a}(o,n_0));[C_0]_!;\mathbf{rel}(\mathfrak{d}(o,n_0)) \\
\mathit{phase2}((\text{false},\text{true}),n_0,n_1,C_0,C_1) & = & \mathbf{acq}(\mathfrak{a}(o,n_1));[C_1]_!;\mathbf{rel}(\mathfrak{d}(o,n_1)) \\
\mathit{phase2}((\text{false},\text{false}),n_0,n_1,C_0,C_1) & = & \langle\mathbf{miracle}\rangle \quad .
\end{array}$$

The **miracle** action is an action that is never enabled. Phase 1 together with the first action of phase 2 mean try until at least one of a set of locks is available and then acquire one lock in that set. When the set is empty, this forces an infinite sequence of trys.

3.8.3 Early return

In an accept statement, it is possible to return early. The client and server then proceed in parallel. The syntax (without guards) is

$$(\mathbf{accept}\ n_0()\ C_0\ \mathbf{then}\ D_0 \mid n_1()\ C_1\ \mathbf{then}\ D_1\ \mathbf{accept}) \quad .$$

The server semantics is

$$\begin{array}{rl}
& [(\mathbf{accept}\ n_0()\ C_0\ \mathbf{then}\ D_0 \mid n_1()\ C_1\ \mathbf{then}\ D_1\ \mathbf{accept})]_! \\
= & \text{trys}(\mathfrak{a}(o,n_0),\mathfrak{a}(o,n_1)); \left(\begin{array}{rl} & \mathbf{acq}(\mathfrak{a}(o,n_0));[C_0]_!;\mathbf{rel}(\mathfrak{d}(o,n_0));[D_0]_! \\ \cup & \mathbf{acq}(\mathfrak{a}(o,n_1));[C_1]_!;\mathbf{rel}(\mathfrak{d}(o,n_1));[D_1]_! \end{array}\right) \quad .
\end{array}$$

It is straightforward to extend the semantics for guarded accepts to early returns.

3.8.4 Parameters

Parameters fall into two categories, in parameters are copied in at the start, while out parameters are copied out at the end. We'll consider a method with no choice but with one in and one out parameter

$$(\textbf{accept}\ n(\textbf{in}\ b : T_0, \textbf{out}\ c : T_1)\ C\ \textbf{accept}) \quad .$$

As local variables, the parameters are considered field names, which I'll take to be $o.b$ and $o.c$.

We need the following locks. All are initially acquired.

- $\mathfrak{a}(o,n)$ — delays the server until it is called
- $\mathfrak{b}(o,n)$ — delays the caller until it is safe to copy the in parameters
- $\mathfrak{c}(o,n)$ — delays the server until the in parameters have been copied
- $\mathfrak{d}(o,n)$ — delays the client until it is time to copy the out parameters
- $\mathfrak{e}(o,n)$— delays the server until it the out parameters have been copied

Now the server semantics is just an elaboration on the parameter free case.

$$[(\textbf{accept}\ n()\ C\ \textbf{accept})]_! = \text{enter}(\mathfrak{a}(o,n)); \textbf{rel}(\mathfrak{b}(o,n)); \text{enter}(\mathfrak{c}(o,n)); [C]_!; \textbf{rel}(\mathfrak{d}(o,n)); \text{enter}(\mathfrak{e}(o,n)) \quad .$$

The client is responsible for copying the parameter data

$$[E_0.n(E_1,E_2)]_! = \bigcup_{\substack{(\sigma_0,o)\in[E_0]_@,(\sigma_1,v_1)\in[E_1]_\$,(\sigma_2,l)\in[E_2]_@,v_2 \\ |\sigma_0\smile\sigma_1\smile\sigma_2\wedge o\neq\bigstar\wedge v_1\neq\bigstar\wedge l\neq\bigstar}} \left(\begin{array}{l} \textbf{start}(\sigma_0\cup\sigma_1\cup\sigma_2, \{l, \text{field}(o,b)\}); \\ \textbf{fin}(\emptyset, \delta(\sigma_0\cup\sigma_1\cup\sigma_2)); \\ \textbf{rel}(\mathfrak{a}(o,n)); \\ \text{enter}(\mathfrak{b}(o,n)); \\ \textbf{fin}(\text{field}(o,b)\mapsto v_1, \emptyset); \\ \textbf{rel}(\mathfrak{c}(o,n)); \\ \text{enter}(\mathfrak{d}(o,n)); \\ \textbf{start}(\text{field}(o,c)\mapsto v_2, \{l\}); \\ \textbf{fin}(l\mapsto v_2, \{\text{field}(o,c)\}); \\ \textbf{rel}(\mathfrak{e}(o,n)) \end{array}\right)$$
$$\cup \left\{ \begin{array}{c} (\sigma_0,o)\in[E_0]_@, (\sigma_1,v_1)\in[E_1]_\$, (\sigma_2,l)\in[E_2]_@ \\ \mid o=\bigstar \vee v_1=\bigstar \vee l=\bigstar \cdot \langle\textbf{chaos}(\sigma)\rangle \end{array}\right\} \quad .$$

The sequence is may seem intimidating, but is understandable in smaller pieces.

- Evaluate all three expressions (start and fin)
- Signal the server and wait for an acknowledgement (rel and enter)
- Copy the in parameter (start and fin)
- Signal the server and wait for the server to finish (rel and enter)
- Copy the out parameter (start and fin)
- Signal the server that the out parameter has been copied (rel)

Adding choice, guarding and early returns to the parameter case is straight forward as the client semantics is not affected and the change in server semantics from no parameters to parameters is simply a matter of adding enter and rel sequences for the additional locks. Having more or fewer parameters is also conceptually straight forward.

4 Extending $\rightarrowtail$

The relation $\overset{p}{\rightarrowtail}$ can be extended to finite traces in the obvious way

$$\begin{array}{ll} x \overset{\epsilon}{\rightarrowtail} y & \text{iff } x = y \\ x \overset{ps}{\rightarrowtail} y & \text{iff } \exists z \cdot x \overset{p}{\rightarrowtail} z \overset{s}{\rightarrowtail} y, \text{ where } s \in P^* \end{array} \quad .$$

Next we extend $\overset{p}{\rightarrowtail}$ to infinite traces. Define $x \uparrow_s$ to mean there exists a function $f : \delta(s) \rightarrow Z$ such that

$$f(i) \overset{s(i)}{\rightarrowtail} f(i+1)$$

and $f(0) = x$. The existence of such a function means that from x, the computation s may proceed to the end of s, if s is finite, or infinitely, if s is infinite. Now define

$$x \overset{s}{\rightarrowtail} y \text{ iff } x \uparrow_s \text{, where } s \in P^\omega \quad .$$

I.e., an infinite computation is considered to be chaotic.

Now we define $\overset{S}{\rightarrowtail}$ for a trace set S by

$$x \overset{S}{\rightarrowtail} y \text{ iff } \exists s \in S \cdot x \overset{s}{\rightarrowtail} y$$

and $\overset{C}{\rightarrowtail}$ for a command C by

$$x \overset{C}{\rightarrowtail} y \text{ iff } x \overset{[C]_!}{\rightarrowtail} y \quad .$$

All these varieties of arrows can be extended to sets

$$X \rightarrowtail Y \text{ iff } \forall x \in X \cdot \exists y \in Y \cdot x \rightarrowtail y \quad .$$

Now

$$\mathrm{wp}(C, Y) = \left\{ x \in Z \mid x \overset{C}{\rightarrowtail} y \right\} \quad .$$