A Compositional Semantics for Explicit Naming

SKY WILSHAW, School of Computer Science, University of Nottingham, UK
GRAHAM HUTTON, School of Computer Science, University of Nottingham, UK

Naming enables values to be shared between different parts of a computation. We study a notion of *explicit naming*, where names are first-class citizens, and explicit primitives are provided for creating, using and freeing names. Operationally, such primitives provide a form of manual memory management using pointers. Using this interpretation, we can define a simple semantics for explicit naming by threading through a heap that maps names to values. However, this leads to a non-compositional semantics, which complicates inductive reasoning. To address this, we introduce a lambda calculus with explicit naming, and develop a compositional semantics for it that is provably equivalent to its non-compositional counterpart.

1 Introduction

One of the most basic operations in programming is to bind a value to a name. Often, no distinction is made between a name and its value. For example, in the body of the term

$$\mathit{let}\ \ x = 1 + 2\ \ \mathit{in}\ \ \mathit{print}\ (x + x)$$

we might understand the name x as *being* the value 3, but outside the body of the term the name loses this meaning as it is out of scope. In this article, we consider a notion of *explicit naming*, where names are first-class citizens in a language, and explicit operations are provided by the language to bind a value to a name, to look up the value bound to a name, and to free a name when it is no longer needed. For example, in this paradigm, in the body of

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

the name x might be understood as a *reference* to the value 3, and we use explicit operations to read the value of x, and to free the name when the scope of the body ends. We focus on immutable bindings, where once a value is assigned to a name it cannot be changed.

A key aspect of explicit naming is that a name can escape its scope of definition, as it can be treated just like any other first-class citizen. In particular, names can be returned as results, passed as arguments, and stored in data structures. Because of this, the operation of freeing a name can affect other parts of the program that later use it. For example, attempting to use a name that has been freed may lead to a program crashing or having undefined behaviour. This kind of 'action at a distance' is a primary source of complexity when reasoning about explicit naming.

Explicit naming can naturally be viewed as a form of manual memory management, where names correspond to pointers. In particular, binding a value to a name corresponds to allocating memory, reading the value associated to a name corresponds to dereferencing a pointer, and freeing a name corresponds to deallocating memory. This analogy motivates a direct way to give a semantics for explicit naming, by threading through a heap that maps names to values, with the heap being updated whenever a fresh name is introduced or an unwanted name is freed.

With a threaded-heap semantics, it is important that the operations on the heap are executed in a particular sequence. This means that the semantics is not compositional at the level of heaps, as the result of evaluating a complex term with a given heap is not determined by the results of evaluating its subterms with the same heap. For example, to evaluate $e_1 + e_2$, we might first evaluate e_1 with the current heap, then evaluate e_2 with the heap obtained after evaluating e_1. This lack of compositionality complicates inductive reasoning about such a semantics.

Authors' Contact Information: Sky Wilshaw, School of Computer Science, University of Nottingham, UK, sky.wilshaw@nottingham.ac.uk; Graham Hutton, School of Computer Science, University of Nottingham, UK, graham.hutton@nottingham.ac.uk.

However, this non-compositionality is not inherent. In this article, we show how to define a compositional semantics for explicit naming. In particular, we present a lambda calculus with explicit naming, and give a new semantics based on tracking effects. This new semantics is compositional at the level of heaps, in the sense that the heap is not threaded through each computation. This means that subexpressions in a larger program can now be analysed independently. For example, when evaluating $e_1 + e_2$, the subexpressions e_1 and e_2 are now evaluated in the same context. Crucially, this semantics is equivalent to the non-compositional heap-based semantics, and we can transfer useful properties across this equivalence. Our choice to use a minimal language such as the lambda calculus is inessential, but was chosen in order to focus on the essence of explicit naming. The most important property of the lambda calculus that we use is that values are immutable, and our ideas are more generally applicable in other languages with this property.

Concretely, we make the following contributions:

- We introduce a lambda calculus with explicit naming, define a heap-based semantics for the language, and provide examples of reasoning using this semantics (section 2);
- We present another semantics for the same language, which exploits effect tracking to ensure the semantics is compositional at the level of heaps (section 3);
- We state the equivalence of the two semantics, and show how to transfer results across the equivalence, including specific examples and general transformations (section 4);
- We present a proof of this equivalence, by showing that both semantics are equivalent to an intermediate semantics based on a 'clairvoyant heap' (section 5).

We discuss related work in section 6 and future work in section 7. To make the article accessible to a broad audience, we assume only a basic knowledge of programming language semantics (evaluation semantics, big-step operational semantics, rule induction), lambda calculus (call-by-value semantics, closures), and order theory (partial orders, semilattices).

2 A Lambda Calculus with Explicit Naming

In this section, we introduce a lambda calculus with explicit naming. In this language, a name can be thought of as a pointer to a value, with explicit operations being provided for creating, using and freeing names. The section concludes with some examples of reasoning to demonstrate various properties of the heap-based semantics we define for the language.

2.1 Heap-Based Semantics

Let us begin with a simple call-by-value lambda calculus without explicit naming, whose syntax is specified by the following grammar, where x ranges over an infinite set of (variable) names:

$$e \coloneqq x \mid \lambda x.e \mid e\ e$$

To define the semantics for the language, we use an approach and notation inspired by Launchbury [1993]. In particular, we use a *heap* to keep track of the assignments to names, which is given by a partial function from names to values. For now, an expression is a *value* if it is a lambda abstraction. We write $\{\}$ for the empty heap, and $(H, x \mapsto v)$ for the extension of a heap with a new binding; for ease of identification, heaps are grey and bindings are red.

Judgments in our semantics are of the form $H_1 : e \Downarrow H_2 : v$, where H_1 and H_2 are heaps, e is an expression, and v is a value. This can be read as 'the expression e can be evaluated with initial heap H_1 to produce the value v and final heap H_2'. The semantics is defined by the following rules:

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

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

The variable rule Var specifies that a name x evaluates to the value v bound to it in the heap. The Lam rule specifies that a lambda abstraction $\lambda x.\, e$ is already fully evaluated. The App rule states that an application $e_1\ e_2$ proceeds by first evaluating e_1 to an abstraction $\lambda x.\, e$, then evaluating e_2 to a value v, and finally evaluating the body e of the abstraction with a new binding $x \mapsto v$ on the heap. This new binding will persist even after the expression has finished being evaluated, as we have not yet introduced a mechanism for freeing names. Note that the heap is threaded sequentially through this rule, with the final heap in each premise being used as the initial heap in the next.

We adopt the convention that the name x in the App rule is chosen to be different from all names used so far, to avoid unintended name collisions on the heap. As such, we can think of the App rule as a mechanism for creating fresh names. We discuss fresh names in more detail in section 5.

2.2 First-Class Names

The above version of the Var rule means that every *mention* of a name is a *use* of it, preventing us from using names as first-class citizens. Therefore, to make a language with explicit naming, we split the variable rule into two rules, to distinguish a name from its value:

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

The new Var rule has a transparent reading, 'names are values', and we extend the notion of values accordingly. Given this rule, names are now first-class citizens that can be passed as arguments, returned as results, and stored in the bodies of abstractions for later use. The Read rule introduces a new operator, written '$*$', for reading the value bound to a name. This rule operates on an arbitrary expression that evaluates to a name, not just an expression that is syntactically a name.

Example 2.1 (reading variables). In our new semantics, the expression $*x$ ('read the value bound to x') has the same behaviour as x does in the usual lambda calculus. More explicitly, in the heap $(H, x \mapsto v)$, we can show that $*x$ evaluates to v without altering the heap:

$$\frac{\dfrac{}{(H, x \mapsto v) : x \Downarrow (H, x \mapsto v) : x}\ \textsc{Var}}{(H, x \mapsto v) : {*x} \Downarrow (H, x \mapsto v) : v}\ \textsc{Read}$$

◇

Example 2.2 (identity function). In this semantics, the expression $\lambda x.\, x$ is no longer the identity function. Consider the expression $(\lambda x.\, x)\ (1+2)$, assuming that we have incorporated numbers into our language. We will describe the evaluation of this expression using an informal 'computation trace', underlining the expression under reduction.

	heap	*expression*
	$\{\}$	$(\lambda x.\, x)\ \underline{(1+2)}$
$\rightsquigarrow$	$\{\}$	$\underline{(\lambda x.\, x)\ 3}$
$\rightsquigarrow$	$\{x \mapsto 3\}$	x

The result is that 3 is bound in the heap to the variable x, and the name x itself is returned. This shows that $\lambda x.\, x$ is not the identity function. However, $\lambda x.\, {*x}$ does behave as the identity:

$$\begin{array}{lll} & \{\} & (\lambda x.\, {*x})\ \underline{(1+2)} \\ \rightsquigarrow & \{\} & \underline{(\lambda x.\, {*x})\ 3} \\ \rightsquigarrow & \{x \mapsto 3\} & \underline{*x} \\ \rightsquigarrow & \{x \mapsto 3\} & 3 \end{array}$$

In general, regular lambda terms can be converted for this semantics by replacing every occurrence of a variable x with $*x$ where this is syntactically valid. For example, the Church numeral 2, which usually takes the form $\lambda x.\, \lambda y.\, x\ (x\ y)$, would be written as $\lambda x.\, \lambda y.\, {*x}\ ({*x}\ {*y})$. ◇

2.3 Freeing Names

To free a name when it is no longer needed, we would ideally like to have an expression '*free x*' that simply frees the name x from the heap. However, we must also decide what value such an expression should produce. Returning some form of dummy value would be rather cumbersome, so instead, we use an operator of form 'e_1; *free* e_2'. This expression first evaluates e_1 to a value v, then frees the name that e_2 evaluates to, and finally returns v:

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

Note that this rule can only be applied if the heap contains a binding for the name being freed. In particular, we cannot free a name that has not been allocated on the heap, and we cannot free a name multiple times. These two cases are usually considered undefined behaviour in manual memory management systems, which we reflect here by disallowing such behaviour. Our choice for the form of the freeing operator is motivated by the following example.

Example 2.3 (self-cleaning identity). Previously, we saw that evaluating $(\lambda x.\, {*x})\ (1+2)$ with the empty heap resulted in the value 3, but had the side effect of 'polluting' the heap with the binding $x \mapsto 3$. To avoid this, we can free x before returning from the function:

$$\begin{array}{lll} & \{\} & (\lambda x.\, ({*x};\ \mathit{free}\ x))\ \underline{(1+2)} \\ \rightsquigarrow & \{\} & \underline{(\lambda x.\, ({*x};\ \mathit{free}\ x))\ 3} \\ \rightsquigarrow & \{x \mapsto 3\} & \underline{*x};\ \mathit{free}\ x \\ \rightsquigarrow & \{x \mapsto 3\} & \underline{3;\ \mathit{free}\ x} \\ \rightsquigarrow & \{\} & 3 \end{array}$$

In this manner, $\lambda x.\, ({*x};\ \mathit{free}\ x)$ is the identity function that 'cleans up after itself' by freeing the name x once it is no longer needed, returning the heap to its original state. This example shows how '$-$; *free x*' adds a freeing operation to a function without altering its returned value. ◇

The resulting lambda calculus, with rules Var, Lam, App, Read and Free, has explicit naming. This is a minimally complete set of rules, in the sense that none of the rules can be removed while preserving the desired naming features. This minimality was our motivation for choosing this particular language, in order to focus on the essence of explicit naming.

