Marrying Hoare Logic to Shape Analysis

Anindya Banerjee and David Schmidt

Computing and Information Sciences Department
Kansas State University

What's been happening

Last year's meeting: I was "cautious" about the prospects of integrating work at Kansas with the projects at UC Berkeley.

Last October: I spent some time at Berkeley; Alex reassured me that it was OK to study foundations that would prove useful to OSQ; e.g., uncover the truths within [Rehof & Fähndrich POPL 2001]:

- CFGs for simply typed lambda calculus [Mossin]
- context-free reachability [Reps]
- polymorphic recursion [Mycroft, Henglein]
- 0-CFA as a subtyping logic [Palsberg]

About the same time: Anindya Banerjee came to Kansas and promoted O'Hearn/Reynolds BI-logic.

After some significant background work, I became bothered about understanding modular analysis; it seemed to underlie most of what I was reading....

Objectives

- study formats of modular data-flow/static analysis
- study a canonical example: shape/escape analysis
- develop a semantical foundation in terms of BI (bunched-implication) logic [Ishtiaq and O'Hearn POPL01].

Aspects of flow analysis

- flow insensitive (order of statements ignored) versus flow sensitive
- context insensitive (monovariant) versus context sensitive (polyvariant)
- global (whole program) versus local (modular)

Due to lack of time today `)-:` , we skip past the first two items and move directly to the third....

Flow sensitivity

Consider

```
function f() { a = &b;
               g(a, b);
               *a = &c; }
```

A flow-insensitive analysis ignores the order of statements:

$$\mathcal{A}[\![\mathtt{C}_1;\mathtt{C}_2]\!] = \mathcal{A}[\![\mathtt{C}_1]\!] \sqcup \mathcal{A}[\![\mathtt{C}_2]\!]$$

For example,

$$\mathcal{A}[\![\mathtt{f}()\{...\}]\!] \;=\; \{a \hookrightarrow b\} \;\cup\; \mathcal{A}[\![\mathsf{g}(\mathsf{a},\mathsf{b})]\!] \;\cup\; \{\forall\alpha.\ \alpha \hookrightarrow c \text{ if } c \hookrightarrow \alpha\}$$

In contrast, a flow-sensitive analysis remembers statement ordering:

$$\mathcal{A}[\![\mathtt{C}_1;\mathtt{C}_2]\!] = \mathcal{A}[\![\mathtt{C}_2]\!] \circ \mathcal{A}[\![\mathtt{C}_1]\!]$$

So,

$$\mathcal{A}[\![\mathtt{f}()\{...\}]\!] \;=\; \lambda\sigma.\ (*a \hookrightarrow c)\,(\mathcal{A}[\![\mathsf{g}(\mathsf{a},\mathsf{b})]\!]\,((a \hookrightarrow b)\sigma))$$

Aspects of context sensitivity

Consider these three functions:

```
function f() { ...                function h(x) {
\alpha:        h(a);                 ...
               ... }               }

function g() { ... for (i = ...)
\beta:             { ... h(b_i); ... }
               ... }
```

A monovariant (context insensitive) analysis analyses each function component as if it were invoked once, somewhat like this:

$$\begin{aligned} let\ h &= \mathcal{A}[\![\mathtt{h(x)}]\!](\mathcal{A}[\![\mathtt{a}]\!] \ \bigsqcup \ \sqcup \{\mathcal{A}[\![\mathtt{b}_i]\!] | i \in ...\}) \\ in\ ...\ f &= \mathcal{A}[\![\mathtt{f()}]\!] h\ ... \\ ...\ g &= \mathcal{A}[\![\mathtt{g()}]\!] h\ ... \end{aligned}$$

This analyzes call-returns as if they are gotos.

