Heaps denote finitely partitioned forests

Paul Blain Levy

University of Birmingham

July 13, 2018
Outline

1. Background
2. Language
3. The semantics
4. Computations and heaps
A reference is a memory address storing a value. It may be read or updated.

ML-like languages allow generation of references.
A reference is a memory address storing a value. It may be read or updated.

ML-like languages allow generation of references.

ML provides general references; they may store values of any type, even functions and thunks.

Sometimes we study languages that allow only ground references, i.e. references to integers or booleans.

Sometimes we study languages with full ground references, i.e. references to integers and also references to references to integers, but not references to code.
Some denotational models of higher-order languages with references:

- Game semantics of full ground references, using strong nominal sets (Laird; Murawski and Tzevelekos)
- Game semantics of “good” general references, using strong nominal sets (Laird; Murawski and Tzevelekos)
- Kripke semantics of full ground references, using ends and coends (Kammar, Levy, Moss and Staton)
Some denotational models of higher-order languages with references:

- Game semantics of full ground references, using strong nominal sets (Laird; Murawski and Tzevelekos)
- Game semantics of “good” general references, using strong nominal sets (Laird; Murawski and Tzevelekos)
- Kripke semantics of full ground references, using ends and coends (Kammar, Levy, Moss and Staton)

This talk is just about a first-order language with full ground references.

I hope this will eventually yield an improved account of these higher-order models.
We consider a first order language—no function types. The type syntax is:

\[ A ::= 0 \mid A + A \mid 1 \mid A \times A \mid \text{Ref} \mid X \mid \text{rec } X. A \]

A reference (memory location) has type \text{Ref} and stores a value of type

\[ D \overset{\text{def}}{=} \text{bool} \times \text{Ref} \times \text{Ref} + \text{nat} \]

Fancy version

Several sorts of references, e.g. red and blue.
A red reference has type \text{Ref}_{\text{red}} and stores a value of type \( D_{\text{red}} \).
A blue reference has type \text{Ref}_{\text{blue}} and stores a value of type \( D_{\text{blue}} \).
The calculus is “fine-grain call-by-value”.

The judgements are

- \( w, \Gamma \vdash^v V : A \) for values
- \( w, \Gamma \vdash^c M : A \) for computations.

Here \( w \) is a *world*, which is a number—the size of the memory. Or a finite sequence of sorts, if there are several sorts.
Here is the term syntax:

\[
\begin{align*}
V, W & ::= x \mid l \mid \text{in}_i V \mid \langle \rangle \mid \langle V, W \rangle \mid \text{roll } V \\
M, N & ::= \text{return } V \mid M \text{ to } x. M \\
& \quad \mid \text{match } V \text{ as } \ldots \\
& \quad \mid \text{if } V = W \text{ then } M \text{ else } N \\
& \quad \mid V ::= W. M \\
& \quad \mid \text{read } V \text{ as } x. M \\
& \quad \mid \text{new } x ::= V. M
\end{align*}
\]
For a computation $w \vdash^c M : A$ we have

$$w, M \Downarrow w', V$$

where $w' \equiv w$ and $w' \vdash^v V : A$. 
For a computation $w \vdash^c M : A$ we have

$$w, M \Downarrow w', V$$

where $w' \sqsupseteq w$ and $w' \vdash^v V : A$.

The observational preorder is obtained from programs of boolean type.

Reference non-use and swap:

$$M \sim \text{new } x := \overrightarrow{V}. \ M$$

$$\text{new } x := \overrightarrow{V}, y := \overrightarrow{W}, y' := \overrightarrow{W'}, z := \overrightarrow{V'}. \ M$$

$$\sim \text{new } x := \overrightarrow{V}, y' := \overrightarrow{W'}, y := \overrightarrow{W}, z := \overrightarrow{V'}. \ M$$
Fine-grain call-by-value is modelled by a distributive Freyd category.

- A category $\mathcal{C}$
- A category $\mathcal{K}$ with the same objects as $\mathcal{C}$
- An identity-on-objects functor $\iota : \mathcal{C} \to \mathcal{D}$
- Extra structure for the product and sum types.

A type denotes an object of $\mathcal{C}$.

A value $\Gamma \vdash^v V : A$ denotes a $\mathcal{C}$-morphism $[[\Gamma]] \to [[A]]$.

A computation $\Gamma \vdash^c M : A$ denotes a $\mathcal{K}$-morphism $[[\Gamma]] \to [[A]]$.

More categories are required for terms $w, \Gamma \vdash^v V : A$ and $w, \Gamma \vdash^c M : A$. 
Basic semantics of a computation (Levy 2002)

What is the meaning of a computation $\nu, \Gamma \vdash^c M : A$?
What is the meaning of a computation $w, \Gamma \vdash^c M : A$?

Given

- some local cells $\overrightarrow{I}$
- an environment $\rho$ for $\Gamma$, using cells in $w, \overrightarrow{I}$
- a state $s$ for $w, \overrightarrow{I}$

the computation terminates with

- some more local cells $\overrightarrow{I'}$
- a value $v$ of type $A$, using cells in $w, \overrightarrow{I}, \overrightarrow{I'}$
- a state $s'$ for $w, \overrightarrow{I}, \overrightarrow{I'}$

In summary $\llbracket M \rrbracket : \overrightarrow{I}, \rho, s \mapsto \overrightarrow{I'}, v, s'$. 
Equivariance

In the basic model, a computation denotes a function:

\[ [M] : \vec{I}, \rho, s \mapsto \vec{I}', v, s' \]

In the strong nominal set model and the end/coend model
- functions are identified up to reference non-use and swap
- functions are constrained to propagate reference non-use and swap.

