Storeless Semantics and Separation Logic
(many preliminary ideas, few technical results!)

David Schmidt (conversations with Anindya Banerjee)

Computing and Information Sciences Department
Kansas State University

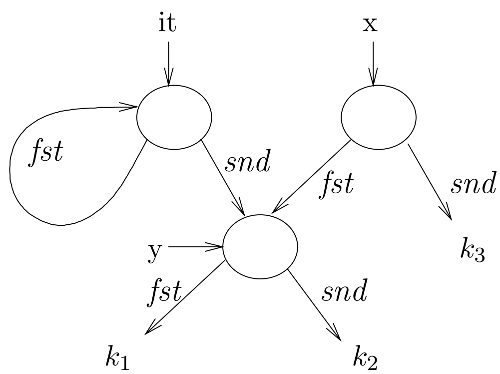
Storeless semantic models

Jonkers and then Deutsch proposed "storeless" semantic models for heap storage. Example: A heap of three pair objects,

$$\begin{array}{l} \ell_1 \mapsto \ell_1, \ell_2 \\ \ell_2 \mapsto k_1, k_2 \\ \ell_3 \mapsto \ell_2, k_3 \end{array}$$

where k_1, k_2, k_3 are non-pointer "constants" and x binds to ℓ_3 and y binds to ℓ_2. This might be modelled by

– a directed graph, where nodes represent objects and arcs represent fields held within the objects:



– a partitioned set of paths from the heap's "entry point," it, where each partition identifies an object:

$$\begin{array}{ll} \{fst^i \mid i \geq 0\} & \{fst^i.snd \mid i \geq 0\} \\ \{fst^i.snd.fst \mid i \geq 0\} & \{fst^i.snd^2 \mid i \geq 0\} \end{array}$$

– a set of paths from the heap's objects of interest to its entry point:

$$\{y.snd^{-1}.(fst^{-1})^i \mid i \geq 0\} \cup \{x.fst.snd^{-1}.(fst^{-1})^i \mid i \geq 0\}$$

These modellings hide the locations that name the objects.

The semantic models can be abstracted to finite-node "shape graphs" [Sagiv,Reps,Wilhelm] or to regular expressions that denote path sets [Deutsch] or to state names in finite automata, one automaton for the entry and each named object of interest [Blanchet].

Axiomatic logic is a storeless semantic model

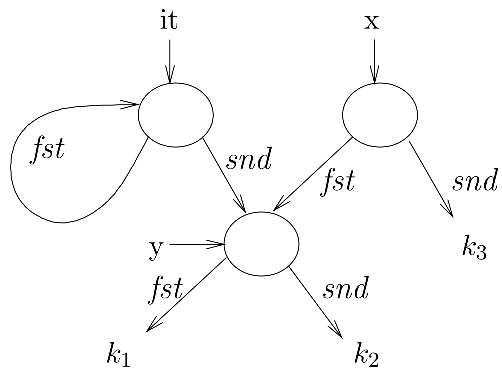
Hoare logic hides the store in its assertions, e.g.,

$$\{[\mathtt{E}/\mathtt{x}]P\}\ \mathtt{x{:}=\ E}\ \{P\}$$

defines the semantics of assignment while hiding x's location.

Perhaps a Hoare logic of pointers and objects will help us understand and formalize "storeless semantics" and its abstractions?

For



we can use the assertion language of Reynolds's and O'Hearn's *separation logic* to assert the heap shape. Here are some assertions that describe the graph with respect to it:

$$\exists \ell . \exists m.\ (it \mapsto it, \ell)\ *\ (\ell \mapsto k_1, k_2)\ *\ (m \mapsto \ell, k_3)$$

$$\exists \ell . \exists m.\ (it \mapsto it, \ell)\ *\ (\ell \mapsto k_1, k_2)\ *\ true$$

The assertion language of separation logic is a language for defining storeless semantic models.