Polyvariant analysis

Functions are analysed as invoked multiple times:

1. By means of *call site* (or call string/calling context):

$$\begin{aligned} let\ alpha \ &= \ \mathcal{A}[\![\mathtt{h(x)}]\!](\mathcal{A}[\![a]\!]) \\ let\ beta \ &= \ \mathcal{A}[\![\mathtt{h(x)}]\!](\sqcup_i \mathcal{A}[\![b_i]\!]) \\ in\ ...\ f \ &= \ \mathcal{A}[\![\mathtt{f()}]\!] alpha\ ... \\ ...\ g \ &= \ \mathcal{A}[\![\mathtt{g()}]\!] beta\ ... \end{aligned}$$

2. By means of *argument* (actual parameter); a function's denotation has arity,

$$\mathcal{A}[\![\mathtt{h(x)}]\!] \in \{\mathcal{A}[\![a]\!], ..., \mathcal{A}[\![b_i]\!], ...\} \rightarrow Result$$

   If the domain has low cardinality, then a finite number of specializations can be constructed, *à la* type inference with finite-intersection types.

   If the domain has high cardinality, then a schematic representation of $\mathcal{A}[\![\mathtt{h(x)}]\!]$ is required, like type inference with universally quantified types and/or polymorphic recursion [Mycroft 84, Henglein TOPLAS 93].

Aspects of global versus local (modular) analysis

Assume that a program consists of n components (functions) and some linkage code:

```
main() {                          function f1(...) {
 ... f1 ... fn ...                    ... fj ...
}                                 }
                               ...
                                  function fn(...) {
                                      ... fj ...
                                  }
```

A *global analysis* calculates all denotations simultaneously:

$$(lfp\ \lambda(m, a_1, ..., a_n).\ (\mathcal{A}[\![\texttt{main}()\{...\}]\!](m, a_1, ..., a_n), \mathcal{A}[\![\texttt{f1}(...)\{...\}]\!](a_1, ..., a_n), \cdots, \mathcal{A}[\![\texttt{fn}(...)\{...\}]\!](a_1, ..., a_n)) \downarrow 1$$

A *local analysis* first solves each component individually and then combines the analyses at link-time:

$$\{\ let\ a_i = lfp\ \lambda a_i.\ \mathcal{A}[\![\texttt{fi}(...)\{...\}]\!](a_1, ..., a_n)\ \}_{i \in 1..n}$$
$$in\ lfp\ \lambda m.\ \mathcal{A}[\![\texttt{main}()\{...\}]\!](m, a_1, ..., a_n)$$

Reanalysis of the `fi`'s at link-time is unnecessary. How we resolve the free references, $a_1, ...a_{i-1}, a_{i+1}, ...a_n$, in the definition of each a_i characterizes the form of local analysis.

Forms of local (modular) analysis [Cousot & Cousot CC'02]

$$\begin{array}{l} let\ \{a_i\ =\ lfp\ \lambda a_i.\ \mathcal{A}[\![\texttt{fi}(...)\{...\}]\!](a_1, ..., a_n)\}_{i \in 1..n} \\ in\ lfp\ \lambda m.\ \mathcal{A}[\![\texttt{main}()\{...\}]\!](m, a_1, ..., a_n) \end{array}$$

Simplification: The free references are left free, and internal simplication (partial evaluation) is performed on each $\mathcal{A}[\![\texttt{fi}(...)\{...\}]\!]$, giving denotations of the form, $lfp\ \lambda a_i.f_i(a_1, ..., a_n)$. The suspended lfp is solved at link-time by "flattening" the nested fixed points into the global one.

An example is an analysis that generates a constraint set for each `fi` component; the sets are unioned and solved at link time.

Worst-case: The free references are mapped to worst-case ("don't know") denotations:

$$let\ a_i\ =\ lfp\ \lambda a_i.\ \mathcal{A}[\![\texttt{fi}(...)\{...\}]\!](\top, ..., a_i, ..., \top)$$

This is simple and safe; it produces an analysis that can be precise at "first order" and is imprecise at "higher order."

Interface: The user specifies an *interface* (an approximate denotation), v_j, for each a_j. We calculate

$$a_i \;=\; \mathit{lfp}\ \lambda a_i.\ \mathcal{A}[\![\mathtt{fi}(...)\{...\}]\!](v_1, ..., v_{i-1}, a_i, v_{i+1}, ..., v_n)$$

and then check that a_i satisfies its interface: $a_i \sqsubseteq v_i$.