Aim

To reformulate the equivariant model in an explicit way:
- the homset \( \mathcal{K}(A, B) \) will be of the form \( \prod_{x \in X} Y(x) \)
  
   “A computation denotes a function”.
- elements of \( X \) and \( Y(x) \) will not be equivalence classes.
Let **Inj** be the category of finite sets and injections.

Then $\text{Fam}(\text{Inj}^{\text{op}})$ is the category of template sets.

(Equivalent to strong nominal sets.)

A type denotes a template set, i.e. a family of finite sets.

### Templates for $\text{bool} \times \text{Ref} \times \text{Ref} + \text{nat}$

\begin{align*}
\text{inl} \langle b, -0, -0 \rangle & \quad (b \in \mathbb{B}) \quad \text{Arity} = 1 \\
\text{inl} \langle b, -0, -1 \rangle & \quad (b \in \mathbb{B}) \quad \text{Arity} = 2 \\
\text{inr} \ n & \quad (n \in \mathbb{N}) \quad \text{Arity} = 0
\end{align*}

The template set is indexed by $\mathbb{B} + \mathbb{B} + \mathbb{N}$. 
Let $[[\Gamma]] = (X_i)_{i \in I}$ and $[[A]] = (Y_j)_{j \in J}$.

Then $V$ denotes a map of template sets $(X_i)_{i \in I} \rightarrow (Y_j)_{j \in J}$.

For every template $i$, we obtain

- a template $j$
- a map of references $Y_j \rightarrow X_i$.

Reason: every reference in $V[W_x/x]_{x \in \Gamma}$ arises from one in $(W_x)_{x \in \Gamma}$. 
If \([A] = (X_i)_{i \in I}\) and \([B] = (Y_j)_{j \in J}\),
a template for \(A \times B\) consists of

- a template \(i\) for \(A\)
- a template \(j\) for \(B\)
- a matching—i.e., equivalence relation \(R\) on \(A + B\),
  discrete on each component

and the arity of \((i, j, R)\) is \(A + B \setminus R\).
Indistinguishable inputs to $x : \text{bool} + \text{Ref} \times \text{Ref} \vdash^c M : A$

\begin{align*}
\text{l}_0, \text{l}_1, \text{l}_2, \text{l}_3 & \\
\text{x} & \mapsto \text{inr} \langle \text{l}_1, \text{l}_2 \rangle \\
\text{l}_0 & := \text{inr} 17 \\
\text{l}_1 & := \text{inl} \langle \text{true}, \text{l}_1, \text{l}_1 \rangle \\
\text{l}_2 & := \text{inl} \langle \text{false}, \text{l}_0, \text{l}_2 \rangle \\
\text{l}_0, \text{l}_1, \text{l}_2, \text{l}_3, \text{l}_4 & \\
\text{x} & \mapsto \text{inr} \langle \text{l}_4, \text{l}_0 \rangle \\
\text{l}_0 & := \text{inl} \langle \text{false}, \text{l}_2, \text{l}_0 \rangle \\
\text{l}_1 & := \text{inl} \langle \text{false}, \text{l}_0, \text{l}_0 \rangle \\
\text{l}_2 & := \text{inr} 17 \\
\text{l}_3 & := \text{inr} 5 \\
\text{l}_4 & := \text{inl} \text{true}, \text{l}_4, \text{l}_4
\end{align*}
A reference stores a value of type $D$, and $[D] = (C_k)_{k \in K}$.
A reference stores a value of type $D$, and $[D] = (C_k)_{k \in K}$.

For each element $x \in X_i$ we need a template $r(x) \in K$.

For each $c_0 \in C_{r(x)}$ we need a template $r(x, c_0) \in K$.

For each $c_1 \in C_{r(x, c)}$ we need a template $r(x, c_0, c_1) \in K$.

Etc.
A reference stores a value of type $D$, and $[D] = (C_k)_{k \in K}$.

For each element $x \in X_i$ we need a template $r(x) \in K$.

For each $c_0 \in C_{r(x)}$ we need a template $r(x, c_0) \in K$.

For each $c_1 \in C_{r(x, c)}$ we need a template $r(x, c_0, c_1) \in K$.

Etc.

In summary, we need a (non-well-founded) forest with $X_i$ roots, where each node is labelled with $k \in K$ and has $C_k$ children.
Heap = finitely partitioned tree

A reference stores a value of type $D$, and $[D] = (C_k)_{k \in K}$.

For each element $x \in X_i$ we need a template $r(x) \in K$.

For each $c_0 \in C_{r(x)}$ we need a template $r(x, c_0) \in K$.

For each $c_1 \in C_{r(x, c)}$ we need a template $r(x, c_0, c_1) \in K$.

Etc.

In summary, we need a (non-well-founded) forest with $X_i$ roots, where each node is labelled with $k \in K$ and has $C_k$ children.

This gives a family of finite sets, one at each node. Each node is a reference. Reference equality is observable, so the nodes are partitioned into finitely many parts.

In summary, a heap is a finitely partitioned forest.
A computation $\Gamma \vdash^c \mu : \mathcal{A}$ denotes a function taking a template (environment) and finitely partitioned forest (heap) to a template (value) and finitely partitioned forest (heap). No equivariance constraint or identification are used, as a finitely partitioned forest gives the observable data about a heap and does not mention unreachable references or reference order. We thus obtain a Freyd category on $\text{Fam}(\text{Inj}^{\text{op}})$. 