Separation logic of heap storage

Developed by Ishtiaq, O'Hearn, Pym, Reynolds, and Yang, separation-logic is a Hoare logic for reasoning about imperative programs that use heap storage, in terms of Hoare triples, $\{p\}S\{p'\}$.

$$
\begin{array}{lcl}
S & ::= & \mathtt{x} := E \;\mid\; \mathtt{x} := E.i \;\mid\; E.i := E' \;\mid\; \mathtt{x} := \mathtt{new}\, C(E_1, E_2) \;\mid\; \cdots \\
E & ::= & k \;\mid\; E\; op\; E' \;\mid\; \mathtt{x} \;\mid\; \cdots \\
p & ::= & \alpha \;\mid\; \mathit{false} \;\mid\; p \supset p' \;\mid\; \exists x.p \;\mid\; \mathbf{emp} \;\mid\; p * p' \;\mid\; p \mathrel{-\!\!*} p' \\
\alpha & ::= & E_1 = E_2 \;\mid\; E \mapsto E_1, E_2
\end{array}
$$

For simplicity, assume that objects, a, hold exactly two fields, *fst* and *snd*, e.g, $(a \mapsto b, c)$ where $a.\mathit{fst} = b$ and $a.\mathit{snd} = c$.

Predicates, *p*, express properties about heap shape. Here is a sample Hoare triple, which asserts pre- and post-conditions about a two-object heap:

$$\{\exists a.(\mathtt{x} \mapsto a, b) \;*\; (a \mapsto b, b)\}\; \mathtt{x.fst} := \;\mathtt{x}\; \{\exists a.(\mathtt{x} \mapsto \mathtt{x}, b) \;*\; (a \mapsto b, b)\}$$

Here, $\mathtt{x}$ and b are free identifiers — local variables or "dangling pointers."
Intuitions about the new logical connectives:

- $E \mapsto E_1, E_2$ holds for heap, h, iff h consists of just *one cell* whose address is E and whose contents are E_1, E_2.
- $p_1 * p_2$ holds for heap, h, iff h can be partitioned into $h = h_1 \cdot h_2$, and p_i holds for h_i.
- $p' \mathrel{-\!\!*} p$ holds for heap h iff when h is extended by some h' that makes p' hold, then p holds for the extended heap, $h' \cdot h$.
- $*$ and $\mathrel{-\!\!*}$ are categorical adjoints.

Separation logic proves correctness properties

Because we can assert disjointness of objects, we can write assertions, like this one, which defines (tail-)noncircular lists:

$$ncList(\ell) \text{ iff}_{lfp} \ (\ell \doteq null) \ \lor \ ((\ell \mapsto hd, tl) * ncList(tl))$$

The asterisk ensures that all objects in the list's tail are disjoint from the front object, addressed at ℓ. We might prove that a copy function constructs a noncircular list:

```
copy(C x) {
  {ncList(x)}
    y := if x = null
            then x
            else new C(x.fst, copy(y.snd));
  return y;
  {ncList(y)}
}
```

Concrete semantics of statements

A statement, S, operates on a state:

$$\begin{aligned}
&State = Stack \times Heap\\
&Stack = Var \rightarrow Value\\
&Heap = Location \rightarrow Object\\
&Object = Value \times Value\\
&Value = Constant \cup Location
\end{aligned}$$

$$[\![E]\!] : Stack \rightarrow Value$$

$$[\![S]\!] : State \rightarrow State$$

Here are some tersely stated semantics definitions:

$$[\![\mathtt{x} := E]\!]s, h = [s|\mathtt{x} = [\![E]\!]s], h$$

$$[\![\mathtt{x} := E.i]\!]s, h = [s|\mathtt{x} = h([\![E]\!]s).i], h$$

$$[\![E.fst := E']\!]s, h = s, [h|h([\![E]\!]s) \mapsto ([\![E']\!]s, h([\![E]\!]s).snd)]$$

$$[\![\mathtt{x} := \mathtt{new}\, C(E_1, E_2)]\!]s, h = [s|\mathtt{x} = \ell_{fresh}], [h|\ell_{fresh} \mapsto ([\![E_1]\!]s, [\![E_2]\!]s)]$$

Semantics of predicates

Let $h \# h'$ denote that (the domains of) heaps h and h' are disjoint.