This approach is standard to data-type checking of modular programs. A variation is to synthesize the interfaces with a quick, crude global analysis and then do an interface-local analysis.

Symbolic relational: The free references are kept free, but creative effort *solves the local least-fixed point calculation for* a_i and synthesizes a schematic transfer function (or symbolic relational representation) of the form,

$$\mathit{let}\ a_i \;=\; \lambda a_j. \cdots \lambda a_n. v_i[a_j, ..., a_n]$$

where $v_i[a_j, ..., a_n]$ is some nonrecursive schematic/algorithmic expression.

One notable example: ML-style type inference, especially Mycroft's algorithm for inferring types for polymorphic recursion.

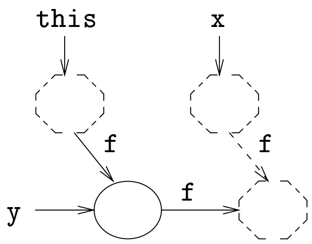
Shape analysis

Shape analysis has been defined in virtually all the variants of flow analysis described; it predicts the heap objects and links generated/used by a program [Emami, Ghiya, & Hendren PLDI94; Landi & Ryder PLDI92; Liang & Harold SAS01; Sagiv, Reps, & Wilhelm TOPLAS98, POPL99; Whaley & Rinard OOPSLA99; many others].

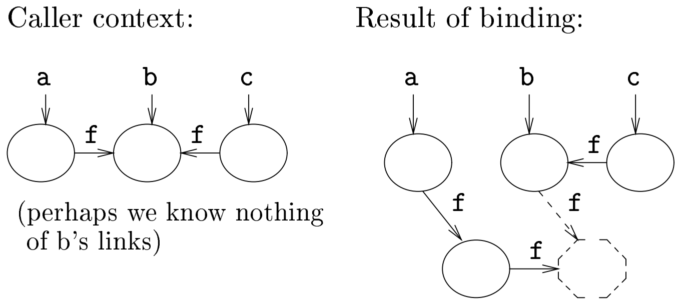
Consider this example and its analysis, based on Whaley and Rinard's algorithm:

```
class C { private C f;
            public void m(C x) {
               C y = new C();
               this.f = y;
               y.f = x.f; }
        }
```

Method m's analysis generates a "transfer function," which maps the input shape (the dotted parts) to an output shape:



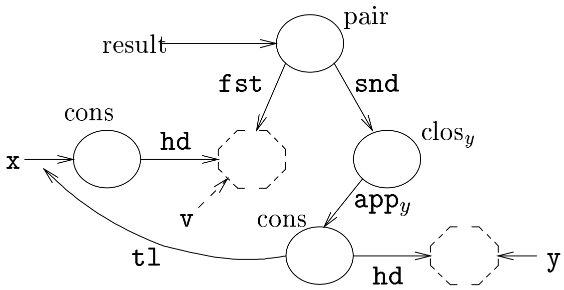
An invocation, like a.m(b), links the actual-parameter's shape to the formal's:



ML escape analysis

Escape analysis asks: Which bindings escape from an expression's result? Consider this ML-like expression and the shape constructed by its evaluation:

```
let x = v::nil
in  (hd x,  lambda y. y::x)
```



Which "slices" of the shape are of interest? The answer can be expressed as a set of *paths*, within the shape, from an expression's free variables to the result object. For the example, the path set reads

$$\begin{aligned}\mathcal{R} = \{&v.fst^{-1},\\ &x.tl^{-1}.app^{-1}_{[x \mapsto v.hd^{-1}]}.snd^{-1},\\ &y.hd^{-1}.app_y^{-1}.snd^{-1}\}\end{aligned}$$

