Compositional memory management in the λ-calculus

Sky Wilshaw[1] and Graham Hutton[2]

[1] School of Computer Science, University of Nottingham
sky.wilshaw@nottingham.ac.uk
[2] School of Computer Science, University of Nottingham
graham.hutton@nottingham.ac.uk

1 Motivation and overview

Type systems can be used in programming languages to guarantee memory safety. This idea has been developed in many different directions, for example with *borrowing* from the Rust language [6, 3], the *reachability types* of Bao et al. [1], and the *fractional uniqueness* types of Marshall and Orchard [5]. In order to create new type systems of this form in a principled way, we would like a simple untyped system that supports a basic form of memory management, so that type systems can be built on top of it. However, existing untyped languages of this kind typically suffer from two major problems. First, they often introduce many new primitives for concepts such as memory cells, allocation, freeing, pointers, and stores, complicating the semantics. Second, they are usually non-compositional, as we need to thread the state of the memory store through all derivations; this is a particular weakness when our goal is to design type systems.

In this work, we identify a fragment of manual memory management, which we call *explicit naming*. Under this paradigm, names are first-class citizens in a language, with explicit primitives for creating, using, and freeing names. These primitives correspond to simple operations on memory, interpreting names as pointers to the values they are bound to. We will introduce a lambda calculus with explicit naming, with a stateful semantics that keeps track of the value assigned to each name. Then, to address the lack of compositionality, we will develop a compositional semantics for the same language, and show that it is equivalent to its non-compositional counterpart. This language, with its two equivalent semantics, provides a framework that can be used to develop type systems for explicit naming.

2 Modifying the lambda calculus

We will start with a simple call-by-value lambda calculus, with standard reduction rules and notation inspired by Launchbury's semantics for lazy evaluation [4]. We use the standard syntax for lambda terms, writing expressions with the letter e and values with the letter v. Judgments are of the form $H_1 : e \Downarrow H_2 : v$, where H_1 and H_2 are *heaps*, finite partial functions from names to values. This can be read as 'expression e can be evaluated with heap H_1 to produce a value v and a resulting heap H_2'.

Our initial rules are as follows. Problems relating to name collisions and α-equivalence will not be discussed here in order to focus on the key ideas of this contribution. Colours are not syntactically important.

$$\frac{}{(H, x \mapsto v) : x \Downarrow (H, x \mapsto v) : v}\ \textsc{Var}' \qquad \frac{}{H : \lambda x.\, e \Downarrow H : \lambda x.\, e}\ \textsc{Lam}$$

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

Here, the variable rule Var′ tells us that a name x evaluates to the value v bound to it in the heap. This means that every *mention* of a name is a *use* of it, or alternatively, that we cannot have a pointer without dereferencing it. This prevents us from using names as first-class citizens. Therefore, to make a lambda calculus with explicit naming, we need to divide the variable rule into the following two rules:

$$\frac{}{H : x \Downarrow H : x}\ \textsc{Var} \qquad \frac{H_1 : e \Downarrow (H_2, x \mapsto v) : x}{H_1 : !e \Downarrow (H_2, x \mapsto v) : v}\ \textsc{Read}$$

We have introduced a new operator, written '!', for looking up the value bound to a name. With these new rules, names are already values, and do not reduce; a name must be explicitly dereferenced using '!' in order to access its value.

To model explicit naming, it remains to provide a method to free names. This allows us to model one of the key challenges with memory management, namely, that dereferences might fail after a deallocation. To do this, we add in a second new primitive, which evaluates an expression and then frees a name (which itself may be the result of evaluating some other expression).

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

3 Recovering compositionality

The main disadvantage of the presented semantics is its non-compositionality, as the heap state must be threaded through each derivation. This prevents us from building type systems on this language. To combat this, we will describe a new semantics which calculates the *denotation* of an expression separately from computing the *effect* on the heap of evaluating this expression. Crucially, we do not need to compute intermediate heap states in order to calculate denotations or effects of compound expressions; it is in this sense that our effect-based semantics is compositional.

In order to do this, we need to reduce our dependence on the heap state, by moving some of the information from the heap into the values. In particular, abstractions now evaluate to closures, and variables now keep track of their assigned values. These new values are called *denotational values* and written with the letter w. Rather than in a heap, denotational values are stored in a *context* Γ, a finite partial function from names to denotational values.