$s, h \models \mathit{false}$	never
$s, h \models p \supset p'$	iff $s, h \models p$ implies $s, h \models p'$
$s, h \models \exists x.p$	iff exists $v \in \mathit{Value}$ s.t. $[s \mid x \mapsto v], h \models p$
$s, h \models E{=}E'$	iff $[\![E]\!]s \; = \; [\![E']\!]s$
$s, h \models E \mapsto E_1, E_2$	iff $\{[\![E]\!]s\} = \mathit{dom}(h),\; h([\![E]\!]s) = r,$ $r.\mathit{fst} = [\![E_1]\!]s,$ and $r.\mathit{snd} = [\![E_2]\!]s$
$s, h \models \mathbf{emp}$	iff $\mathit{dom}(h) = \{\}$
$s, h \models p * p'$	iff there exist h_0, h_1 such that $h_0 \# h_1$, $h = h_0 \cdot h_1,\; s, h_0 \models p,$ and $s, h_1 \models p'$
$s, h \models p \mathrel{-\!\!*} p'$	iff for all h', if $h' \# h$ and $s, h' \models p$, then $h \cdot h' \models p'$

Remember that

- $E \mapsto E_1, E_2$ holds only for a *one-object heap* whose domain is address E
- $\mathbf{emp}$ holds only for the empty heap
- assertions p and p' about disjoint regions of heap are composed as $p * p'$
- $p' \mathrel{-\!\!*} p$ describes possible future worlds, say, due to object allocation

A language of pairs

We use separation-logic assertions to define a concrete semantics for a language of pair objects (let $k \in Const$):

$$e \;\; ::= \;\; k \;\; | \;\; (e_1, e_2) \;\; | \;\; e.i \qquad i \in \{\mathit{fst}, \mathit{snd}\}$$

(This language is much simpler to study than the imperative language.)

A expression evaluates to a Value and constructs objects in the heap:

$$\begin{array}{l} \mathit{Value} = Const \cup \mathit{Location} \\ Heap = \mathit{Location} \rightarrow Object \\ Object = Value \times Value \end{array}$$

Here are the big-step concrete semantics rules, where a configuration, $\vdash e \Downarrow v, h$, asserts that expression e evaluates to v and constructs subheap h:

$$\vdash k \Downarrow k, \{\}$$

$$\frac{\vdash e_i \Downarrow v_i, h_i \qquad i \in 1..2}{\vdash (e_1, e_2) \Downarrow \ell_{\mathit{fresh}},\; (\ell_{\mathit{fresh}} \mapsto v_1, v_2) \;\; * \;\; h_1 \;\; * \;\; h_2}$$

$$\frac{\vdash e \Downarrow \ell, h \qquad (\ell \mapsto v_1, v_2) \in h}{\vdash e.\mathit{fst} \Downarrow v_1, h}$$

Note: $*$ is used as heap concatenation, and $\ell \mapsto v_1, v_2$ represents an object.

We can read a pair, v, h, as the state, $[it = v], h$.

For example, we can deduce that

$$\vdash ((k_1, k_2).snd,\; (k_3, k_4)) \Downarrow \ell_3,\; (\ell_1 \mapsto k_1, k_2) * (\ell_2 \mapsto k_3, k_4) * (\ell_3 \mapsto k_2, \ell_2)$$

Garbage sensitivity

When we deduce $\vdash e \Downarrow v, h$, we can read this

- *operationally*, as "e computes to the state, $[it = v], h$"
- *axiomatically*, as "e has the property that $\bar{\exists}\ell_i.it = v \wedge h$"

Example:

$$\vdash (k_1, k_2).snd \Downarrow k_2, (\ell_1 \mapsto k_1, k_2)$$

— which computes the state, $[it = k_2], (\ell_1 \mapsto k_1, k_2)$;
— which asserts $\exists \ell_1.it = k_2 \wedge \ell_1 \mapsto k_1, k_2$.

(Of course, if $\vdash e \Downarrow v, h$, then $[it = v], h \models \bar{\exists}\ell_i.it = v \wedge h$.)

When we read the concrete semantics as an axiomatic logic, we see it is "garbage sensitive," that is, assertions can be invalidated by garbage collection.

For example, if we garbage-collect the above heap, producing the state, $[it = k_2], []$, we see that

$$[it = k_2], [] \not\models \exists \ell_1.it = k_2 \wedge \ell_1 \mapsto k_1, k_2$$

This warns us: An axiomatic logic for the pairs language should derive "garbage insensitive" assertions.