2.4 Reasoning

We now show some examples of reasoning using our language. We assume whenever it is convenient that bound variables have never been used before. Our first example is straightforward using the threaded-heap semantics, but the second example shows something that is difficult to prove.

Example 2.4 (immutability). Bindings in our language are *immutable*: once a value is assigned to a name, the value cannot be changed. This property holds because the only way to assign a value to a name is to bind that value to a fresh name in the App rule. To see how immutability can help with reasoning, consider the following example, where e is an unknown expression:

$$(\lambda y.\, {*x})\ e$$

Evaluating this example in a heap where x has the value 4 proceeds as follows:

$$\begin{array}{lll} & \{x \mapsto 4\} & (\lambda y.\, {*x})\ \underline{e} \\ \rightsquigarrow & \cdots & \\ \rightsquigarrow & H & \underline{(\lambda y.\, {*x})\ v} \\ \rightsquigarrow & H, y \mapsto v & \underline{*x} \end{array}$$

Here H is the heap obtained after evaluating e to the value v. If x is not in the domain of H, then we cannot complete the derivation. This occurs if x is freed by e. However, if x does occur in the domain of H, then we know by immutability of bindings that the value bound to x must be 4. So regardless of what e actually is, the example either evaluates to 4, or does not evaluate. ◇

Example 2.5 (reordering computations). Consider the expressions

$$e_1 + e_2 \quad \text{and} \quad e_2 + e_1$$

Under the normal left-to-right evaluation order for addition, the first expression will evaluate e_1 before e_2, while the second expression will evaluate e_2 before e_1. As a result, in general the two expressions do not have the same behaviour. For example, consider:

$${*x} + (1;\ \mathit{free}\ x) \quad \text{and} \quad (1;\ \mathit{free}\ x) + {*x}$$

The first expression will produce a value if x is bound to a number in the heap, but the second expression can never evaluate because it frees x and then attempts to read from it. However, it can be shown that if the two expressions $e_1 + e_2$ and $e_2 + e_1$ both evaluate in some initial heap, then in fact they evaluate to the same value. A simple example is given by the following expressions:

$$({*x};\ \mathit{free}\ x) + ({*y};\ \mathit{free}\ y) \quad \text{and} \quad ({*y};\ \mathit{free}\ y) + ({*x};\ \mathit{free}\ x)$$

Using an initial heap that binds x and y to numbers, the first expression evaluates as follows:

$$\begin{array}{lll} & \{x \mapsto 1, y \mapsto 2\} & (\underline{*x};\ \mathit{free}\ x) + ({*y};\ \mathit{free}\ y) \\ \rightsquigarrow & \{x \mapsto 1, y \mapsto 2\} & (\underline{1;\ \mathit{free}\ x)} + ({*y};\ \mathit{free}\ y) \\ \rightsquigarrow & \{y \mapsto 2\} & 1 + (\underline{*y};\ \mathit{free}\ y) \\ \rightsquigarrow & \{y \mapsto 2\} & 1 + (\underline{2;\ \mathit{free}\ y)} \\ \rightsquigarrow & \{\} & \underline{1 + 2} \\ \rightsquigarrow & \{\} & 3 \end{array}$$

Using the same initial heap, the second expression evaluates to the same final heap and value, but the derivations have no common intermediate states:

$$\begin{array}{lll} & \{x \mapsto 1, y \mapsto 2\} & (\underline{*y};\ \mathit{free}\ y) + ({*x};\ \mathit{free}\ x) \\ \rightsquigarrow & \{x \mapsto 1, y \mapsto 2\} & (\underline{2;\ \mathit{free}\ y)} + ({*x};\ \mathit{free}\ x) \\ \rightsquigarrow & \{x \mapsto 1\} & 2 + (\underline{*x};\ \mathit{free}\ x) \\ \rightsquigarrow & \{x \mapsto 1\} & 2 + (\underline{1;\ \mathit{free}\ x)} \\ \rightsquigarrow & \{\} & \underline{2 + 1} \\ \rightsquigarrow & \{\} & 3 \end{array}$$

We might like to show this commutativity property holds for any choice of expressions e_1 and e_2, but it is hard to prove. Indeed, these examples demonstrate that reordering subexpressions can drastically change the way a derivation looks, and so even stating the required induction hypothesis is difficult. To address this, in section 3 we will introduce a compositional semantics for our language with explicit naming that allows us to present a simple proof of this fact. ◇

2.5 Reflection

In this section, we reflect on our design decisions for explicit naming.

Choice of primitives. Our decision to use a lambda calculus to present our ideas was motivated by the desire to keep the semantics as simple as possible. It is notable that the App rule performs two roles: creating new names, and providing a notion of computation. We could instead have introduced a 'let' expression to create new names, which would simplify some of the presentation, but we would still need another operation to give our language expressive power. However, our ideas are not tied to this lambda calculus, and it is simple to create variants or extensions of this language with new constructs as desired. For example, one could introduce a primitive to compare two names for equality, which corresponds to testing equality of pointers. We could also introduce a 'mutual let' construction to introduce cycles onto a heap.

Generality of primitives. One goal of our work is to produce a language that can be used to investigate type systems for safe memory management. In light of this, it is important that our primitives are as permissive as possible, so that the language can be applied in a range of settings. For example, even though many such type systems forbid freeing aliased pointers, we permit this behaviour, since there are some situations in which this might be allowed.

Immutable bindings. The restriction to immutable bindings is the default for pure functional languages such as Haskell. When translated into the language of memory management, immutability means that we forbid manipulation of values behind pointers. Pointers with this restriction are used in practice: for example, Rust has a notion of *immutable borrows* [Matsakis and Klock 2014], a kind of pointer whose data cannot be modified, and Haskell has *stable pointers* [Peyton Jones et al. 2000; Reid 1994], which are pointers to (immutable) Haskell objects that may be passed to foreign functions, behaving similarly to explicit names. We note that many of the challenges of manual memory management are present even in the absence of mutation, most notably use-after-free errors, one of the key issues encountered when programming with pointers.

3 A Compositional Semantics

The semantics defined in the previous section simultaneously handles two tasks: performing a computation, and tracking when names are freed. In this section, we show how to separate these two tasks, factoring our heap-based semantics into two simpler parts. The resulting semantics that we obtain is equivalent to the heap-based semantics of section 2.

First of all, we give an evaluation semantics for our lambda calculus with explicit naming. This semantics fully captures the denotational behaviour of an expression, but does not model the operational aspect of freeing names. Next, we describe a system of *effects* to track the way that evaluating an expression interacts with a heap. Crucially, this is done in a way that maintains compositionality at the level of heaps: we do not need to thread the heap through our calculations in order to determine the effect of a complicated expression. And finally, we define a partial order on effects, which can be used to reason about effects in a compositional manner, even in the presence of uncertainty; this feature has no direct counterpart in the heap-based semantics.

3.1 Denotations of Expressions

In this section, we give an evaluation semantics for our language, which tracks only the meaning of expressions and not their effects on the heap. Our partial denotation function $[\![-]\!]_{(-)}$ maps an expression e and a *context* Γ to a *denotational value* w. Here, a context is a partial mapping from names to denotational values, and we will shortly define what a denotational value is. To distinguish such values from the notion of values defined in section 2, we will sometimes refer to the latter as *operational values*. Unlike heaps, the contexts used in our evaluation semantics will not be threaded sequentially through our semantic rules. For example, in an application $e_1\ e_2$, the same context will be used to evaluate both subexpressions e_1 and e_2.

Recall that the value assigned to a given name, if it exists, will never change (example 2.4). This suggests that we might be able to track the value assigned to a name within the denotation of the name itself, rather than in an external heap. To this end, we define the following rules:

$$\frac{\Gamma(x) = w}{[\![x]\!]_\Gamma = (x \mapsto w)}\ \text{D-VAR} \qquad\qquad \frac{[\![e]\!]_\Gamma = (x \mapsto w)}{[\![*e]\!]_\Gamma = w}\ \text{D-READ}$$

Here, $(x \mapsto w)$ is the denotational value corresponding to a name x that is bound to the denotational value w. Thus, the D-VAR rule states that a name evaluates to itself, but that we additionally store the value it is bound to. In contrast to the VAR rule, this means that a name that does not appear in the domain of its context has no denotation. The D-READ rule does not access the context to read from a name, but instead retrieves the stored value directly from its argument. Therefore, as this rule does not access a heap, it can never fail if its argument evaluates to a name.

Example 3.1 (evaluating variables). The denotation of $*x$ in a context Γ is precisely $\Gamma(x)$. We may directly calculate the following: $[\![*x]\!]_\Gamma = w \iff \exists y.\ [\![x]\!]_\Gamma = (y \mapsto w) \iff \Gamma(x) = w$. ◇

Because this semantics does not model name freeing, we also define the following rule, which states that the operation of freeing a name has no effect on denotations:

$$\frac{}{[\![e;\ \mathit{free}\ e']\!]_\Gamma = [\![e]\!]_\Gamma}\ \text{D-FREE}$$

Finally, we add rules for abstraction and application. Because we are now using a context in place of a heap, abstractions denote *closures*, storing the context in which they were defined:

$$\frac{}{[\![\lambda x.\, e]\!]_\Gamma = \lambda^\Gamma x.\, e}\ \text{D-LAM} \qquad\qquad \frac{[\![e_1]\!]_\Gamma = \lambda^{\Gamma'} x.\, e}{[\![e_1\ e_2]\!]_\Gamma = [\![e]\!]_{\Gamma', x \mapsto [\![e_2]\!]_\Gamma}}\ \text{D-APP}$$

The D-LAM rule states that an abstraction evaluates to itself, additionally remembering the context in which it is evaluated. The D-APP rule describes the usual way to evaluate applications, where the body e of the closure is evaluated in its stored context Γ', extended by binding the bound variable x to the result of evaluating e_2. In a similar way to the APP rule, the D-APP rule should be viewed as choosing a fresh name x for the bound variable.

