Substitution, jumps, and algebraic effects

Marcelo Fiore
Computer Laboratory, University of Cambridge

Sam Staton
ICIS, Radboud University Nijmegen

Abstract
Algebraic structures abound in programming languages. The starting point for this paper is the following theorem: (first-order) algebraic signatures can themselves be described as free algebras for a (second-order) algebraic theory of substitution. Transporting this to the realm of programming languages, we investigate a computational metalanguage based on the theory of substitution, demonstrating that substituting corresponds to jumping in an abstract machine. We use the theorem to give an interpretation of a programming language with arbitrary algebraic effects into the metalanguage with substitution/jumps.

Categories and Subject Descriptors [Theory of computation]: Semantics and reasoning.

1. Introduction
In studying computational effects for functional programming languages, it is appropriate to distinguish between what we will call actual and virtual effects. This paper is about the relationship between the two.

• By actual effects we mean extensions to a pure functional language that permit access to its (abstract) machine. In this sense SML/NJ has many actual effects such as memory access, network primitives and access to the control stack.

• By virtual effects we mean a style of programming that has the appearance of performing actual effects on an abstract machine, but where the effects are in reality handled within the pure language. For example, one can use a state monad in Haskell to write Haskell programs that appear to directly access memory without actually requiring a memory management unit in the abstract G-machine.

We make two main contributions:
1. We give a novel denotational semantics for the actual effects involving code pointers and jumping, based on a mathematical theory of substitution.
2. We give a translation from a class of virtual effects into the actual effects of jumping. This is motivated and justified by the following basic observations:

• virtual effects can be formalized in terms of algebraic signatures;
• algebraic signatures are exactly the free models of the theory of substitution.

We thus derive a semantic explanation of current programming practice from a fundamental mathematical result.

1.1 Virtual effects and algebraic signatures
Consider an algebraic signature with one binary operation \( \oplus \). Terms in the signature are binary trees with branches labelled \( \oplus \) and leaves labelled by free variables. We might think of such a term as a very simple program with virtual effects: a binary decision tree.

\[
(x \oplus y) \oplus z \quad i.e.
\]

\[
\begin{array}{c}
\text{x} \oplus \text{y} \oplus \text{z} \\
\end{array}
\]

Indeed, think of \( \oplus \) as meaning ‘read from a fixed boolean memory cell; if true then branch left, if false then branch right’. Or think of \( \oplus \) as meaning ‘read from a stream of booleans’, or as ‘make a probabilistic choice’, or as an undetermined boolean in a non-deterministic computation. The leaves (\( x, y, z \)) are thought of as pointers to the continuation of the computation.

Thus a program involving virtual effects is typically decomposed into two parts: first the main program, which creates a computation tree rather than actually performing the effects; secondly an auxiliary mechanism that ‘runs’ the virtual effects by processing the tree.

The tree is a tree of computations and not a tree of data, so it is reasonable to treat it differently. Informally, we can think of the edges as code pointers rather than as edges in a concrete datatype. (Efficient ML implementations in this vein do this by manipulating the control stack (e.g. [2,25]; Haskell implementations use free monads and laziness (e.g. [17, 25, 27, 45]).)

1.2 Substitution and the actual effects of code pointers
The terms for an algebraic signature have additional algebraic structure given by substituting a term for a free variable. Indeed, the computation tree (1) can be written using substitution instead of nested terms: \((x \oplus y) \oplus z = (a \oplus z)[x[\oplus y/a]]\). To this end we study substitution as a second-order algebraic theory, following [15] (also [39], [13], [28], [41]). It comprises two operations, \( \text{sub} \) and \( \text{var} \). It involves a basic type \( \ell \), which might be thought of as a type of variables or indices, and the following term formation rules:

\[
\begin{align*}
\Gamma, a : \ell &\vdash t \\
\Gamma &\vdash u \\
\Gamma &\vdash t : \ell
\end{align*}
\]

\[
\frac{
\Gamma \vdash \text{var} t \\
\Gamma, a : \ell \vdash u \\
\Gamma, a : \ell \vdash \text{sub}(a.t, u)
}{
\Gamma \vdash \text{sub}(a.t, u)
}\]

Informally, we think of \( \text{sub}(a.t, u) \) as ‘substitute \( u \) for each occurrence of \( \text{var} a \) in \( t \)’. We work up to \( \alpha \)-renaming in \( \ell \) and equations such as \( \text{sub}(a.\text{var} a, u) \equiv u \). We have the following theorem (The-
A suitable computational reading of the type \( \ell \) as describing effects. We are thus led to consider substitution itself as an actual effect, one that in some ways subsumes the virtual ones. A suitable computational reading of the type \( \ell \) is as a type of labels or code pointers, so that \( \text{sub}(a,t,u) \) can be read as 'create a new code pointer to \( u \), bind it to \( a \) and continue as \( t' \), and \( \text{var} \) a can be read as 'jump to label \( a \)'.