Paths semantics of the pairs language

We write the big-step definition of the paths semantics for the language:

A expression evaluates to a $Pathset$:

$$\begin{aligned}&Pathset = \mathcal{P}(Path)\\&Path = Const.Selector^*\\&Selector = \{\mathit{fst}^{-1}, \mathit{snd}^{-1}, \mathit{fst}, \mathit{snd}\}\end{aligned}$$

We use Blanchet's presentation: a path travels from a point of interest (here, the constants) to the result of the expression ("it"):

$$\vdash k \Downarrow \{k\}$$

$$\frac{\vdash e_i \Downarrow S_i \quad i \in 1..2}{\vdash (e_1, e_2) \Downarrow S_1 \circ \mathit{fst}^{-1} \ \cup \ S_2 \circ \mathit{snd}^{-1}}$$

$$\frac{\vdash e \Downarrow S}{\vdash e.\mathit{fst} \Downarrow S \circ \mathit{fst}}$$

Note: $\circ$ is path composition,

$$S \circ i = \{s \cdot i \mid s \in S\}, \quad i \in \{\mathit{fst}, \mathit{snd}, \mathit{fst}^{-1}, \mathit{snd}^{-1}\}$$

where destructors cancel constructors:

$$p \cdot \mathit{fst}^{-1} \cdot \mathit{fst} = p$$

The analysis calculates a semantics that lists the "paths of interest" to it and is a sound semantics with respect to the concrete semantics.

The definition of $\circ$ implies a kind of "garbage collection." For example,

$$\vdash ((k_1, k_2).snd,\ (k_3, k_4)) \Downarrow \{k_2.\mathit{fst}^{-1}, \ \ k_3.\mathit{fst}^{-1}.snd^{-1}, \ \ k_4.snd^{-1}.snd^{-1}\}$$

because

$$\vdash (k_1, k_2).snd \Downarrow \{k_1.\mathit{fst}^{-1}, \ \ k_2.snd^{-1}\} \circ snd \ = \ \{k_2\}$$

But more precisely, the definition of $\circ$ makes the assertions garbage insensitive.

Paths semantics translated into separation logic

We can use separation logic to better understand paths analysis, by translating the paths into assertions in the logic. For example, the path $k.\mathit{fst}^{-1}$ will be written hereon as $k.\mathit{fst}^{-1} = it$,

which will abbreviate the assertion, $(it \mapsto k, \_)$

where $a \mapsto b, \_$ abbreviates $\exists c.a \mapsto b, c$. The assertion holds true for a heap region that holds the object named by it whose first component is k.

The abbreviations generalize to finite sequences, e.g., $k.\mathit{fst}_1^{-1}.\cdots.\mathit{fst}_n^{-1} = it$ abbreviates

$$\exists \ell_1 \cdots \exists \ell_{n-1}.(it \mapsto \ell_{n-1}, \_) * (\ell_{n-1} \mapsto \ell_{n-2}, \_) * \cdots * (\ell_1 \mapsto k, \_)$$

Such a path identifies a heap region in which one can traverse from k to the entry location, it.

Dually, a path with a destructor, like $a.\mathit{fst} = it$, is expressed

$$(a \mapsto it, \_)$$

Destructors should cancel constructors, e.g.,

$k.\mathit{fst}^{-1}.\mathit{fst} = it$ lets us deduce that $k = it$

and we must give a translation into separation logic that supports this. Let $p \in Path$; translate $p = it$ as $[\![p]\!]_{it}$, where

$$p \quad ::= \quad k \quad | \quad p.\mathit{fst} \quad | \quad p.\mathit{fst}^{-1}$$

$$\begin{aligned}
&[\![k]\!]_{it} = (k = it)\\
&[\![p.\mathit{fst}^{-1}]\!]_{it} = \exists \ell.(it \mapsto \ell, _) * [\ell/it][\![p]\!]_{it}\\
&[\![p.\mathit{fst}]\!]_{it} = \exists m.[m/it][\![p]\!]_{it} \wedge ((m \mapsto it, _) * true)
\end{aligned}$$

The first clause asserts no objects are needed to proceed from k to it; the second clause asserts that the objects that constitute p are augmented by one more; the third clause asserts that the objects along path p include a head object, m.