Example 3.2 (identity function). In the previous section we saw that evaluating the expression $(\lambda x.\, *x)\ (1 + 2)$ with the empty heap resulted in the value 3 and the heap with the binding $x \mapsto 3$. The denotation of the same expression in a context Γ gives the same value:

$$\begin{aligned} [\![(\lambda x.\, *x)\ (1+2)]\!]_\Gamma &= [\![*x]\!]_{\Gamma, x \mapsto [\![1+2]\!]_\Gamma} \\ &= [\![*x]\!]_{\Gamma, x \mapsto 3} \\ &= 3 \end{aligned}$$ ◇

We are now in a place to define that a *denotational value* is either a closure $(\lambda^{\Gamma} x.\, e)$, where Γ is a context, or a name bound to a denotational value $(x \mapsto w)$. Importantly, while expressions and operational values are defined inductively, we define denotational values *coinductively*. This means that we allow for infinite chains of bindings such as:

$$(x \mapsto (y \mapsto (x \mapsto (y \mapsto \cdots))))$$

and we allow bindings in a context Γ to refer to Γ itself, as in:

$$\Gamma(x) \;=\; \lambda^{\Gamma} y.\, e$$

Example 3.3 (contexts with cycles). Consider the heap with a cycle mapping x to y and y to x. Given such a heap, we can show that dereferencing x twice gives x itself:

$$\begin{array}{lll} & \{x \mapsto y, y \mapsto x\} & **\underline{x} \\ \rightsquigarrow & \{x \mapsto y, y \mapsto x\} & \underline{*y} \\ \rightsquigarrow & \{x \mapsto y, y \mapsto x\} & x \end{array}$$

Similar behaviour can be expressed in this evaluation semantics. Consider the context Γ defined by $\Gamma(x) = (y \mapsto \Gamma(y))$ and $\Gamma(y) = (x \mapsto \Gamma(x))$. Then we can calculate the following:

$$\begin{array}{rcl} [\![x]\!]_{\Gamma} & = & (x \mapsto \Gamma(x)) \\ [\![*x]\!]_{\Gamma} & = & \Gamma(x) \;=\; (y \mapsto \Gamma(y)) \\ [\![**x]\!]_{\Gamma} & = & \Gamma(y) \;=\; (x \mapsto \Gamma(x)) \;=\; [\![x]\!]_{\Gamma} \end{array}$$

◇

3.2 Tracking Effects

The evaluation semantics captures the denotational meaning of expressions, but does not model name freeing. For example, consider the following expression:

$$*((\lambda x.\, (x;\, \mathit{free}\; x))\; 4)$$

It has denotation 4, but does not evaluate in the heap-based semantics. In particular, evaluation gets stuck at the end as dereferencing x requires a binding for x in the heap:

$$\begin{array}{lll} & H & *(\underline{(\lambda x.\, (x;\, \mathit{free}\; x))\; 4}) \\ \rightsquigarrow & H, x \mapsto 4 & *(\underline{x;\, \mathit{free}\; x}) \\ \rightsquigarrow & H & *x \end{array}$$

To model this kind of behaviour, we describe a way to abstractly track the *effect* that evaluating an expression has on the heap. The idea is that the effect of an expression behaves like a log, tracking which names have been read from and freed, and in what order. We can then call a computation valid if its log contains no instance of a name being used after it is freed. In a sense to be defined in section 4, valid computations have equal behaviour in both semantics.

We formalise this idea as follows. A *name effect* is an element q of the set

$$\{1, \mathsf{read}, \mathsf{free}\}$$

A name effect describes what happens to a given name over the course of a computation:

- 1 means that the name was not read from or freed;
- read means that the name was read one or more times;
- free means that the name was read zero or more times, and then freed.

Note that we only track reading and freeing, not assignment to a name, and using the notation q for name effects reflects the fact that they form an ‘effect quantale’, as we shall see later on in this section. We can compose two name effects using the partial binary operator $(-) \blacksquare (-)$ called *sequential composition* and pronounced ‘then’, which is defined as follows:

▪	1	read	free
1	1	read	free
read	read	read	free
free	free	$\bot$	$\bot$

Here, $\bot$ denotes that a particular combination is left undefined. The fact that $\blacksquare$ is partial encapsulates our notion of validity: if $q_1 \blacksquare q_2$ is undefined, it is not valid to compose the two name effects. For example, $\mathsf{free} \blacksquare \mathsf{read}$ is undefined, because it is not valid to use a name after freeing it. Note that composition with an undefined value is left undefined:

$$q \blacksquare \bot \;=\; \bot \blacksquare q \;=\; \bot$$

It is easy to check that composition of name effects is associative. That is, when either side in the following equation is defined, so is the other, and they are equal:

$$q_1 \blacksquare (q_2 \blacksquare q_3) \;=\; (q_1 \blacksquare q_2) \blacksquare q_3$$

Moreover, 1 is the identity for composition, so name effects form a partial monoid:

$$1 \blacksquare q \;=\; q \;=\; q \blacksquare 1$$

Example 3.4 (composing name effects). Intuitively, the name effect of x in the expression $*x$; *free* x is given by $\mathsf{read} \blacksquare \mathsf{free}$, which simplifies to free. This is formalised by the semantics given below. ⋄

We now define an *effect* to be a total function mapping each name to its name effect. This means that we track the effect on each name separately: reading or freeing x has no impact on the validity of reading or freeing any other name y. We use the letter q for both name effects and effects; in practice, it will be clear which kind of effect is meant. We define a partial monoid structure on effects by $(q \blacksquare q')(x) = q(x) \blacksquare q'(x)$ and $1(x) = 1$. We will also find it useful to define basic effects that read and free a given name x, and have no effect on any other names:

$$(\mathsf{read}\ x)(y) \;=\; \begin{cases} \mathsf{read} & \text{if } x = y \\ 1 & \text{otherwise} \end{cases}$$

$$(\mathsf{free}\ x)(y) \;=\; \begin{cases} \mathsf{free} & \text{if } x = y \\ 1 & \text{otherwise} \end{cases}$$

Using the above ideas, we can now define an *effectful* semantics that simultaneously computes the denotation and effect of an expression. Judgments are of the form $\Gamma \vdash e \Downarrow q : w$, which can be read as ‘in context Γ, the expression e has effect q and denotes w’. This can be seen as an *instrumentation* of the evaluation semantics, with the rules for the effectful semantics having similar form to the corresponding rules for the evaluation semantics, but tracking extra information. Indeed, our rules satisfy the following implication, up to choice of fresh names:

$$\Gamma \vdash e \Downarrow q : w \;\implies\; [\![e]\!]_\Gamma = w$$

We will use the convention that whenever a composition $q_1 \cdot q_2$ occurs in a rule, we assume that this composition is in fact defined. The rules themselves are defined as follows:

$$\frac{\Gamma(x) = w}{\Gamma \vdash x \Downarrow 1 : (x \mapsto w)}\ \textsc{E-Var} \qquad \frac{}{\Gamma \vdash \lambda x.\, e \Downarrow 1 : \lambda^{\Gamma} x.\, e}\ \textsc{E-Lam}$$

$$\frac{\Gamma \vdash e_1 \Downarrow q_1 : \lambda^{\Gamma'} x.\, e \qquad \Gamma \vdash e_2 \Downarrow q_2 : w \qquad (\Gamma', x \mapsto w) \vdash e \Downarrow q_3 : w'}{\Gamma \vdash e_1\ e_2 \Downarrow q_1 \cdot q_2 \cdot q_3 : w'}\ \textsc{E-App}$$

$$\frac{\Gamma \vdash e \Downarrow q : (x \mapsto w)}{\Gamma \vdash {*e} \Downarrow q \cdot \mathsf{read}\ x : w}\ \textsc{E-Read} \qquad \frac{\Gamma \vdash e_1 \Downarrow q_1 : w \qquad \Gamma \vdash e_2 \Downarrow q_2 : (x \mapsto w')}{\Gamma \vdash e_1;\ \mathit{free}\ e_2 \Downarrow q_1 \cdot q_2 \cdot \mathsf{free}\ x : w}\ \textsc{E-Free}$$

There are a number of points to note about these rules. First of all, the rules E-Var and E-Lam for names and lambda abstractions always yield the 'do-nothing' effect 1. This corresponds to the fact that the heap-based Var and Lam rules do not read from or modify the heap. The other rules (E-App, E-Read, E-Free) similarly correspond to rules from the heap-based semantics (App, Read, Free), describing the order of operations carried out on the heap.

Secondly, the effects of intermediate expressions are never analysed, but are only used to compute the overall effect via sequential composition. In turn, while we track effects that capture how evaluation interacts with the heap, the context used to evaluate each effect is not threaded through the rules, with the same context being used for each subexpression in a compound term. The only rule that modifies the context is E-App, which updates the captured context Γ' with a new binding for x. And finally, specifying both the denotation and effect in a single rule set, rather using separate rules for each part, avoids side conditions about the choice of fresh names in the two parts.

We conclude by returning to our initial example, $*((\lambda x.\, (x;\ \mathit{free}\ x))\ 4)$. Using our effectful semantics gives the following derivation, in which uses of E-Var and E-Lam are elided for simplicity:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x \mapsto 4 \vdash x;\ \mathit{free}\ x \Downarrow \mathsf{free}\ x : (x \mapsto 4)}}{\Gamma \vdash (\lambda x.\, (x;\ \mathit{free}\ x))\ 4 \Downarrow \mathsf{free}\ x : (x \mapsto 4)}}{\Gamma \vdash *((\lambda x.\, (x;\ \mathit{free}\ x))\ 4) \Downarrow \mathsf{free}\ x \cdot \mathsf{read}\ x : 4}$$

However, the final composition of effects, $\mathsf{free}\ x \cdot \mathsf{read}\ x$, is undefined as it involves a read of a name after it is freed. Hence, the above derivation is not actually valid, which corresponds to the fact that the expression fails to evaluate in our heap-based semantics.

3.3 Reasoning

In this section, we present some examples of how our new semantics can be used to reason about the denotational and effectful behaviour of expressions.