1.3 Translating virtual effects into code pointers

We can now understand a computation that involves the virtual effect \( \odot \) as a computation of type \( (\ell \times \ell) \) that involves the actual effects of substitution/jumps. The computation tree (1) can be written in our metalanguage as \( \text{sub}(a,\text{return}(a,z),\text{return}(x,y)) \). It returns the pair of labels \((a,z)\), where label \( a \) is a pointer to the computation that returns the pair \((x,y)\) of labels.

For another example without dangling pointers, consider the computation \( (\text{return}\ tt \odot \text{return}\ ff) : \text{bool} \) which returns either \( tt \) or \( ff \) depending on the outcome of the virtual effect \( \odot \). This is translated into our metalanguage as

\[
\text{sub}(a,\text{sub}(b,\text{return}\ \text{inr}(a,b)),\text{return}\ \text{inl}(tt)) : \text{bool} = (\ell \times \ell)
\]

The type \( (\text{bool} \odot (\ell \times \ell)) \) is inhabited by programs that either immediately return a boolean \( (\text{in} v) \) or that return a pair of pointers \((\text{inr}(a,b))\) describing how to continue according to the outcome of the virtual effect \( \odot \).

1.4 Contributions

In summary, our main contributions are as follows:

- a typed metalanguage based around the theory of substitution/jumps (§2), with an abstract machine and an adequate denotational semantics (§3);
- a sound translation of a language with virtual effects into the metalanguage with substitution (§5), based on the fact that algebraic signatures are free substitution algebras.

We also develop the following advanced topics more briefly:

- semantics of effect handlers (§6), which are (roughly) a mechanism for manipulating trees such as (1);
- the addition of other effects to the theory of substitution (§7). Adding a stack of code pointers to our abstract machine amounts to extending the theory of substitution with the \( \beta \)-equality of the untyped \( \lambda \)-calculus (§7.2).

2. Substitution and actual code pointers

We introduce a metalanguage which extends Moggi’s monadic language [30, 34] with substitution/jumps.

2.1 Algebraic presentation of substitution

The theory of substitution can be presented as an equational theory in typed lambda calculus ([15, Def. 3.1], [11, §B]). It has two types, \( \ell \) and \( \ell' \), an operation \( \text{sub}(: (\ell' \times \ell) \times \ell \rightarrow \ell \) and an operation \( \text{var} : \ell \rightarrow \ell \). The theory has four equations:

\[
x : \ell' \Rightarrow \text{sub}(\ell) \Rightarrow x
\]

\[
x : \ell', y : \ell \Rightarrow \text{sub}(\ell) \Rightarrow x
\]

\[
a : \ell, x : \ell' \Rightarrow \text{sub}(\ell) \Rightarrow a
\]

\[
x : \ell \times \ell \Rightarrow x, y : \ell \Rightarrow z : \ell \Rightarrow \text{sub}(\ell) \Rightarrow x
\]

The idea is that \( \ell \) is a type of terms, \( \ell' \) is a type of variables (or ‘labels’), and \( \text{var} \) includes the variables in the terms and \( \text{sub}(\ell) \) substitutes \( u \) for \( a \) in \( \ell \).

One can also read \( \text{sub}(\ell) \) as ‘bind \( u \) to \( a \) in \( \ell \)’, and \( \text{var} \) as ‘jump to \( a \)’. Indeed, the theory of substitution is a fragment of Thielecke’s CPS calculus [44, §2.1] with the restriction that the jumps take no parameters. The four axioms for substitution are essentially the four axioms of that calculus.

2.2 Metalanguage

Types The types of the metalanguage are

\[A, B ::= \ell \mid A \rightarrow B \mid A_1 \ast \cdots \ast A_n \mid A_1 + \cdots + A_n \quad (n \in \mathbb{N})\]

We have a special type \( \ell \) of labels. As a special case of the \( n \)-ary product type when \( n = 0 \) we have the type unit. We decorate the arrow of the function type \( (\rightarrow) \) to emphasise that the functions are not pure, they might contain substitution effects.