For example,

$$[\![k.\mathit{fst}^{-1}.\mathit{fst}]\!]_{it} = (\ \exists m \exists \ell.((m \mapsto \ell, \_) * k = \ell) \wedge ((m \mapsto it, \_) * true)\ ) \ \Rightarrow\ k = it$$

A set of paths, S, is translated to $\bigwedge_{p \in S} [\![p]\!]_{it}$

Now, we might formalize the definition of $\circ$ in separation logic as an assertion-normalization operation and prove it sound and prove its results as garbage insensitive.

Inserting heap regions into the paths semantics

We can analyze the heap's internal structure in finer detail. For example, the construction, (e_1, e_2), assembles the subheap constructed by e_1 with that constructed by e_2 and a new object. We revise the semantics to show the subheaps.

Now, assertions will have this normal form:

$$S_{x_1} * S_{x_2} * \cdots * S_{x_n}$$

where each S_{x_i} has the form, $\{v.s^* = x_i\}$, describing paths from values to the heap region entry point, x_i. (The $*$ asserts that the paths in a region, S_x, are using objects that are disjoint from all other heap regions, S_y, $Y \neq x$.)

The entry points can be it or *an entry point of a disjoint heap region.*

$$\begin{aligned}
&k \in Const\\
&v \in Leaf = Const \cup Id\\
&x \in Entry = Id \cup \{it\}\\
&s \in Selector = \{fst, snd, fst^{-1}, snd^{-1}\}\\
&p \in Path = Leaf.Selector^* = Entry
\end{aligned}$$

Here are the revised rules:

$$\vdash k \Downarrow \{k = it\}$$

$$\frac{\vdash e_1 \Downarrow S_1 \qquad \vdash e_2 \Downarrow S_2}{\vdash (e_1, e_2) \Downarrow \{m.fst^{-1} = it, \;\; n.snd^{-1} = it\} \; * \; [m/it]S_1 \; * \; [n/it]S_2}$$

(Note: m and n are implicitly existentially quantified.)

The third rule remains the same:

$$\frac{\vdash e \Downarrow S}{\vdash e.fst \Downarrow S \circ fst} \text{ where } \quad \begin{aligned}&(S_{x_1} * S_{x_2} * \cdots * S_{x_n} * S_{it}) \circ i\\ &\quad = S_{x_1} * S_{x_2} * \cdots * S_{x_n} * (S_{it} \circ i)\end{aligned}$$

Suspended paths

When we introduced subregions, we introduced possible "suspended" paths. For example, this heap description,

$$\{k.\mathit{fst}^{-1} = m\} * \{m.\mathit{fst} = it\}$$

should reduce to $k = it$, by merging the two regions.

But if m is free, e.g., the interface point to an external region, then the path, $m.\mathit{fst} = it$, is "suspended" — the destructor cannot cancel a constructor.

To understand suspended paths and region merging, we provide a more detailed translation of paths into the assertion language of separation logic:

$$p \ ::= \ k \ \mid \ x \ \mid \ p.\mathit{fst} \ \mid \ \mathit{fst}^{-1}$$

A path, $p = x$, is translated as $[\![p]\!]_x$:

$[\![k]\!]_x = (k = x)$

$[\![p.\mathit{fst}^{-1}]\!]_x = \exists \ell.(x \mapsto \ell, \_) * [\ell/x][\![p]\!]_x$

$[\![p.\mathit{fst}]\!]_x = \exists m.\ ([m/x][\![p]\!]_x) \ * \ (\forall v.\ (m \mapsto v, \_) \mathrel{-\!\!*} ((m \mapsto x, \_) \wedge v = x)\ )$

Recall the semantics of $p \mathrel{-\!\!*} p'$:

$$s, h \models p \mathrel{-\!\!*} p' \quad \text{iff} \quad \text{for all } h',\ \text{if } h' \# h \text{ and } s, h' \models p,\ \text{then } h \cdot h' \models p'$$

The point is: $(m \mapsto \_, \_)$'s presence is not guaranteed; the path is "suspended" until the antecendent object is supplied.