Example 3.5 (let expressions). We can define syntax for 'let' expressions as follows:

$$(\mathit{let}\ x = e_1\ \mathit{in}\ e_2) \ \coloneqq\ (\lambda x.\, e_2;\ \mathit{free}\ x)\ e_1$$

Their behaviour is then captured by the following derivation tree:

$$\dfrac{\Gamma \vdash e_1 \Downarrow q_1 : w_1 \qquad \dfrac{\Gamma, x \mapsto w_1 \vdash e_2 \Downarrow q_2 : w_2}{\Gamma, x \mapsto w_1 \vdash e_2;\ \mathit{free}\ x \Downarrow q_2 \cdot \mathsf{free}\ x : w_2}}{\Gamma \vdash (\lambda x.\, e_2;\ \mathit{free}\ x)\ e_1 \Downarrow q_1 \cdot q_2 \cdot \mathsf{free}\ x : w_2}$$

This derivation shows that the denotation of *let* $x = e_1$ *in* e_2 is given by the denotation of e_2 in the context extended by binding x to the denotation of e_1, and the effect is that of first evaluating e_1, then e_2, and finally freeing x. Hence, we have the following derivable rule:

$$\frac{\Gamma \vdash e_1 \Downarrow q_1 : w_1 \qquad \Gamma, x \mapsto w_1 \vdash e_2 \Downarrow q_2 : w_2}{\Gamma \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \Downarrow q_1 \cdot q_2 \cdot \mathsf{free}\ x : w_2}$$

For example, we can use this rule to obtain the semantics of the expression *let* $x = e$ *in* $*x$:

$$\frac{\Gamma \vdash e \Downarrow q : w \qquad \dfrac{}{\Gamma, x \mapsto w \vdash *x \Downarrow \mathsf{read}\ x : w}}{\Gamma \vdash \mathit{let}\ x = e\ \mathit{in}\ *x \Downarrow q \cdot \mathsf{read}\ x \cdot \mathsf{free}\ x : w}$$

The overall effect simplifies to $q \cdot \mathsf{free}\ x$, where q is the effect of evaluating the expression e, and the overall denotation w is simply the result of this evaluation. This example shows how we can reason about derived concepts such as 'let' expressions in a simple manner. ◇

Example 3.6 (commuting effects). If q_1 and q_2 act on disjoint sets of names, then their composites $q_1 \cdot q_2$ and $q_2 \cdot q_1$ are always defined and are equal. This commutativity property allows us to reorder computations without changing the overall effect. For example, the expressions

$$e;\ \mathit{free}\ x;\ \mathit{free}\ y \quad \text{and} \quad e;\ \mathit{free}\ y;\ \mathit{free}\ x$$

always have the same denotation and effect, because freeing names is commutative:

$$\frac{\dfrac{\Gamma \vdash e \Downarrow q : w}{\Gamma \vdash e;\ \mathit{free}\ x \Downarrow q \cdot \mathsf{free}\ x : w}}{\Gamma \vdash e;\ \mathit{free}\ x;\ \mathit{free}\ y \Downarrow q \cdot \mathsf{free}\ x \cdot \mathsf{free}\ y : w} \qquad \frac{\dfrac{\Gamma \vdash e \Downarrow q : w}{\Gamma \vdash e;\ \mathit{free}\ y \Downarrow q \cdot \mathsf{free}\ y : w}}{\Gamma \vdash e;\ \mathit{free}\ y;\ \mathit{free}\ x \Downarrow q \cdot \mathsf{free}\ y \cdot \mathsf{free}\ x : w}$$

As another example, let us revisit the following expressions from example 2.5:

$$e_1 + e_2 \quad \text{and} \quad e_2 + e_1$$

During evaluation of both expressions, the subexpressions e_1 and e_2 are evaluated in the same context Γ. Concretely, the two expressions have the following derivations:

$$\frac{\Gamma \vdash e_1 \Downarrow q_1 : w_1 \qquad \Gamma \vdash e_2 \Downarrow q_2 : w_2}{\Gamma \vdash e_1 + e_2 \Downarrow q_1 \cdot q_2 : w_1 + w_2} \qquad \frac{\Gamma \vdash e_2 \Downarrow q_2 : w_2 \qquad \Gamma \vdash e_1 \Downarrow q_1 : w_1}{\Gamma \vdash e_2 + e_1 \Downarrow q_2 \cdot q_1 : w_2 + w_1}$$

Whenever $q_1 \cdot q_2$ and $q_2 \cdot q_1$ are both defined, they are equal. Therefore, by commutativity of addition, if both $e_1 + e_2$ and $e_2 + e_1$ evaluate in a given context, they behave identically. ◇

3.4 Ordering Effects

Our notion of effects can naturally be given a partial order $\leq$ that respects the sequential composition operation $\cdot$ in a suitable sense. This allows us to reason compositionally by considering *bounds* on effects, even in cases where we do not know the exact effect that evaluating an expression will have. Concretely, we give name effects a linear order by setting

$$1 < \mathsf{read} < \mathsf{free}$$

If an expression might free a name (free) or do nothing to the heap (1), an *upper bound* for both cases is free. Similarly, if it might read from a name but might do nothing, an upper bound for the effect in either case is read. If we have no information about what an expression might do to a name, other than that the effect is valid, the loosest possible bound on the effect is free.

We write $q_1 \sqcup q_2$ for the least upper bound of name effects q_1 and q_2, and extend $\leq$ and $\sqcup$ to effects in a pointwise manner. The ordering respects sequential composition in the sense that:

$$q \cdot q_1 \cdot q' \text{ is defined } \wedge \ q_2 \leq q_1 \implies q \cdot q_2 \cdot q' \text{ is defined } \wedge \ q \cdot q_2 \cdot q' \leq q \cdot q_1 \cdot q'$$

Because the validity of a computation is determined by whether its effect is defined, upper bounds on effects can be used to conservatively estimate whether computations will be valid even when the effect of an intermediate expression is not known exactly.

Example 3.7 (joining effects). Consider an expression of the following form.

$$e \ = \ if \ *x \ then \ e_1 \ else \ e_2$$

Suppose we know that e_1 and e_2 evaluate as follows.

$$\Gamma \vdash e_1 \Downarrow q_1 : w_1 \qquad\qquad \Gamma \vdash e_2 \Downarrow q_2 : w_2$$

If we do not know whether x is true or false in Γ, then we do not know the overall result of this computation, but an upper bound for its effect in either case is $\text{read } x \cdot (q_1 \sqcup q_2)$. We can use this bound to reason about larger programs that include this as a subexpression without resorting to case splitting on the truth value of x in Γ. ◇

These definitions make the set of effects into an effect quantale, a notion we will now define.

Definition 3.8 (effect quantale). An *effect quantale* [Gordon 2021] is a set E of *effects*, together with partial binary operations $\sqcup$ and $\cdot$ and an identity element $1 \in E$, satisfying various laws:

- $(E, \sqcup)$ is a partial join-semilattice;
- $(E, \cdot, 1)$ is a partial monoid, which means that the identities $a \cdot 1 = 1 \cdot a = a$ always hold, and associativity $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ holds when either side is defined;
- Sequencing distributes over joins in both directions, so both $a \cdot (b \sqcup c) = (a \cdot b) \sqcup (a \cdot c)$ and $(a \sqcup b) \cdot c = (a \cdot c) \sqcup (b \cdot c)$ hold whenever either side is defined.

It is easy to check that name effects, and therefore effects, form an effect quantale.

3.5 Other Effect Systems

We used a particular partial monoid to track effects, but other choices are possible too. By way of example, we present an alternative partial monoid based on *ordinals*, which allows us to capture more information about the behaviour of computations. We begin by replacing the set of name effects $\{1, \text{read}, \text{free}\}$ with the collection of countable ordinals:

$$0, 1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots, \omega \cdot 2, \omega \cdot 2 + 1, \ldots$$

In this setting, we view the ordinal 0 as representing the ‘do-nothing’ effect, finite ordinals $1, 2, \ldots$ as corresponding to that number of reads, and the first infinite ordinal ω as corresponding to freeing a name. Ordinals above ω are considered to be invalid effects.

Sequential composition is given by ordinal addition. For example, the ordinal equation $2 + 3 = 5$ means that ‘2 reads’ followed by ‘3 reads’ corresponds to ‘5 reads’. As $n + \omega = \omega$ for any finite ordinal n, the effect of reading from a name any number of times and then freeing it is equal to the effect of simply freeing the name. However, as $\omega + 1 > \omega$, it is invalid to read from a name after freeing it. Similarly, $\omega + \omega = \omega \cdot 2 > \omega$, so freeing a name twice is invalid. In this manner, ordinal addition elegantly captures the notion of sequential composition for this form of effect.

Countable ordinals form an effect quantale where sequential composition is ordinal addition and the join operator is given by $\alpha \sqcup \beta = \max(\alpha, \beta)$. Therefore, total functions from names to countable ordinals also form an effect quantale by pointwise definition, and we call such functions *ordinal* *effects*. Incidentally, this provides a quick proof that our original set of effects $\{1, \mathsf{read}, \mathsf{free}\}$ is an effect quantale, as we can translate proofs from countable ordinals to name effects.

We now describe how our effectful semantics is modified to use ordinal effects. Judgments have the form $\Gamma \vdash e \Downarrow^O q : w$, where q is an ordinal effect; we write the judgment relation as $\Downarrow^O$ to distinguish it from the original effectful semantics. The modified E-Read and E-Free rules are as follows, where $(x \mapsto \alpha)$ is the ordinal effect mapping x to the ordinal α and all other names to 0:

$$\frac{\Gamma \vdash e \Downarrow^O q : (x \mapsto w)}{\Gamma \vdash *e \Downarrow^O q \cdot (x \mapsto 1) : w} \qquad \frac{\Gamma \vdash e_1 \Downarrow^O q_1 : w \qquad \Gamma \vdash e_2 \Downarrow^O q_2 : (x \mapsto w')}{\Gamma \vdash e_1;\, \mathit{free}\ e_2 \Downarrow^O q_1 \cdot q_2 \cdot (x \mapsto \omega) : w}$$

The resulting semantics is a generalisation of our original semantics. To formalise this, we define a partial function from countable ordinals to name effects by:

$$E(\alpha) \;=\; \begin{cases} 1 & \text{if } \alpha = 0 \\ \mathsf{read} & \text{if } 1 \leq \alpha < \omega \\ \mathsf{free} & \text{if } \alpha = \omega \end{cases}$$

It is then a simple rule induction in both directions to prove the following, which makes precise that if we ‘forget’ the number of reads then the ordinal semantics reduces to the original:

$$\Gamma \vdash e \Downarrow q : w \quad\Longleftrightarrow\quad (\exists q'.\, \Gamma \vdash e \Downarrow^O q' : w \;\wedge\; \forall x.\, E(q'(x)) = q(x))$$

Many familiar operations on ordinals correspond to transformations of effects. For example, consider the operation $\omega \cdot (-)$ on countable ordinals. The only way for $\omega \cdot q$ to be valid is if q is 0 or 1. This corresponds to a name that can be read at most once. We can use this to easily construct a new semantics with *affine names* that can only be read at most once. Let A be a particular set of names that we will call ‘affine’, and define $\mathsf{aff}(q)$ to be the ordinal effect given by:

$$\mathsf{aff}(q)(x) \;=\; \begin{cases} \omega \cdot q(x) & \text{if } x \in A \\ q(x) & \text{if } x \notin A \end{cases}$$

Then:

$$\Gamma \vdash e \Downarrow^O q : w \quad\Longleftrightarrow\quad \Gamma \vdash e \Downarrow^A \mathsf{aff}(q) : w$$

where the nontrivial rules for $\Gamma \vdash e \Downarrow^A q : w$ are given by:

$$\frac{\Gamma \vdash e \Downarrow^A q : (x \mapsto w)}{\Gamma \vdash *e \Downarrow^A q \cdot \mathsf{aff}(x \mapsto 1) : w} \qquad \frac{\Gamma \vdash e_1 \Downarrow^A q_1 : w \qquad \Gamma \vdash e_2 \Downarrow^A q_2 : (x \mapsto w')}{\Gamma \vdash e_1;\, \mathit{free}\ e_2 \Downarrow^A q_1 \cdot q_2 \cdot \mathsf{aff}(x \mapsto \omega) : w}$$

Indeed, because ordinal multiplication distributes over addition, we have the following identity:

$$\mathsf{aff}(q_1 \cdot q_2) \;=\; \mathsf{aff}(q_1) \cdot \mathsf{aff}(q_2)$$

This allows us to prove this equivalence by a simple rule induction in both directions.