For

$$\begin{aligned}\mathcal{R} = \{&v.fst^{-1},\\ &x.tl^{-1}.app^{-1}_{[x\mapsto v.hd^{-1}]}.snd^{-1},\\ &y.hd^{-1}.app^{-1}_{y}.snd^{-1}\}\end{aligned}$$

the app^{-1} acts as a "quantifier" to a path's prefix. When the expression is embedded within a context, e.g.,

$$(\texttt{snd}\ (\texttt{let x}\ = \ldots\ \texttt{in}\ \ldots))\ @\ \texttt{nil}$$

The resulting path set is computed as

$$\begin{aligned}(\mathcal{R}.snd).app_{nil} &= \{(v.fst^{-1}.snd).app_{nil},\\ &\quad (x.tl^{-1}.app^{-1}_{[x\mapsto v.hd^{-1}]}.snd^{-1}.snd).app_{nil},\\ &\quad (y.hd^{-1}.app^{-1}_{y}.snd^{-1}.snd).app_{nil}\}\\[1em] &= \{x.tl^{-1}.app^{-1}_{[x\mapsto v.hd^{-1}]}.app_{nil},\\ &\quad y.hd^{-1}.app^{-1}_{y}.app_{nil}\}\\[1em] &= \{(x.tl^{-1})[x \mapsto v.hd^{-1}],\\ &\quad (y.hd^{-1})[y \mapsto nil]\}\\[1em] &= \{v.hd^{-1}.tl^{-1}\}\end{aligned}$$

How should we formalize the "algebra" of shapes, slices, and linking?

A foundation: Bunched-implication (BI) logic of heap storage

Developed by Ishtiaq, O'Hearn, Pym, Reynolds, and Yang, BI-logic is a Hoare logic for reasoning about properties of 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 $a.snd = c$.

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

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

Here, w, $\mathtt{x}$, y, and $\mathtt{z}$ are free identifiers — auxiliary variables, or local variables whose bindings live in the stack, or "dangling pointers."

Intuitions about the new logical connectives:

- $p_1 * p_2$ 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 for every disjoint h' that makes p' hold, p also holds for $h' \cdot h$.
- $*$ and $\mathrel{-\!\!*}$ are categorical adjoints.

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

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

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

Operational semantics of statements

A configuration is a statement, stack, heap, triple: $S,\ s, h$.
Ordinary assignment, $\mathtt{x} := E$, updates s; field assignment, $E.i := E'$, updates h.

Here are the small-step transition rules; note that the semantics of expressions *depends only on the stack* and not the heap:

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

$$\frac{[\![E]\!]s = \ell \in \mathit{Location},\ h(\ell) = r}{\mathtt{x} :=\ E.i,\ s, h \Longrightarrow [s|\mathtt{x} \mapsto r.i], h}$$

$$\frac{[\![E]\!]s = \ell \in \mathit{Location},\ h(\ell) = r,\ [\![E']\!]s = v'}{E.i :=\ E',\ s, h \Longrightarrow s, [h|\ell \mapsto (r|i \mapsto v')]}$$

$$\frac{\ell \in \mathit{Location},\ \ell \notin \mathit{dom}(h),\ [\![E_1]\!]s = v_1,\ [\![E_2]\!]s = v_2}{\mathtt{x} :=\ \mathtt{new}\, C(E_1, E_2),\ s, h \Longrightarrow [s|\mathtt{x} \mapsto \ell], [h|(\ell \mapsto v_1, v_2)]}$$

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 $\{[\![E]\!]s\} = dom(h)$, $h([\![E]\!]s) = r$, $r.fst = [\![E_1]\!]s$, and $r.snd = [\![E_2]\!]s$
$s, h \models \mathbf{emp}$	iff $dom(h) = \{\}$
$s, h \models p * p'$	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 in a one-object heap at 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

Semantics of Hoare triples

It's the expected partial correctness:

$\{p\}S\{p'\}$ holds iff

$s, h \models p$ implies
(i) S, s, h is safe (never stuck/deadlocked), and
(ii) $S, s, h \Longrightarrow^* s', h'$ implies $s', h' \models p'$

Small axioms for assignments

An abbreviation: $E \doteq E'$ abbreviates $\quad E = E' \wedge \mathbf{emp}$.

