Compositional Memory Management in the λ-calculus

Or: A Compositional Semantics for Explicit Naming

Sky Wilshaw, Graham Hutton

School of Computer Science, University of Nottingham

12th June 2025

Explicit naming

$$let\ x = 1 + 2\ in\ \underbrace{print\ (x + x)}_{\text{scope of } x}$$

Explicit naming

$$let\ x = 1 + 2\ in\ \underbrace{print\ (x + x)}_{\text{scope of } x}$$

With *explicit naming*, we use explicit operations to manipulate names:

$$bind\ x\ to\ 1 + 2\ in\ print\ (read\ x + read\ x);\ free\ x$$

Names are first-class citizens:

$$bind\ x\ to\ 4\ in\ x \quad \text{returns } x\text{, not } 4$$

Names as pointers

In explicit naming, names are like pointers

- *bind* x *to* 7 allocates memory to hold the value 7
- *read* x dereferences the pointer x
- *free* x deallocates the memory pointed to by x

Important: The value bound to a name cannot change
(so names are like ‘immutable pointers’)

Explicit naming is a fragment of manual memory management

Examples

We track bindings using a *heap*, mapping names to values

The heap is updated during every computation step

	heap	expression	
	$\{\}$	$\underline{\mathit{bind}\ x\ \mathit{to}\ 7}\ \mathit{in}\ \mathit{print}\ (\mathit{read}\ x);\ \mathit{free}\ x$	
$\rightsquigarrow$	$\{x \mapsto 7\}$	$\mathit{print}\ (\underline{\mathit{read}\ x});\ \mathit{free}\ x$	
$\rightsquigarrow$	$\{x \mapsto 7\}$	$\underline{\mathit{print}\ 7};\ \mathit{free}\ x$	7 is printed
$\rightsquigarrow$	$\{x \mapsto 7\}$	$\underline{\mathit{free}\ x}$	
$\rightsquigarrow$	$\{\}$		

Examples

What if we free x before reading from it?

	heap	*expression*	
	$\{\}$	$\underline{\textit{bind } x \textit{ to } 7} \textit{ in free } x;\ \textit{print } (\textit{read } x)$	
$\rightsquigarrow$	$\{x \mapsto 7\}$	$\underline{\textit{free } x};\ \textit{print } (\textit{read } x)$	
$\rightsquigarrow$	$\{\}$	$\textit{print } (\underline{\textit{read } x})$	can't continue!

Behaviour is very sensitive to order of evaluation

The heap is threaded through the computation, so the semantics is *non-compositional*

How can we fix this?

Currently, our evaluator is a partial function of type

$$\text{Expr} \to \text{Heap} \rightharpoonup \text{Heap} \times \text{Value}$$

We can write this as

$$\text{Expr} \to T\,\text{Value} \quad \text{where} \quad T = \text{Heap} \rightharpoonup \text{Heap} \times (-)$$

Can we replace T with a better (more compositional) monad?

How can we fix this?

Currently, our evaluator is a partial function of type

$$\mathrm{Expr} \to \mathrm{Heap} \rightharpoonup \mathrm{Heap} \times \mathrm{Value}$$

We can write this as

$$\mathrm{Expr} \to T\ \mathrm{Value} \quad \text{where} \quad T = \mathrm{Heap} \rightharpoonup \mathrm{Heap} \times (-)$$

Can we replace T with a better (more compositional) monad?

$$U = \underbrace{\mathrm{Context}}_{\text{like a fixed heap}} \rightharpoonup \underbrace{(\mathrm{Heap} \rightharpoonup \mathrm{Heap})}_{\text{effect on the heap}} \times (-)$$

Crucially: The context isn't modified by heap effects, and effects are composed using their monoid structure.

This semantics is **equivalent** to the stateful semantics!

Examples

$$\mathrm{eff}_\Gamma(\mathit{read}\ x)(H) = \begin{cases} H & \text{if } x \in \operatorname{dom} H \\ \uparrow & \text{otherwise} \end{cases}$$

$$\mathrm{eff}_\Gamma(\mathit{free}\ x)(H) = \begin{cases} H' & \text{if } H = H', x \mapsto v \\ \uparrow & \text{otherwise} \end{cases}$$

$$\mathrm{eff}_\Gamma(e_1;\ e_2) = \mathrm{eff}_\Gamma(e_2) \circ \mathrm{eff}_\Gamma(e_1)$$

The same context Γ is used for both e_1 and e_2

Conclusions

- In *explicit naming*, we manipulate names manually using explicit operations
- Explicit naming can be viewed as a fragment of manual memory management
- The evaluator can be thought of as a monadic function $\mathrm{Expr} \to T\ \mathrm{Value}$
- By replacing T with a 'better' monad U we reduce dependence on state
- Paper coming soon!

Thank you!

Equivalence theorem

$$H : e \Downarrow H' : v \iff$$
$$(\exists w\ f,\ \overline{w} = v \ \wedge\ f(H) = H' \ \wedge\ \mathrm{tr}(H) \vdash e \Downarrow f : w)$$

$$\frac{}{H : x \Downarrow H : x}\ \textsc{H-Var} \qquad \frac{}{H : \lambda x.\, e \Downarrow H : \lambda x.\, e}\ \textsc{H-Lam}$$

$$\frac{H_1 : e_1 \Downarrow H_2 : \lambda x.\, e \qquad H_2 : e_2 \Downarrow H_3 : v \qquad (H_3, x \mapsto v) : e \Downarrow H_4 : v'}{H_1 : e_1\ e_2 \Downarrow H_4 : v'}\ \textsc{H-App}$$

$$\frac{H_1 : e \Downarrow (H_2, x \mapsto v) : x}{H_1 : {*e} \Downarrow (H_2, x \mapsto v) : v}\ \textsc{H-Read}$$

$$\frac{H_1 : e_1 \Downarrow H_2 : v \qquad H_2 : e_2 \Downarrow (H_3, x \mapsto v') : x}{H_1 : e_1; \mathit{free}\ e_2 \Downarrow H_3 : v}\ \textsc{H-Free}$$

$$\frac{\Gamma(x) = w}{\Gamma \vdash x \Downarrow \mathsf{id} : (x \mapsto w)}\ \textsc{E-Var} \qquad \frac{x \notin \operatorname{dom} \Gamma}{\Gamma \vdash \lambda x.\, e \Downarrow \mathsf{id} : \lambda^{\Gamma} x.\, e}\ \textsc{E-Lam}$$

$$\frac{\Gamma \vdash e_1 \Downarrow f_1 : \lambda^{\Gamma'} x.\, e \qquad \Gamma \vdash e_2 \Downarrow f_2 : w \qquad (\Gamma', x \mapsto w) \vdash e \Downarrow f_3 : w'}{\Gamma \vdash e_1\ e_2 \Downarrow f_3 \circ f_2 \circ f_1 : w'}\ \textsc{E-App}$$

$$\frac{\Gamma \vdash e \Downarrow f : (x \mapsto w)}{\Gamma \vdash {*e} \Downarrow \mathsf{read}\ x \circ f : w}\ \textsc{E-Read}$$

$$\frac{\Gamma \vdash e_1 \Downarrow f_1 : w \qquad \Gamma \vdash e_2 \Downarrow f_2 : (x \mapsto w')}{\Gamma \vdash e_1; \mathit{free}\ e_2 \Downarrow \mathsf{free}\ x \circ f_2 \circ f_1 : w}\ \textsc{E-Free}$$