Another common operation on ordinals is the *natural* or *Hessenberg sum*, written $\oplus$. This is another notion of ordinal addition which is commutative (and associative and has unit 0). It is defined on ordinals at most ω by the following equations:

$$\begin{aligned} n \oplus m &= m \oplus n = n + m && \text{if } n, m \text{ finite} \\ n \oplus \omega &= \omega \oplus n = \omega + n && \text{if } n \text{ finite} \\ \omega \oplus \omega &= \omega + \omega \end{aligned}$$

This corresponds to the *parallel composition* of effects. For example, reading from a name twice, while in parallel reading from that name three times, yields five reads from that name in total. It is not valid to read from a name and free it in parallel, since the behaviour of such a computation depends on the order of execution. This corresponds to the fact that $1 \oplus \omega = \omega \oplus 1 = \omega + 1 > \omega$.

In general, it can be shown that $\alpha + \beta \leq \alpha \oplus \beta$ and so $\beta + \alpha \leq \alpha \oplus \beta$ by commutativity of $\oplus$, and hence the parallel composition of two effects is an upper bound for both possible orders of sequential composition. In fact, for ordinals at most ω, it is easy to show that $\alpha \oplus \beta = \max(\alpha + \beta, \beta + \alpha)$, so it is the least upper bound of both orders of sequential composition. It is in this sense that the analogy to parallel computation is made precise. This suggests a way to define a rule for a parallel composition primitive, where $\oplus$ is defined pointwise on ordinal effects:

$$\frac{\Gamma \vdash e_1 \Downarrow^O q_1 : w_1 \qquad \Gamma \vdash e_2 \Downarrow^O q_2 : w_2 \qquad \Gamma, x \mapsto w_1, y \mapsto w_2 \vdash e \Downarrow^O q_3 : w}{\Gamma \vdash (\text{let } x = e_1 \parallel y = e_2 \text{ in } e) \Downarrow^O (q_1 \oplus q_2) \cdot q_3 \cdot (\text{free } x \oplus \text{free } y) : w} \text{ E-PAR}$$

This section demonstrates that variations to our effect-based semantics are easy to produce, and they are sufficiently general to model a variety of problems.

4 Equivalence of the Semantics

In this section, we state the equivalence between the semantics of sections 2 and 3, and discuss how we can transport useful results across the equivalence. Informally, the theorem says that if an expression e evaluates in the heap semantics using a given heap H, then the expression also evaluates in the effectful semantics with a particular context derived from H, and vice versa. Moreover, when this holds, the two computations produce the same value. To make this equivalence precise, we need to establish some conversions between the heaps and operational values of section 2 and the contexts and denotational values of section 3.

First, we define a way to 'forget' the extra data held by a denotational value to convert it into an operational value. This operation is written $w \mapsto \overline{w}$, and is defined by the following equations:

$$\overline{(x \mapsto w)} \;=\; x \qquad\qquad \overline{\lambda^\Gamma x.\, e} \;=\; \lambda x.\, e$$

Next, we define a translation from heaps to contexts. Given a heap H, its translation is written $\mathrm{tr}(H)$, and is given by the following coinductive rules:

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

For example, consider the heap

$$H \;=\; \{x \mapsto (\lambda t.\, e), y \mapsto x, z \mapsto t\}$$

This has translation $\mathrm{tr}(H) = \Gamma$ given by

$$\Gamma(x) \;=\; \lambda^\Gamma t.\, e \qquad\qquad \Gamma(y) \;=\; (x \mapsto \lambda^\Gamma t.\, e)$$

Note that $\Gamma(z)$ is undefined because $H(z) = t$ is a *dangling pointer*: it is a name not in the domain of H. In general, we will say that a heap H is *closed* if whenever $H(x)$ is defined, the free variables of $H(x)$ are contained in the domain of H. This should be viewed as a well-formedness constraint on heaps, ensuring that translations to contexts are faithful. It is easy to expand any heap H into a closed heap $H' \supseteq H$, for example by setting $H'(t) = t$ whenever t is a free variable of some $H(x)$.

Using the above, we can now state the equivalence theorem:

THEOREM 4.1 (EQUIVALENCE OF SEMANTICS). *Suppose H is a closed heap containing all free variables of the expression e in its domain. Then we have the equivalence*

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

where the same sequence of fresh names was chosen by each semantics.

This theorem states that e evaluates in the heap-based semantics using initial heap H if and only if it evaluates in the effect-based semantics using context $\mathrm{tr}(H)$. Moreover, when this holds, the two computations produce the same value: if the operational value produced by the heap-based semantics is v and the denotational value produced by the effect-based semantics is w, then $v = \overline{w}$.

We will prove this theorem in section 5. In the remainder of this section, we explore some examples to show different ways that this theorem can be used. In particular, we will demonstrate that it is easier to reason about various program transformations in the effectful semantics, but that we can use the equivalence theorem to transfer results to the heap semantics. We will assume without comment that relevant heaps satisfy the hypotheses of the equivalence theorem.

Example 4.2 (commuting effects, revisited). In example 3.6, we exploited compositionality to show that if $e_1 + e_2$ and $e_2 + e_1$ both evaluate in some context Γ, then they both evaluate to the same result. We would like to prove the same about the heap semantics, but this would be difficult to do directly because the derivations in this semantics for $e_1 + e_2$ and $e_2 + e_1$ may look completely different. Instead, we will make use of our equivalence theorem. Suppose that

$$H_1 : e_1 + e_2 \Downarrow H_2 : v \qquad\qquad H_1 : e_2 + e_1 \Downarrow H_2' : v'$$

By theorem 4.1, we obtain

$$\mathrm{tr}(H_1) \vdash e_1 + e_2 \Downarrow q : w \qquad\qquad \mathrm{tr}(H_1) \vdash e_2 + e_1 \Downarrow q' : w'$$

where $\overline{w} = v$ and $\overline{w'} = v'$. But by example 3.6, we know that $w = w'$, so $v = v'$. ◇

Example 4.3 (common subexpression elimination). Consider the expression $e + e$, which evaluates the expression e twice. We may want to transform this expression into $\mathit{let}\ x = e\ \mathit{in} *x + *x$, which would only evaluate e once. Suppose that we attempt to prove the validity of the transformation in the heap semantics by computing the following derivations:

$$\frac{H_1 : e \Downarrow H_2 : v_1 \qquad H_2 : e \Downarrow H_3 : v_2}{H_1 : e + e \Downarrow H_3 : v_1 + v_2} \qquad\qquad \frac{H_1 : e \Downarrow H_2 : v_1}{H_1 : \mathit{let}\ x = e\ \mathit{in} *x + *x \Downarrow H_2 : v_1 + v_1}$$

In this semantics, we would need to prove that $v_1 = v_2$ to show that the transformation is valid. We would also need to check that the remainder of the program that follows the evaluation of $e + e$ can be executed starting with H_2 and not H_3, which would require us to reason in detail about the changes on the heap that could be caused by evaluating e for a second time. This is significantly easier in the effect semantics, in which we have the following derivations:

$$\frac{\Gamma \vdash e \Downarrow q : w}{\Gamma \vdash e + e \Downarrow q \cdot q : w + w} \qquad\qquad \frac{\Gamma \vdash e \Downarrow q : w}{\Gamma \vdash \mathit{let}\ x = e\ \mathit{in} *x + *x \Downarrow q \cdot \mathrm{free}\ x : w + w}$$

Due to the compositional properties of this semantics, both evaluations of e occur within the same context Γ, and so must produce the same value w. This shows that both $e+e$ and $\mathit{let}\ x = e\ \mathit{in} *x+*x$ must produce the same value. Moreover, in this semantics it is easy to show that it is always valid to replace the former with the latter inside any complicated expression. Indeed, as x is fresh,

$$q_1 \cdot (q \cdot q) \cdot q_2 \text{ is defined} \implies q_1 \cdot (q \cdot \mathrm{free}\ x) \cdot q_2 \text{ is defined}$$

where q_1 and q_2 encode the effect of the surrounding parts of the expression. Therefore, by making use of our equivalence theorem, the same is true of the heap semantics: the expression $e + e$ can be replaced with $\mathit{let}\ x = e\ \mathit{in} *x + *x$ without changing the behaviour of an overall program. ◇

Example 4.4 (dead code elimination). Consider the expression

$$\mathit{let}\ x = e_1\ \mathit{in}\ e_2$$

where x does not appear free in e_2. We want to show that

$$(H_1 : (\mathit{let}\ x = e_1\ \mathit{in}\ e_2) \Downarrow H_2 : v) \implies (\exists H_2'.\ H_1 : e_2 \Downarrow H_2' : v)$$

under the assumption that the free variables of e_1 and e_2 are in the domain of the closed heap H_1. This is difficult to show in the heap semantics alone, because the heap used to evaluate e_2 on the left-hand side is not H_1, and depends on the way that e_1 interacts with the heap. However, by translating to the effectful semantics, we are able to entirely ignore the behaviour of e_1. Indeed, by theorem 4.1, we obtain w and q such that $\overline{w} = v$ and

$$\mathrm{tr}(H_1) \vdash (\mathit{let}\ x = e_1\ \mathit{in}\ e_2) \Downarrow q : w$$

Analysing the proof tree, we obtain

$$\mathrm{tr}(H_1) \vdash e_1 \Downarrow q_1 : w_1 \qquad\qquad \mathrm{tr}(H_1), x \mapsto w_1 \vdash e_2 \Downarrow q_2 : w$$

As the name x does not appear free in the expression e_2, we can eliminate it from the context in the derivation for e_2. More precisely, we can show by a simple rule induction in the effectful semantics that there exists w' with $\overline{w'} = \overline{w} = v$ such that

$$\mathrm{tr}(H_1) \vdash e_2 \Downarrow q_2 : w'$$

Note that it might be the case that $w \neq w'$; this can happen if w and w' contain closures, because the context may or may not contain the binding $x \mapsto w_1$. Then, applying theorem 4.1 in the reverse direction, we obtain the heap derivation for e_2 as required.

$$\exists H_2'.\ H_1 : e_2 \Downarrow H_2' : v$$

The initial heap for this derivation is H_1, not an intermediate heap obtained after evaluating e_1. ⋄

5 Proving the Equivalence

In this section, we prove the equivalence theorem from section 4. In order to do this, we first we adjust our semantics to make the allocation of fresh names precise, allowing us to state our equivalence theorem more formally. We then present the complete proof. The strategy for our proof is to show that the semantics of sections 2 and 3 are both equivalent to a new 'clairvoyant semantics', inspired by the approach of Hackett and Hutton [2019].