Since $*$ and $\mathrel{-\!\!*}$ are adjoints, the cancellation properties of paths are preserved. For example,

$$\begin{array}{l}
[\![k.\mathit{fst}^{-1}.\mathit{fst}]\!]_{it} \\
= \exists m.(m \mapsto k, _) \ * \ (\forall v.(m \mapsto v, _) \mathrel{-\!\!*} ((m \mapsto it, _) \wedge v = it)) \\
\Rightarrow \exists m.(\mathbf{emp} * (m \mapsto k, _)) \ * \ ((m \mapsto k, _) \mathrel{-\!\!*} ((m \mapsto it, _) \wedge k = it)) \\
\Rightarrow \exists m.\mathbf{emp} * ((m \mapsto k, _) \ * \ ((m \mapsto k, _) \mathrel{-\!\!*} ((m \mapsto it, _) \wedge k = it)))
\end{array}$$

The adjunction, $A * (A \mathrel{-\!\!*} B) \Rightarrow B$, lets us deduce that

$$[\![k.\mathit{fst}^{-1}.\mathit{fst}]\!]_{it} \Rightarrow \exists m.(m \mapsto it, \_) \wedge k = it \Rightarrow k = it$$

The translation as a concretization map

The translation of a path into a separation-logic assertion *suggests* that the latter is a concretization of the former.

This *suggests* that the translation, $[\![\,\cdot\,]\!]_m$, induces an upper adjoint of a Galois connection between $\mathcal{P}(SeprationLogicAssertion)$ and $\mathcal{P}(Path)$.

This also *suggests* that the lower adjoint of the Galois connection defines a "reachability analysis" of a set of assertions in separation logic.

But the details are not yet developed.

Escape analysis and garbage collection with suspended paths

Let $x \in \mathit{Identifier}$ label an object of interest or an extry point into a heap region. We might "suspend" x to learn which components of an expression's answer use x's storage components:

If $\vdash e \Downarrow S$ and there exists a path, $x.s^* \in S$, then x *escapes* from e's computation.

Example: let x and z label objects of interest, and let y name a suspended region of heap objects. We can deduce that

$$\begin{aligned}\vdash x\colon ((y\colon (k_1, k_2)).snd,\ (z\colon (k_3, k_4)).\mathit{fst}) \\ \Downarrow \{x = it, y.snd.\mathit{fst}^{-1} = it, k_3.snd^{-1} = it\} \\ * \ \{k_1.\mathit{fst}^{-1} = y, k_2.snd^{-1} = y\}\end{aligned}$$

Hence, object x escapes and objects within region y is needed to compute the result.

Because path assertions are garbage-insensitive, object z can be garbage collected. Indeed, if we prove a "soundness" result (namely, that paths analysis generates all paths that might appear in the computation's result), then we can argue that z's absence justifies garbage-collecting the object z names.

Adding identifiers

Syntax:

$$e \quad ::= \quad k \quad | \quad (e_1, e_2) \quad | \quad e.i \quad | \quad x \quad | \quad \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

$$\begin{array}{l} \mathit{Store} = \mathit{Identifier} \rightarrow \mathit{Value} \\ \mathit{Heap} = \mathit{Location} \rightarrow \mathit{Object} \end{array}$$

A configuration has the format

$$s, h \vdash e \Downarrow v, h'$$

where v, h' can be read as $[it = v], h'$.

The concrete semantics:

$$s, h \vdash k \Downarrow k, []$$

$$\frac{s, h \vdash e_i \Downarrow v_i, h_i \qquad i \in 1..2}{s, h \vdash (e_1, e_2) \Downarrow \ell_{\mathit{fresh}},\ (\ell_{\mathit{fresh}} \mapsto v_1, v_2)\ *\ h_1\ *\ h_2}$$

$$\frac{s, h \vdash e \Downarrow \ell, h' \qquad (\ell \mapsto v_1, v_2) \in h' * h}{s, h \vdash e.\mathit{fst} \Downarrow v_1, h'}$$

$$s, h \vdash x \Downarrow s(x), []$$

$$\frac{s, h \vdash e_1 \Downarrow v_1, h_1 \qquad [s|x = v_1], h_1 * h \vdash e_2 \Downarrow v_2, h_2}{s, h \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, h_1 * h_2}$$

Paths semantics (Blanchet)

$$\vdash k \Downarrow \{k = it\} \qquad\qquad \vdash x \Downarrow \{x = it\}$$

$$\frac{\vdash e_1 \Downarrow S_1 \qquad \vdash e_2 \Downarrow S_2}{\vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow [S_1/x]S_2}$$

This makes `let` into a syntactic device and generates suspended paths whenever a let-defined identifier is referenced.