Typed terms We have two typing judgements: one for pure computations \( (\vdash) \) and one for computations with substitution effects \( (\vdash_\ell) \). The terms in context are defined as follows. Firstly we have rules for sums and products of pure computations:

\[
\Gamma, x : A, \Gamma' \vdash x : A \quad \Gamma \vdash \mathbf{\#} t : A_1 \\
\Gamma \vdash t : A_1 + \cdots + A_n \quad \Gamma \vdash (t_1 \ldots t_n) : A_1 \ast \cdots \ast A_n
\]

Secondly standard term formation for sequencing and returning in a call-by-value-like language:

\[
\Gamma \vdash_\ell t : A \quad \Gamma, x : A \vdash u : B \quad \Gamma \vdash t : A
\]

Thirdly the specific term formation rules for the metalanguage:

\[
\Gamma \vdash_\ell \text{let val } x = t \text{ in } u : B \quad \Gamma \vdash_\ell \text{return } t : A
\]

And finally function abstraction and application. In the call-by-value tradition, functions freeze (‘thunk’) computational effects.

\[
\Gamma, x : A \vdash_\ell t : B \\
\Gamma \vdash t : A \rightarrow_\ell B \quad \Gamma \vdash u : A
\]

We will use simple syntactic sugar. We write ‘bool’ for the type unit + unit, and ‘tt’ and ‘ff’ for \( \text{inj}_1() \) and \( \text{inj}_2() \) respectively. We write \( (t_1 u_1 \ldots u_n) \) for (case \( t_1 \text{ of } \text{inj}_1() \Rightarrow u_1 \text{ } \text{inj}_2() \Rightarrow u_2 \), and we write \( (t ; u) \) instead of \( (\text{let val } t = t \text{ in } u) \). We only included rules for case on pure terms, but they can be derived for computation terms too, using pure terms and functions (e.g. [33, §2]). We sometimes write \( (t u v) \) for \( (\text{let val } t = (t u) \text{ in } (fv)) \), and so on.

Equational theory We have an equality judgement on pure typed terms, \( \Gamma \vdash t =_\ell u : A \) and an equality judgement on effectful terms, \( \Gamma \vdash_\ell t =_\ell u : A \). The judgements are generated as follows:

- equality is reflexive, symmetric, transitive and substitutive;
- on pure terms we include the standard \( \beta/\eta \) laws:

\[
\#i (t_1 \ldots t_n) \equiv t_1 \quad t \equiv \lambda t_1 \ldots \lambda t_n t
\]

\[
\text{case inj}_0 t \text{ of } \ldots \text{inj}_0 (x_i) \Rightarrow u_0 \ldots \equiv u_i [t_i/_{x_i}]
\]

\[
\text{case } t \text{ of } \ldots \text{inj}_n (x_i) \Rightarrow u [t_i/_{x_i} \ldots] \equiv u_i[t_i/_{x_i}]
\]
• on effectful terms we have the following standard laws [30, 34]:

\[
\begin{align*}
\text{let} \ val \ x &= \ return \ t \ in \ u \equiv u[i^t_{/u}] \quad t \equiv \text{let} \ val \ x &= \ t \ in \ return \ x \\
\text{let} \ val \ x &= \ t \ in \ (\text{let} \ y = u \ in \ w) \\
& \equiv \text{let} \ y = (\text{let} \ val \ x = t \ in \ u) \ in \ w \ (y \not\in \text{fv}(t)) \\
\text{let} \ fn \ x & \Rightarrow (t \ u) \equiv t[i^u_{/u}] \\
\text{let} \ fn \ x \Rightarrow (t \ x) \ (x \not\in \text{fv}(t))\end{align*}
\]

(3)

• the judgement \( \Gamma \vdash t \equiv u : A \) includes all the equations for substitution (§2.1); together with two ‘algebraicity’ equations [40], which propagate the effects:

\[
\begin{align*}
\text{let} \ val \ x &= \ sub(a.t, u) \ in \ w \\
& \equiv \ sub(a.(\text{let} \ val \ x = t \ in \ w), let \ val \ x = u \ in \ w) \\
\text{let} \ val \ x &= var(t) \ in \ w \equiv var(t)
\end{align*}
\]

Informal semantics We may think of \( \ell \) as a type of code pointers or labels, so that sub(a.t, u) creates a new label \( a \), and continues as \( t \); if and when var(a) is called, the program jumps back to the point where \( a \) was created, and continues as \( u \) instead. The type of labels \( \ell \) is thus like a type of statically-scoped exceptions, as used for instance in Herbelin’s study of Markov’s principle [21]. We can think of sub(a.t, u) as installing a new statically-scoped exception \( a \) with handler \( u \), and var(a) as raising the exception.