5.1 Handling Fresh Names

There are various approaches to handling fresh name generation in the literature. One approach is to augment the semantics with a *name supply list*, in which a list of fresh names is threaded through each judgment. To generate a fresh name in an inference rule, the first element of the list can be removed, passing the tail of the list to later derivations. An alternative approach is to augment each judgment with a single finite set to track which fresh names it created [Pitts and Stark 1993]. When multiple judgments are combined in an inference rule, we typically assume that the sets of fresh names contained in the hypotheses are disjoint, and take their union to find the set of fresh names created by the derived judgment.

We have chosen a variant of the latter approach for our semantics in order to minimise the amount of threading. Instead of a finite set of names, we use a list in order to emphasise the sequential nature of computation. Lists of names are written l, and list concatenation is written $\diamond$. In order to simplify the presentation of our inference rules, we do not add a disjointness condition into each rule; rather, we typically assume that the name lists in completed judgments contain no duplicate names. Judgments in the heap semantics are now written $H_1 : e \Downarrow (l)\ H_2 : v$, and judgments in the effectful semantics are now written $\Gamma \vdash e \Downarrow (l)\ q : w$. For instance, the heap judgment can now be read as 'we can evaluate expression e in initial heap H_1, using the sequence

$$\frac{}{H : x \Downarrow ([])\ H : x} \qquad \frac{}{H : \lambda x.\, e \Downarrow ([])\ H : \lambda x.\, e}$$

$$\frac{H_1 : e_1 \Downarrow (l_1)\ H_2 : \lambda x.\, e \qquad H_2 : e_2 \Downarrow (l_2)\ H_3 : v \qquad (H_3, x \mapsto v) : e \Downarrow (l_3)\ H_4 : v'}{H_1 : e_1\ e_2 \Downarrow (l_1 \diamond l_2 \diamond [x] \diamond l_3)\ H_4 : v'}$$

$$\frac{H_1 : e \Downarrow (l)\ (H_2, x \mapsto v) : x}{H_1 : *e \Downarrow (l)\ (H_2, x \mapsto v) : v} \qquad \frac{H_1 : e_1 \Downarrow (l_1)\ H_2 : v \qquad H_2 : e_2 \Downarrow (l_2)\ (H_3, x \mapsto v') : x}{H_1 : e_1;\ \mathit{free}\ e_2 \Downarrow (l_1 \diamond l_2)\ H_3 : v}$$

$$\frac{\Gamma(x) = w}{\Gamma \vdash x \Downarrow ([])\ 1 : (x \mapsto w)} \qquad \frac{x \notin \operatorname{dom} \Gamma}{\Gamma \vdash \lambda x.\, e \Downarrow ([])\ 1 : \lambda^{\Gamma} x.\, e}$$

$$\frac{\Gamma \vdash e_1 \Downarrow (l_1)\ q_1 : \lambda^{\Gamma'} x.\, e \qquad \Gamma \vdash e_2 \Downarrow (l_2)\ q_2 : w \qquad (\Gamma', x \mapsto w) \vdash e \Downarrow (l_3)\ q_3 : w'}{\Gamma \vdash e_1\ e_2 \Downarrow (l_1 \diamond l_2 \diamond [x] \diamond l_3)\ q_1 \cdot q_2 \cdot q_3 : w'}$$

$$\frac{\Gamma \vdash e \Downarrow (l)\ q : (x \mapsto w)}{\Gamma \vdash *e \Downarrow (l)\ q \cdot \mathsf{read}\ x : w} \qquad \frac{\Gamma \vdash e_1 \Downarrow (l_1)\ q_1 : w \qquad \Gamma \vdash e_2 \Downarrow (l_2)\ q_2 : (x \mapsto w')}{\Gamma \vdash e_1;\ \mathit{free}\ e_2 \Downarrow (l_1 \diamond l_2)\ q_1 \cdot q_2 \cdot \mathsf{free}\ x : w}$$

Fig. 1. Heap-based and effect-based inference rules with name tracking

of fresh names l, to obtain the value v in final heap H_2'. To illustrate the idea, the App rule is now written as follows; the full list of rules is given in fig. 1.

$$\frac{H_1 : e_1 \Downarrow (l_1)\ H_2 : \lambda x.\, e \qquad H_2 : e_2 \Downarrow (l_2)\ H_3 : v \qquad (H_3, x \mapsto v) : e \Downarrow (l_3)\ H_4 : v'}{H_1 : e_1\ e_2 \Downarrow (l_1 \diamond l_2 \diamond [x] \diamond l_3)\ H_4 : v'}$$

This rule defines that the fresh names needed in order to evaluate an application $e_1\ e_2$ are those created by evaluating e_1, then those created by evaluating e_2, then a new name x for the bound variable, then the names created by evaluating the body of the abstraction.

When we say that a name x is *fresh for* an object, we mean that it does not occur free in that object. To use this definition with denotational values, heaps and contexts, we need to define what it means for a name to appear free in these objects. The names appearing free in a heap H are the names in its domain, as well as the names that appear free in any values $H(x)$. We define whether a name x appears free in a context or denotational value inductively, using the following rules:

- If x appears free in w, or if $x = y$, then x appears free in $(y \mapsto w)$;
- If x appears free in Γ, or if $x \neq y$ and x appears free in e, then x appears free in $\lambda^{\Gamma} y.\, e$;
- If x is in the domain of Γ or appears free in any $\Gamma(y)$, then x appears free in Γ.

Unlike an informal description of freshness, this is a well-defined predicate, and has no dependence on a global name state. For convenience, we allow ourselves to say that a list of names l is *fresh for* an object if each of its elements is fresh for that object. The idea of an object being 'fresh for' another arbitrary object is explored further in the study of *nominal sets* [Pitts 2013].

We can now restate our equivalence theorem more precisely:

Theorem 5.1 (equivalence of semantics). *Let l be a list of names without duplicates, fresh for H and e. Suppose additionally that H is closed and contains all free variables of e. Then:*

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

5.2 Clairvoyant Heaps

The main reason the two semantics of sections 2 and 3 are difficult to compare is because of their differing viewpoints on the region in which a name is considered to be bound to a value. In the heap semantics, bindings to names are created and destroyed sequentially, whereas in the effectful semantics, a name can be bound to a value in a context only in a subtree of a derivation.

To reconcile these notions, we use the idea of a *clairvoyant heap*. Rather than updating as a computation proceeds, a clairvoyant heap can 'see the future', and has already stored every binding that will be made. For instance, the clairvoyant application rule has the following rough shape:

$$\frac{C : e_1 \Downarrow \lambda x.\, e \qquad C : e_2 \Downarrow C(x) \qquad C : e \Downarrow v}{C : e_1\ e_2 \Downarrow v}$$

This rule asserts that the result of evaluating e_2 has already been bound to the name x in C. This means that we can use the same clairvoyant heap C when evaluating the body of the abstraction. In general, the same clairvoyant heap will be used for evaluating all parts of an expression.

The idea to create a semantics that can 'see the future' is inspired by the clairvoyant semantics of Hackett and Hutton [2019], in which lazy evaluation is modelled by non-deterministically choosing whether to evaluate an expression or not. In both their paper and ours, the semantics are given some knowledge about future computations to make them easier to reason about.

Our proof strategy is to define a new semantics using clairvoyant heaps, and then show it is equivalent to both the heap semantics and the effectful semantics. For this to be true, our clairvoyant semantics needs to track some kind of extra data in order to correctly determine when computations should fail. It turns out that a convenient sort of data to track is a list of instructions that the computation would perform on a heap if it were to be evaluated using the heap semantics.

Definition 5.2. A *heap transformation* is a list of *instructions* of one of the following forms:

$$x \coloneqq v \qquad \mathrm{rd}\ x \qquad \mathrm{fr}\ x$$

Heap transformations form a monoid; the concatenation of lists t_1 and t_2 is written $t_1 \diamond t_2$.

Heap transformations can be thought of in two ways. First of all, a heap transformation can be viewed as a partial function from heaps to heaps. To describe this, we associate such a partial function to each individual instruction as follows:

- $x \coloneqq v$ corresponds to the partial function given by $H \mapsto (H, x \mapsto v)$, where x is not in the domain of H;
- $\mathrm{rd}\ x$ corresponds to the partial function that is the identity on heaps that contain x in their domain, and undefined elsewhere;
- $\mathrm{fr}\ x$ corresponds to the partial function that removes the entry with name x from the heap if present, and undefined elsewhere.

Then, the partial function associated to a heap transformation $[t_1, \ldots, t_n]$ is their composition:

$$[t_1, \ldots, t_n](H) \ =\ t_n(\cdots t_1(H) \cdots)$$

Alternatively, a heap transformation t can be thought of as a refinement of the data in an effect q. We define an operation to convert a heap transformation into an effect, written $t \mapsto \overline{t}$. Individual instructions are translated according to

$$\overline{x \coloneqq v} \;=\; 1 \qquad \overline{\mathrm{rd}\ x} \;=\; \mathsf{read}\ x \qquad \overline{\mathrm{fr}\ x} \;=\; \mathsf{free}\ x$$

and the translation of a list of instructions is given by

$$\overline{[t_1, \ldots, t_n]} \;=\; \overline{t_1} \cdot \cdots \cdot \overline{t_n}$$

This operation is partial in general, but if $\overline{t_1 \diamond t_2}$ is defined, then so are $\overline{t_1}$ and $\overline{t_2}$. These two descriptions of heap transformations demonstrate how they subsume the notions of effect tracking used by both the heap-based and effect-based semantics.

We can now present our clairvoyant semantics. Judgments are of the form $C : e \Downarrow (l)\ t : v$, where C is a heap (named using the letter C to emphasise its interpretation as a clairvoyant heap), and t is a heap transformation. The semantics is defined by the following rules:

$$\frac{}{C : x \Downarrow ([])\ [] : x}\ \textsc{C-Var} \qquad \frac{}{C : \lambda x.\, e \Downarrow ([])\ [] : \lambda x.\, e}\ \textsc{C-Lam}$$

$$\frac{C : e_1 \Downarrow (l_1)\ t_1 : \lambda x.\, e \qquad C : e_2 \Downarrow (l_2)\ t_2 : C(x) \qquad C : e \Downarrow (l_3)\ t_3 : v}{C : e_1\ e_2 \Downarrow (l_1 \diamond l_2 \diamond [x] \diamond l_3)\ t_1 \diamond t_2 \diamond [x \coloneqq C(x)] \diamond t_3 : v}\ \textsc{C-App}$$

$$\frac{C : e \Downarrow (l)\ t : x}{C : {*e} \Downarrow (l)\ t \diamond [\mathrm{rd}\ x] : C(x)}\ \textsc{C-Read} \qquad \frac{C : e_1 \Downarrow (l_1)\ t_1 : v \qquad C : e_2 \Downarrow (l_2)\ t_2 : x}{C : e_1;\ \mathit{free}\ e_2 \Downarrow (l_1 \diamond l_2)\ t_1 \diamond t_2 \diamond [\mathrm{fr}\ x] : v}\ \textsc{C-Free}$$

5.3 Equivalence of Clairvoyant and Heap Semantics

In this subsection, we prove that every derivation in the heap semantics can be converted to one in the clairvoyant semantics, and vice versa. This proof makes use of the fact that heap transformations t can be treated as partial functions on heaps, allowing us to exactly determine what the final heap of a computation is. We will first prove some useful lemmas.