Beyond paths

Let's try to write a sound-and-relatively-complete axiomatic semantics for the pairs language that derives assertions in full separation logic. Configurations take the form,

$$M \vdash e \Downarrow S$$

where M is an assertion about the state in which e computes, and S is an assertion about the result state generated by e. That is, $M \vdash e \Downarrow S$ is sound iff, for all s, h,

if $s, h \models M$
and $s, h \vdash e \Downarrow v, h'$ in the concrete semantics ,
then $[s|it = v], h' \models S$

Syntax:

$$e \quad ::= \quad k \quad | \quad x \quad | \quad \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \quad | \quad (e_1, e_2) \quad | \quad x.i$$

Let $E_1 \doteq E_2$ abbreviate $E_1 = E_2 \wedge \mathbf{emp}$.

$$\mathbf{emp} \vdash k \Downarrow it \doteq k$$

$$[x/it]P \wedge \mathbf{emp} \vdash x \Downarrow P \wedge \mathbf{emp}$$

$$\frac{M \vdash e_1 \Downarrow S_1 \qquad (\exists x.M) * [x/it]S_1 \vdash e_2 \Downarrow S_2}{M \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow \exists x.[x/it]S_1 * S_2}$$

(When evaluating the body of the `let`, we must remember that some existing paths might reference the hidden name, x, and not the local one.)

$$\frac{M \vdash e_1 \Downarrow S_1 \qquad M \vdash e_2 \Downarrow S_2}{M \vdash (e_1, e_2) \Downarrow \exists m.\exists n.(it \mapsto m, n) * [m/it]S_1 * [n/it]S_2}$$

$$(x \mapsto E_1, E_2) \vdash x.fst \Downarrow it \doteq E_1$$

These rules can generate garbage-sensitive assertions; indeed, we can compute the concrete semantics with them.

We require some structural rules to aid the basic ones; first, there is a Consequence Rule:

$$\frac{M' \Rightarrow M \quad M \vdash e \Downarrow S \quad S \Rightarrow S'}{M' \vdash e \Downarrow S'}$$

where $S \Rightarrow S'$ iff for all s, h, $s, h \models S$ implies $s, h \models S'$.

Next, there is a Frame Rule:

$$\frac{M \vdash e \Downarrow S}{M' * M \vdash e \Downarrow S}$$

There is also quantifier rule:

$$\frac{M \vdash e \Downarrow S}{\exists \ell . M \vdash e \Downarrow \exists \ell . S} \ \ \ell \text{ not free in } e$$

and a substutution rule:

$$\frac{M \vdash e \Downarrow S}{[E_i/x_i](M \vdash e \Downarrow S)}$$

Garbage sensitivity

It is inconvenient to prove explicitly that an assertion is garbage insensistive. O'Hearn and his colleagues have shown that this restricted syntax of assertions:

$$\begin{array}{lcl} p & ::= & \alpha \mid \mathit{false} \mid p \supset p' \mid \exists x.p \\ \alpha & ::= & E_1 = E_2 \mid E \mapsto E_1, E_2 \end{array}$$

is garbage insensitive, when the above formulas are translated by the double negation translation and interpreted in an intuitionistic model theory. (Heaps are partially ordered: $h \sqsubseteq h'$ iff $graph(h) \subseteq graph(h')$.)

Future research

This work is not yet completed; many basic results must be proved.

Our long-term objective is to develop a methodology for modular static analysis, where the modularity can be based on either

- program component (class or function)
- storage region (storage hierarchies or encapsulation levels)

We want to analyze languages that dynamically allocate storage and use a "controlled" assignment — assignment restricted to a storage region. Since assignment does a free-wheeling "swing" of a pointer variable, we had big trouble adapting path-based analyses to storage regions named by source-program identifiers. So, we are starting from scratch to understand what we can do with named regions and "pointer swing."

References

The slides for this talk: http://www.cis.ksu.edu/~schmidt/papers

Peter O'Hearn's web page:
http://www.dcs.qmul.ac.uk/~ohearn/

John Reynolds's web page (his course notes and LICS2002 paper, in particular):
http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/jcr/www/

Bruno Blanchet's PhD thesis:
http://www.di.ens.fr/~blanchet/escape-eng.html