The reader should note that sub(a.t, u) as textual substitution of \( u \) for \( a \) in \( t \) is a meta-operation that is typically nonsense in this context, and indeed it is inconsistent with the algebraicity equations. (The construct sub(a.t, u) can however perhaps be thought of roughly as binding \( u \) with the current continuation to \( a \) in \( t \).)

Much has been made of the relationship between control effects and classical logic. We make a few remarks on this topic. In some ways the type \( \ell \) behaves like “–unit”. The operation \( var \) is like negation elimination. It is often helpful to work with the ‘generic effects’ that are associated to each algebraic operation [37]; the generic effect of sub is

\[
\text{fn}_- \Rightarrow sub(a.\text{return inj}_1(a), \text{return inj}_2(\_)) : \text{unit} \to \ell \ {\text{+}} \ \text{unit}
\]

(NB we do not mean the textual substitution of \( a \) by \( \text{return inj}_2(\_ \_ a) \) in \( \text{return inj}_1(a) \), which does not make sense.) This generic effect is like the law of the excluded middle “–unit \( \lor \) \text{unit}”. Recall a computational intuition for the excluded middle (e.g. [6],[20, §3.1.4]): it first introduces the left hand disjunct (“–unit”); but if the subsequent proof eliminates this negation at some point then the whole proof backtracks to the disjunction which introduces the right hand disjunct (unit) instead. (We do not claim to have a full model of classical logic: it is a subtle model that contains some aspects of classical logic.)

3. Semantics of the metalanguage

3.1 Denotational semantics of the metalanguage

3.1.1 Context-indexed sets

The theory of substitution (§2.1) is not a classical algebraic theory, but rather a parameterized algebraic theory in the sense of [41, 42], i.e., a second-order algebraic theory [13, 14] in which the exponents are all of a special kind. As such it is not well-suited to naive set-theoretic models, since there is no canonical set for interpreting \( \ell \) (e.g. [41, §VI-C]). To resolve this we consider sets indexed by contexts of labels \([a_1; \ell : \cdot \cdot \cdot a_n; \cdot \ell]\). To abstract away from the choice of names for labels, we take a context to be a natural number \( n \) considered as a set with \( n \) elements \( \{1 \ldots n\} \).

Definition 1. A context-indexed-set \( P \) is given by, for each natural number \( n \), a set \( P(n) \), and for each function \( f : m \to n \), a function \( Pf : P(m) \to P(n) \), such that identities and composition are respected. That is, a context-indexed-set is a functor \( P : \text{Ctx} \to \text{Set} \), where \( \text{Ctx} \) is the category of natural numbers (considered as sets) and functions between them.

A context-indexed-set \( P \to Q \) is given by a natural family of functions \( \{P(n) \to Q(n)\}_n \).

The objects \( n \) of \( \text{Ctx} \) should not be thought of as arbitrary contexts of the metalanguage, but rather as contexts of labels. The category of context-indexed-sets is an extension of the tiny type theory \((\ell, \times)\), which has nothing but finite products and a distinguished type \( \ell \), to a model of intuitionistic higher-order logic. More precisely, the category \( \text{Set}^{\text{Ctx}} \) of context-indexed-sets is the free cocompletion of \( \text{Ctx}^{op} \), which is equivalent to the syntactic category of \((\ell, \times)\).

We now concretely explain the structure of the category of context-indexed-sets as a basis for our denotational semantics. (See also [15, 41].)

• The product of context-indexed-sets satisfies

\[
(P_1 \times \cdots \times P_k)(n) \cong P_1(n) \times \cdots \times P_k(n).
\]

• The sum of context-indexed-sets satisfies

\[
(P_1 \uplus \cdots \uplus P_k)(n) \cong P_1(n) \uplus \cdots \uplus P_k(n).
\]

• There is a distinguished context-indexed-set \( L \), given by \( L(n) = n \). The Yoneda lemma provides a natural family of bijections \( P(n) \cong \text{Set}^{\text{Ctx}}(L^n, P) \).