Lemma 5.3 (heap immutability). *Suppose that $H_1 : e \Downarrow (l)\ H_2 : v$ in the heap semantics, where l has no duplicates and is disjoint from $\mathrm{dom}\, H_1$. Then in all heaps occurring in the derivation tree for $H_1 : e \Downarrow (l)\ H_2 : v$, each variable x is bound to at most one value v'.*

This lemma is a more precise statement of example 2.4.

Proof. A straightforward rule induction: the only way to add a binding into a heap is for the variable to be named in the list l, but since l has no duplicates and is disjoint from the domain of H_1, each such name can be bound at most once. □

We now give the key preservation property that makes the induction in one direction work.

Lemma 5.4. *If $C : e \Downarrow (l)\ t : v$, and $H \subseteq C$ is such that $t(H)$ is defined, then also $t(H) \subseteq C$.*

Proof. The instructions in the heap transformation t have the form $x \coloneqq v$, $\mathrm{rd}\ x$, or $\mathrm{fr}\ x$. By inspection of the rules for the clairvoyant semantics, the only bindings $x \coloneqq v$ contained in t must be of the form $x \coloneqq C(x)$. Therefore, applying any such instruction to a subheap of C yields another subheap of C. Finally, the $\mathrm{rd}\ x$ and $\mathrm{fr}\ x$ instructions do not disrupt this property, as required. □

Proposition 5.5 (equivalence of clairvoyant and heap semantics). *Let l be a list of names with no duplicates, fresh for H_1 and e. Then we have the following equivalence:*

$$H_1 : e \Downarrow (l)\ H_2 : v \iff (\exists C \supseteq H_1.\ \exists t.\ t(H_1) = H_2 \ \wedge\ C : e \Downarrow (l)\ t : v)$$

Proof. In the forward direction, suppose that $H_1 : e \Downarrow (l)\ H_2 : v$. Let C be the union of all heaps occurring in this derivation tree. By heap immutability (lemma 5.3), each name is assigned to at most one value, and as there are only finitely many such names, C is a heap.

It is then easy to show by rule induction that for any heap derivation $H_1 : e \Downarrow (l)\ H_2 : v$, if the clairvoyant heap C is a superset of all heaps occurring in the derivation tree, then there exists a heap transformation t such that $C : e \Downarrow (l)\ t : v$ and $t(H_1) = H_2$. This weakening of the statement is necessary in order to invoke the inductive hypothesis. To show the inductive step, the only nontriviality is to observe that as C contains all heaps occurring in the derivation tree, whenever a binding $(x \mapsto v)$ appears in the heap derivation, we know that $C(x) = v$.

Conversely, in the backward direction, suppose that $C : e \Downarrow (l)\ t : v$, that $C \supseteq H_1$, and that $t(H_1)$ is defined. We show by rule induction that under these assumptions, we have $H_1 : e \Downarrow (l)\ t(H_2) : v$. Lemma 5.4 is the key preservation property that allows us to apply the inductive hypothesis, and the remainder of the proof is purely mechanical. □

5.4 Equivalence of Clairvoyant and Effectful Semantics

In this section, we show that derivations in the effectful semantics correspond to derivations in the clairvoyant semantics. To help with our proof in the previous section, we used a clairvoyant heap as an upper bound for all heaps that appeared in a derivation. In order to apply the same idea to the effectful semantics, we need to define an ordering on contexts. We define relations $\leq$ on denotational values and contexts corecursively as follows:

- $\Gamma_1 \leq \Gamma_2$ if, whenever $\Gamma_1(x)$ is defined, so is $\Gamma_2(x)$, and in this case, $\Gamma_1(x) \leq \Gamma_2(x)$;
- $\lambda^{\Gamma_1} x.\, e \leq \lambda^{\Gamma_2} x.\, e$ whenever $\Gamma_1 \leq \Gamma_2$;
- $(x \mapsto w_1) \leq (x \mapsto w_2)$ whenever $w_1 \leq w_2$.

The fact that our definition is corecursive essentially means that a derivation tree used to prove $\Gamma_1 \leq \Gamma_2$ or $w_1 \leq w_2$ is allowed to be infinite. Now, we call a context Γ *small* if $\Gamma \leq \mathrm{tr}(H)$ for some heap H. This is a kind of finiteness condition on contexts: even though a context may be an infinite structure, its behaviour can be imitated by a finite heap. Using this notion, we can state a kind of immutability result for contexts analogously to lemma 5.3:

Lemma 5.6 (context immutability). *Suppose that $\Gamma \vdash e \Downarrow (l)\ q : w$, where l has no duplicates and is fresh for Γ, and Γ is small. Then there is a heap C such that whenever a mapping $(x \mapsto w)$ occurs anywhere in the given derivation tree, we have $C(x) = \overline{w}$.*

By a mapping $(x \mapsto w)$ 'occurring', we mean that there is a context Γ' somewhere in the derivation tree such that $\Gamma'(x) = w$, or that the denotational value $(x \mapsto w)$ arises somewhere in the derivation, even inside a context or another denotational value. In particular, the conclusion of the lemma implies that $\Gamma \leq \mathrm{tr}(C)$.

Proof. We define C by the given property: if $(x \mapsto w)$ occurs anywhere, we define $C(x) = \overline{w}$, and otherwise $C(x)$ is undefined. It remains to check that C is a well-defined function and that its domain is finite.

Suppose that the mappings $(x \mapsto w)$ and $(x \mapsto w')$ both appear in the derivation tree. As Γ is small and l is fresh for it, we have $\Gamma \leq \mathrm{tr}(H)$ for some H with domain disjoint from l. Then, there are two cases: either $x \in \mathrm{dom}\, H$ or $x \in l$.

If $x \in \mathrm{dom}\, H$, then it is easy to see that $\overline{w} = \overline{w'} = H(x)$. If $x \in l$, then the two bindings were introduced in a branch of the derivation tree rooted at an application rule introducing the name x, and since l has no duplicates, these two roots coincide. So $w = w'$.

Therefore, C is well-defined. Finally, as the only names that can be domain elements of C are either in $\mathrm{dom}\, H$ or l, the function C has finite domain as required. □

Proposition 5.7 (equivalence of clairvoyant and effectful semantics). *Suppose that Γ is a small context defined on the free variables of e, and that the contexts Γ' in every closure $\lambda^{\Gamma'} x.\, e'$ appearing in Γ are defined on the free variables of e' other than x. Suppose further that l is a list of names without duplicates, fresh for Γ and e. Then we have the following equivalence:*

$$(\exists w.\ \overline{w} = v\ \wedge\ \Gamma \vdash e \Downarrow (l)\ q : w) \iff (\exists C\, t.\ \Gamma \leq \mathrm{tr}(C)\ \wedge\ \overline{t} = q\ \wedge\ C : e \Downarrow (l)\ t : v)$$

Proof. In the forward direction, suppose that $\Gamma \vdash e \Downarrow (l)\ q : w$. Setting C to be a clairvoyant heap given by lemma 5.6, the right-hand side follows directly by rule induction.

In the backward direction, suppose that $C : e \Downarrow (l)\ t : v$, where $\Gamma \leq \mathrm{tr}(C)$ and $\overline{t} = q$. For this direction, we prove by rule induction that $\Gamma \vdash e \Downarrow (l)\ q : w$ for some w with $\overline{w} = v$. Throughout this induction, we maintain the invariant that whenever a binding $(x \mapsto w)$ occurs, we have $C(x) = \overline{w}$. Additionally, we ensure throughout this induction that every closure $\lambda^{\Gamma'} x.\, e'$ that appears has the property that Γ' is defined on the free variables of e' other than x. We show the case for the application rule here; the cases for the other rules are trivial.

$$\frac{C : e_1 \Downarrow (l_1)\ t_1 : \lambda x.\, e \qquad C : e_2 \Downarrow (l_2)\ t_2 : C(x) \qquad C : e \Downarrow (l_3)\ t_3 : v}{C : e_1\ e_2 \Downarrow (l_1 \diamond l_2 \diamond [x] \diamond l_3)\ t_1 \cdot t_2 \cdot (x := C(x)) \cdot t_3 : v}\ \textsc{C-App}$$

First, as Γ is defined on the free variables of e_1 and e_2, we can use the inductive hypothesis to obtain the following judgments, where $\overline{w} = C(x)$:

$$\Gamma \vdash e_1 \Downarrow (l_1)\ \overline{t_1} : \lambda^{\Gamma'} x.\, e \qquad\qquad \Gamma \vdash e_2 \Downarrow (l_2)\ \overline{t_2} : w$$

Now, we know by our invariant and inductive hypothesis that $\Gamma', x \mapsto w \leq \mathrm{tr}(C)$. This new context is defined on the free variables of e, so we may apply the inductive hypothesis again to yield

$$\Gamma', x \mapsto w \vdash e \Downarrow (l_3)\ \overline{t_3} : w'$$

where $\overline{w'} = v$, as required. □

5.5 Completing the Proof

We now have all of the tools needed to prove the equivalence theorem.

Theorem 5.1 (equivalence of semantics). *Let l be a list of names without duplicates, fresh for H and e. Suppose additionally that H is closed and contains all free variables of e. Then:*

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

Proof. By assumption, H satisfies the hypotheses for proposition 5.5. Additionally, $\mathrm{tr}(H)$ satisfies the hypotheses for proposition 5.7 as H is closed and contains the free variables of e in its domain. Therefore, we can apply propositions 5.5 and 5.7 to reduce the required result to

$$\begin{aligned} &(\exists C \supseteq H, && \exists t.\ t(H) \text{ is defined} && \wedge\ C : e \Downarrow (l)\ t : v) \\ \iff &(\exists C.\ \mathrm{tr}(H) \leq \mathrm{tr}(C) \wedge && \exists t.\ \overline{t} \text{ is defined} && \wedge\ C : e \Downarrow (l)\ t : v) \end{aligned}$$

Since H is closed, we have $C \supseteq H \iff \mathrm{tr}(H) \leq \mathrm{tr}(C)$. It therefore suffices to show that in this case, whenever $C : e \Downarrow (l)\ t : v$,

$$t(H) \text{ is defined} \iff \overline{t} \text{ is defined}$$

In the forward direction, as $t(H)$ is defined, it must contain no instance of a use of a name after it is freed, so $\overline{t}$ must be defined. Conversely, if $\overline{t}$ is defined, there is no use after free of any name. Due to our freshness conditions, all assignments appearing in t are distinct and fresh for H, and any uses of variables not in $\mathrm{dom}\, H$ always occur after their assignment. Therefore, $t(H)$ is defined. □

6 Related Work

In this section we survey a selection of related work on explicit operations, first-class names, immutable references, memory management, and effect structures.