These "small axioms" form the logic's core; assume that $\mathtt{x}$, *a*, *b*, and *c* are *variables* and that $\mathtt{x} \notin \{a, b, c\}$:

$$\{\mathtt{x} \doteq a\} \;\; \mathtt{x} := E \;\; \{\mathtt{x} \doteq E[a/x]\}$$

$$\{E \mapsto a, b\} \;\; E.\mathit{fst} := E' \;\; \{E \mapsto E', b\}$$

$$\{\mathtt{x} \doteq a\} \;\; \mathtt{x} := \mathtt{new}\, C(E_1, E_2) \;\; \{\mathtt{x} \mapsto E_1[a/\mathtt{x}], E_2[a/\mathtt{x}]\}$$

$$\{(E \mapsto a, b) \wedge \mathtt{x} = c\} \;\; \mathtt{x} := E.\mathit{fst} \;\; \{\mathtt{x} = a \wedge E[c/\mathtt{x}] \mapsto a, b\}$$

The axioms are "transfer-function-like," operating on mere *0- and 1-object heaps*, where expressions' values depend only on the stack.

Structural rules for inference

The small axioms gain utility when used with these structural rules; let $\mathit{modified}(S)$ be those variables that are targets of assignments in S.

$$Consequence: \quad \frac{p \supset p' \quad \{p'\}S\{q'\} \quad q' \supset q}{\{p\}S\{q\}}$$

$$AuxVarElim: \quad \frac{\{p\}S\{q\}}{\{\exists x.p\}S\{\exists x.q\}} \quad \text{where } x \notin free(S)$$

$$Subst: \quad \frac{\{p\}S\{q\}}{(\{p\}S\{q\})[E_1/x_1, \cdots E_k/x_k]} \quad \text{where } \{x_1, \cdots, x_k\} \supseteq \mathit{free}(p, S, q),$$

and $x_i \in \mathit{modified}(S)$ implies E_i is a var, $E_i \notin \mathit{free}(E_j), j \neq i$

$$Frame: \quad \frac{\{p\}S\{p'\}}{\{p * q\}S\{p' * q\}} \quad \text{where } \mathit{modified}(S) \cap \mathit{free}(q) = \{\}$$

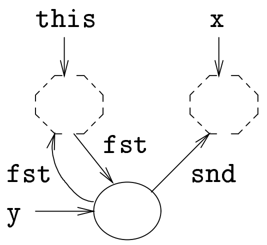
The second and third rules manipulate the free variables in the small axioms. The *Subst* rule motivates the usual rule for procedure invocation (as substitution of actuals for formals).

The *Frame* rule takes a result proved true of a heap "region" and embeds it into a larger heap, *justifying modular reasoning by disjoint heap regions*.

Application of BI-logic to shape analysis

Conventional escape-, alias- and shape-analyses are (abstract interpretations of) BI-logic derivations. Consider this shape analysis of `m(C x){...}`:

```
class C {
   C fst; C snd;
   function m(C x) {
      C y := new C(this,x);
      this.fst := y; }
}
```

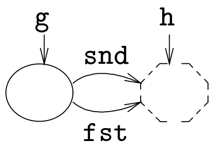


The shape analysis is neatly summarized as this Hoare triple:

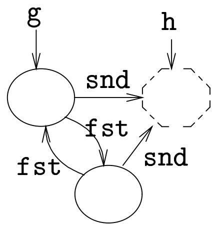
$$\begin{array}{l} \mathtt{m(C\ x)}\{\ \ \{\mathtt{this} \mapsto a, b\} \\ \qquad\qquad \mathtt{C\ y} := \ \mathtt{new\ C(this, x)};\ \ \mathtt{this.fst} :=\ \mathtt{y}; \\ \qquad\quad \{\exists \mathtt{y}.\ (\mathtt{this} \mapsto \mathtt{y}, b) * (\mathtt{y} \mapsto \mathtt{this}, \mathtt{x})\} \\ \qquad\ \} \end{array}$$