To be precise, a denotational value is either a closure of the form $(\lambda^{\Gamma} x.\, e)$ or a variable binding $(x \mapsto w)$. Such a denotational value can be thought of as a value in the usual sense, together with all of the data that it could 'see' in the heap at the time of its creation. We define denotational values coinductively to allow for loops in variable bindings and stored contexts.

Our big-step reduction relation will now have the form $\Gamma \vdash e \Downarrow f : w$, where f is the *effect*, a partial function from heaps to heaps, and w is a denotational value. We may view f as the effect on the heap of evaluating $\Gamma \vdash e \Downarrow w$. Such partial functions evidently form a monoid under composition.

$$\frac{}{(\Gamma, x \mapsto w) \vdash x \Downarrow \mathrm{id} : (x \mapsto w)}\ \textsc{F-Var} \qquad \frac{}{\Gamma \vdash \lambda x.\, e \Downarrow \mathrm{id} : \lambda^{\Gamma} x.\, e}\ \textsc{F-Lam}$$

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

$$\frac{\Gamma \vdash e \Downarrow f : (x \mapsto w)}{\Gamma \vdash !e \Downarrow \mathrm{read}\, x \circ f : w}\ \textsc{F-Read} \qquad \frac{\Gamma \vdash e_1 \Downarrow f_1 : w \qquad \Gamma \vdash e_2 \Downarrow f_2 : (x \mapsto w')}{\Gamma \vdash e_1; \mathrm{free}\ e_2 \Downarrow \mathrm{free}\, x \circ f_2 \circ f_1 : w}\ \textsc{F-Free}$$

Here, $(x := w)$, $\mathrm{read}\, x$, $\mathrm{free}\, x$ denote partial functions that perform the given operation. For instance, $(x := w)$ adds the binding $x \mapsto w$ to any heap where x is not in its domain, and is undefined elsewhere. Similarly, $\mathrm{read}\, x$ is the identity on heaps that contain x in their domain, and undefined elsewhere. In these rules, effects of subexpressions are only used to compute the composite effect of the expression, and are not applied to contexts or used serially between subexpressions. We do not use the full flexibility of partial functions between heaps, and various simpler presentations of our effects are possible, for example using *effect quantales* [2].

4 Equivalence

We can show that the two semantics are equivalent. To formalise this equivalence, we need to set up some conversions between heaps and contexts. First, we can 'forget' the extra data of a denotational value w to recover a value $v = \overline{w}$. This operation is defined by $\overline{(\lambda^{\Gamma} x.\, e)} = \lambda x.\, e$ and $\overline{(x \mapsto w)} = x$. In the other direction, we can translate a heap H into a context $\Gamma = \mathrm{tr}(H)$ defined by the following coinductive rules.

$$\frac{H(x) = y \qquad \mathrm{tr}(H)(y) = w}{\mathrm{tr}(H)(x) = (y \mapsto w)} \qquad \frac{H(x) = \lambda y.\, e}{\mathrm{tr}(H)(x) = \lambda^{\mathrm{tr}(H)} y.\, e}$$

We can then prove the following result.

Theorem. *Let e be an expression and H be a heap. We define inductively that a variable x is* reachable *from e in H if it is free in e or is reachable from $H(y)$ where y is free in e. If all variables reachable from e in H occur in the domain of H, then*

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

where the derivations on the left and right sides of this equivalence pull from the same stream of fresh names.

The hypothesis of this theorem is a minor generalisation of the usual notion of an 'expression-in-context', ensuring that $\mathrm{tr}(H)$ contains all of the free variables of e.

This equivalence theorem allows for more compositional reasoning about memory-sensitive computation. As an example, we can show that if two expressions e_1 and e_2 have disjoint sets of reachable names in H, then they can be evaluated in either order without affecting the overall computation. This is because the effects f_1 and f_2 of evaluating these expressions in $\mathrm{tr}(H)$ operate on disjoint sets of names and therefore commute past one another: $f_1 \circ f_2 = f_2 \circ f_1$.

We hope that this compositional framework is suitable for developing interesting type theories that are capable of encapsulating some of the challenges of memory management.

References

[1] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–32, 2021.

[2] Colin S. Gordon. Polymorphic iterable sequential effect systems. *ACM Trans. Program. Lang. Syst.*, 43(1), April 2021.

[3] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[4] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 144–154, New York, NY, USA, 1993. Association for Computing Machinery.

[5] Danielle Marshall and Dominic Orchard. Functional ownership through fractional uniqueness. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024.

[6] Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, October 2014.