Explicit operations in lambda calculi. In explicit naming, we move the action of reading from a name from the metatheory into the language itself. The idea of moving metatheoretic operations into the object language is not new. A key example is *explicit substitutions* [Abadi et al. 1989], in which the substitutions generated by the β-rule in a lambda calculus are carried out explicitly in evaluation steps, rather than all at once in the metatheory. Similarly, sharing is made explicit in the *atomic lambda calculus* [Gundersen et al. 2013] by providing a new kind of expression that binds the same term to multiple names. In both of these examples, a key goal is to bridge the theory of the lambda calculus with practical implementations of functional languages, since the latter cannot implement the usual lambda calculus semantics directly due to performance considerations.

Names as first-class citizens. The *ν-calculus* [Pitts and Stark 1993] is a simply typed lambda calculus that has both λ-bound variables and a notion of ν-bound *local names*. In this calculus, a name is an opaque object that can be compared for equality, and nothing else. Similarly to explicit naming, names are first-class citizens and evaluate to themselves. In contrast, however, names in the ν-calculus hold no information other than their identity; operationally, names behave as pointers to the unit type, and can be compared for pointer equality. Another similar system is the *$\lambda\nu$-calculus* [Odersky 1994]. This has similar syntax to the ν-calculus, but its semantics aims to more closely resemble the usual λ-calculus, as opposed to the operational behaviour of dynamic name allocation. First-class names have also seen practical use in *FreshML* [Shinwell et al. 2003]. In this system, names are a user-defined type, and the language provides the ability to define binding operations over such names. These systems are discussed in more detail in [Pitts 2013].

Immutable shared references. Our proof that the heap-based semantics is equivalent to our compositional effect-based semantics (theorem 4.1) crucially relies on the fact that shared references are immutable. This assumption naturally holds for pure functional languages like Haskell, where every value is immutable (outside of a stateful monad such as `IO`). However, it is also a common theme in semantics research even for imperative languages, since it allows for powerful invariants on the memory accessible by a program. Examples of this theme include [Huang et al. 2012; Pechtchanski and Sarkar 2002; Tschantz and Ernst 2005], which collectively aim to introduce reference immutability into object-oriented languages such as Java. In one application [Gordon et al. 2012], immutable references are exploited to provide abstractions for safe parallelism in the presence of aliasing. Immutability of shared references is a core part of the type system of Rust [Klabnik and Nichols 2023; Matsakis and Klock 2014], where it is framed as *aliasing* XOR *mutability*. The invariants obtained under this restriction can be used to prove soundness and safety properties of programs written in Rust. Formal developments surrounding the Rust language specifically include the *RustBelt* project [Jung et al. 2017] and *Oxide* [Weiss et al. 2019].

Memory management. Research on memory management systems has a long history. One main development motivating our ideas is *region-based memory management* [Tofte et al. 2004; Tofte and Talpin 1997]. In this system, all allocations are placed into a *region* defined by a scope, and at the end of such a scope, all allocations in this region are freed. Our e_1; *free* e_2 construct can be thought of as a variant of this, disposing of a single name at the end of the scope of e_1. The *capability calculus* [Crary et al. 1999] is a variant of this idea that tracks what regions are used while evaluating a certain expression. A type-correct expression in this system never accesses a region that has already been freed. *Islands* [Hogg 1991] are another related idea, which can be used to provide non-aliasing guarantees for particular objects.

Type-based approaches to memory management. Type systems play an important rule in modern approaches to memory management semantics. We were inspired by the following work in this area, although our work takes place in an untyped setting. *Uniqueness types* [Barendsen and Smetsers 1993] and *ownership types* [Clarke et al. 1998] are methods of aliasing protection that have been used in languages like Rust to enable predictable behaviour of memory. A graded extension of uniqueness types, called *fractional uniqueness* types, have been used to encode ownership and borrowing in the functional language Granule [Marshall and Orchard 2024]. An ownership-like system can also be modelled using *reachability types* [Bao et al. 2021; Wei et al. 2024], using an effect system to determine which objects are reachable from which others. Their main technical tool is *kill effects*, which disable future accesses to a value, analogously to our name freeing construct. In fact, their *effect labels with kill* can be viewed as a variant of our name effects from section 3.2.

Effect structures. Many of the approaches to memory management discussed above use effect systems to track validity of computations, which were first introduced in [Gifford and Lucassen 1986; Lucassen and Gifford 1988]. Effect systems can often be characterised as an instance of a general algebraic structure, such as an effect quantale [Gordon 2021]. This was the notion of effect structure that we chose to work with in this paper due to its ability to represent noncommutative effects without much complexity. There are several other alternatives that have been proposed for noncommutative effects, such as *Kleene algebras* [Kozen 1994]. In such a system, the additive unit 0 corresponds to an invalid effect, and the operators are total; conversely, with effect quantales, there is no designated undefined effect but the join and sequencing operators $\sqcup$ and $\blacktriangleright$ may be partial. Another notable example of such an algebraic structure is that of *preordered monoids* [Katsumata 2014], which is a preorder with a monotone monoid operation representing sequential composition. These can be considered a generalisation of both effect quantales and Kleene algebras, where neither joins of effects nor repetition of an effect is defined in general.

7 Conclusions and Future Work

We have introduced an explicit naming system in which names are first-class citizens, and are manipulated using explicit operations for creating, using and freeing names. The traditional heap-threading semantics is not compositional, but is equivalent to an effect-based semantics that is better suited for reasoning about program behaviour. For example, we were able to show easily that dead code elimination is a valid code transformation for the heap-based semantics, by using the equivalence to translate the problem into the effect-based semantics.

Our equivalence proof makes use of a ‘clairvoyant’ semantics, which does not correspond to a real evaluation strategy that can be carried out in practice, but can naturally express the behaviour of both the heap-based and effect-based approaches. This allowed us to avoid comparing the two semantics directly, which was a significant technical convenience.

This work suggests many potential directions of future study. First of all, it would be interesting to generalise our effect system to allow other effects, such as exceptions or mutability. We could also consider a feature for ‘masking’ externally unobservable effects [Lucassen and Gifford 1988], such as locally manipulating a name that is inaccessible to the rest of a program. Next, we note that a heap-threading evaluation function can be viewed as a monadic function using the state monad on heaps, whereas an effect-based evaluation function can be viewed as a map using a composite of a reader and a writer monad: the context used for evaluation forms the reader part, and the effect produced by an expression forms the writer part. In light of this, our equivalence theorem can be viewed as describing a translation of a program from the state monad to a reader-writer monad, and it would be interesting to investigate whether this idea can be carried out in other settings. And finally, we are interested in developing type systems for explicit naming in which type-correct programs never attempt to read from a freed name, because this would allow us to better understand type systems for safe memory management.

Acknowledgements

This work was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/Y010744/1, *Semantics-Directed Compiler Construction: From Formal Semantics to Certified Compilers*, for which funding is gratefully acknowledged. A preliminary version of this talk was presented at TYPES 2025, and we are grateful for the feedback that we received.

References

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. 1989. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 31–46. https://doi.org/10.1145/96709.96712

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

Erik Barendsen and Sjaak Smetsers. 1993. Conventional and uniqueness typing in graph rewrite systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer Berlin Heidelberg, 41–51.

David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 48–64. https://doi.org/10.1145/286936.286947

Karl Crary, David Walker, and Greg Morrisett. 1999. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 262–275. https://doi.org/10.1145/292540.292564

David K. Gifford and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 4 (April 2021), 79 pages. https://doi.org/10.1145/3450272

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 21–40. https://doi.org/10.1145/2384616.2384619

Tom Gundersen, Willem Heijltjes, and Michel Parigot. 2013. Atomic Lambda Calculus: A Typed Lambda-Calculus with Explicit Sharing. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science* (New Orleans, LA, USA). IEEE, 311–320. https://doi.org/10.1109/LICS.2013.37

Jennifer Hackett and Graham Hutton. 2019. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.* 3, ICFP, Article 114 (July 2019), 23 pages. https://doi.org/10.1145/3341718

John Hogg. 1991. Islands: Aliasing Protection in Object-Oriented Languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*. ACM, Phoenix Arizona USA, 271–285. https://doi.org/10.1145/117954.117975

Wei Huang, Ana Milanova, Werner Dietl, and Michael D Ernst. 2012. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. *ACM SIGPLAN Notices* 47, 10 (2012), 879–896.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego California USA). ACM, New York, NY, USA, 633–645. https://doi.org/10.1145/2535838.2535846

Steve Klabnik and Carol Nichols. 2023. *The Rust programming language.* No Starch Press.

D. Kozen. 1994. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. *Information and Computation* 110, 2 (1994), 366–390. https://doi.org/10.1006/inco.1994.1037

John Launchbury. 1993. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 144–154. https://doi.org/10.1145/158511.158618

John M. Lucassen and David K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

Danielle Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 131 (April 2024), 31 pages. https://doi.org/10.1145/3649848

Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Martin Odersky. 1994. A functional theory of local names. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 48–59. https://doi.org/10.1145/174675.175187

Igor Pechtchanski and Vivek Sarkar. 2002. Immutability specification and its applications. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande* (Seattle, Washington, USA) *(JGI '02)*. Association for Computing Machinery, New York, NY, USA, 202–211. https://doi.org/10.1145/583810.583833

Simon Peyton Jones, Simon Marlow, and Conal Elliott. 2000. Stretching the storage manager: weak pointers and stable names in Haskell. In *Proceedings of the 11th International Workshop on the Implementation of Functional Languages* (proceedings of the 11th international workshop on the implementation of functional languages ed.) *(LNCS)*, Pieter Koopman and Chris Clack (Eds.). Springer Berlin Heidelberg, 37–58. https://www.microsoft.com/en-us/research/publication/stretching-the-storage-manager-weak-pointers-and-stable-names-in-haskell/

Andrew M Pitts. 2013. *Nominal sets: Names and symmetry in computer science.* Cambridge University Press, USA.

Andrew M. Pitts and Ian D. B. Stark. 1993. Observable properties of higher order functions that dynamically create local names, or: What's new?. In *Mathematical Foundations of Computer Science 1993*, Andrzej M. Borzyszkowski and Stefan Sokołowski (Eds.). Springer Berlin Heidelberg, 122–141.

Alastair Reid. 1994. Malloc Pointers and Stable Pointers: Improving Haskell's Foreign Language Interface. In *Draft Proceedings of the Glasgow Functional Programming Workshop.* Ayr, Scotland.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. 2003. FreshML: programming with binders made simple. *SIGPLAN Not.* 38, 9 (Aug. 2003), 263–274. https://doi.org/10.1145/944746.944729

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17 (2004), 245–265.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: adding reference immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 211–230. https://doi.org/10.1145/1094811.1094828

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 393–424. https://doi.org/10.1145/3632856

Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2019. Oxide: The Essence of Rust. https://doi.org/10.48550/ARXIV.1903.00982