(The quantification of `y` is optional but appropriate.) Say there is an invocation, `g.m(h)`:

Caller context:



Result of binding:



The corresponding step in BI-logic is

$$\{\mathtt{g} \mapsto \mathtt{h}, \mathtt{h}\} \quad \mathtt{g.m(h)} \quad \{\exists \mathtt{y}.(\mathtt{g} \mapsto \mathtt{y}, \mathtt{h}) * (\mathtt{y} \mapsto \mathtt{g}, \mathtt{h})\}$$

because $[\mathtt{this/g},\ \mathtt{h/x}]$ by *Subst*.

Alias and escape analysis are reformatted BI-derivations. Recall the shape analysis of method, m:

$$
\begin{array}{l}
\mathtt{m(C\ x)\{}\ \ \{\mathtt{this} \mapsto a, b\} \\
\qquad\qquad \mathtt{C\ y := \ new\ C(this, x);\ \ this.fst := \ y;} \\
\qquad\quad \{\exists \mathtt{y}.\ (\mathtt{this} \mapsto \mathtt{y}, b) * (\mathtt{y} \mapsto \mathtt{this}, \mathtt{x})\} \\
\quad \}
\end{array}
$$

we can reformat the analysis as sets of alias paths to heap objects:

$$
\begin{array}{l}
\mathtt{m(C\ x)\{}\ \ \{\mathtt{this}\} \\
\qquad\qquad \mathtt{y := \ new\ C(this, x);\ \ this.fst := \ y;}_{\mathit{modifies}\ \mathtt{this.fst}} \\
\qquad\quad \{\exists \mathtt{y}.\ \{\mathtt{this},\ \mathtt{y}.\mathit{fst}\} * \{\mathtt{y},\ \mathtt{this}.\mathit{fst},\ \mathtt{x}.\mathit{snd}^{-1}\}\} \\
\quad \}
\end{array}
$$

The Hoare triple gives the aliases created on the two distinct objects; we can extract escape information from the triple also.

There exist many ad-hoc conventions for asserting noncircularity of pointers to "summary" objects in shape graphs; BI-logic gives us the convention for free. Consider,

```
x := this;
while (...) {
  x:= new C(..., x); // \alpha names the summary object
}
```

The loop's approximate invariant, mechanically computed by means of the usual lfp calculation with widening, reads

$$(\mathtt{x} = \alpha) \quad \wedge \quad (\alpha \mapsto ..., (\alpha^* \vee \mathtt{this}))$$

The "Kleene star" is actually the object-tensor operator; the expression abbreviates a sequence of *distinct objects*:

$$\begin{aligned} &\exists i \geq 0.\ (\mathtt{x} = \alpha_i) \\ &\quad \wedge \quad ((\alpha_i \mapsto ..., \alpha_{i-1}) * (\alpha_{i-1} \mapsto ..., \alpha_{i-2}) * \cdots * (\alpha_0 \mapsto ..., \mathtt{this})) \end{aligned}$$

What we are doing

We are trying to express existing pointer- and shape-analyses as (abstract interpretations of) BI-logics:

– Andersen's and Steengaard's pointer analyses
– forms of Whaley and Rinard's modular, multi-threaded shape analyses
– Blanchet's path-based escape analyses for ML and Java

We are trying to apply our understanding of polymorphic recursion to the *symbolic relational* variant of modular analysis, and we hope to establish connections to BI-logic.

A PhD student at LIX (Élodie-Jane Sims) is working jointly with Blanchet, Radhia Cousot, and me to develop the abstract intepretation machinery for abstracting BI-logics (predicate semantics, statement semantics, triple semantics) into escape- and shape-analyses.

Anindya Banerjee will spend the summer working with Peter O'Hearn's group in London.

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`

Martin Rinard's papers on escape analysis:
`http://www.cag.lcs.mit.edu/~rinard/paper`
`/pointerAndEscapeAnalysis.html`