• The category is cartesian closed: there is a context-indexed-set \( P^c \) of context-indexed-functions that satisfies

\[
Q^c(n) \cong \text{Set}^{\text{Ctx}}(P \times L^n, Q).
\]

• In particular the function space \((Q^{L^n}) \) has a natural family of bijections \((Q^{L^n})(m) \equiv Q(m + n) \). So exponentiation by powers of \( L \) is a kind of context extension.

3.1.2 Substitution algebras

We can now consider models of the theory of substitution (§2.1) in the category of context-indexed-sets, interpreting \( \ell \) as \( L \). In more detail, a substitution algebra is a context-indexed-set \( P \) equipped with a family of functions \( \text{sub}_n : P(n + 1) \times P(n) \to P(n) \) and \( \text{var}_n : L(n) \to P(n) \) satisfying naturality requirements and the four equations for substitution [15, Def 3.1]. A homomorphism of substitution algebras \( P \to Q \) is a context-indexed-function that respects the additional substitution structure.

Every context-indexed-set \( P \) admits a free substitution algebra \( SP \), i.e. a substitution algebra \( SP \) together with a context-indexed-function \( \eta : P \to SP \) such that for any substitution algebra \( Q \) and any context-indexed-function \( f : P \to Q \) there is a unique homomorphism \( f^\eta : SP \to Q \) such that \( f = f^\eta \cdot \eta \). This free substitution algebra can be built in a standard way [42, Thm. 2, Prop. 2] by inductively adding sub and var and then quotienting by the equations [19]. This syntactic construction provides a completeness result [41, Prop. 8]: an equation is derivable in the second order theory of substitution if and only if it holds in all substitution algebras.

The free substitution algebra construction \( S \) yields a strong monad on the category of context-indexed-sets, and then quotienting by the equations [19]. We will use the following two characterizations of the monad \( S \) in this paper: as a Kan extension of the construction of terms for a signature (Thm. 9), and as a free monoid (6) for a substitution tensor product.
3.1.3 Denotational semantics

We use the monad S to give a denotational semantics for our metalanguage, essentially following Moggi’s pattern [30, 34]. We interpret types as context-indexed-sets:

\[
\begin{align*}
\lbrack \ell \rbrack & \overset{\text{def}}{=} \mathbb{L} \\
\lbrack A_1 \times \ldots \times A_n \rbrack & \overset{\text{def}}{=} \lbrack A_1 \rbrack \times \ldots \times \lbrack A_n \rbrack \\
\lbrack A \rightarrow B \rbrack & \overset{\text{def}}{=} (\lbrack B \rbrack)^{\lbrack A \rbrack}
\end{align*}
\]

Pure terms (\(\vdash\)) and effectful terms (\(\vdash e\)) are interpreted as context-indexed-functions:

\[
\begin{align*}
\Gamma \vdash t : A & \implies \Gamma \rightarrow S[A] \\
\Gamma \vdash u : A & \implies \Gamma \rightarrow S[A]
\end{align*}
\]

This interpretation is defined by induction on the structure of derivations as usual. For instance,

\[
\begin{align*}
\lbrack \text{let } x = t \text{ in } u \rbrack(\rho) & \overset{\text{def}}{=} \lbrack t \rbrack(\rho) \Rightarrow \lambda x. \lbrack u \rbrack(\rho, x)
\end{align*}
\]

(\(\text{in the internal language of } \text{Set}^{\text{Ctx}}\)). We define \(\lbrack \text{var}(t) \rbrack\) and \(\lbrack \text{sub}(a.t, u) \rbrack\) using the substitution algebra structure of the monad S.

Proposition 2. The semantics of the metalanguage is sound: if \(\Gamma \vdash s t : A\) is derivable in the equality theory, then \(\lbrack s \rbrack = \lbrack u \rbrack : \lbrack \Gamma \rbrack \rightarrow S[A]\).

3.2 An abstract machine

We now cement the computational intuitions about the metalanguage by describing an abstract machine. The machine is similar to what Felleisen and Friedman called a CK-machine (e.g., [8, 29]). However, since we make heavy use of code pointers it is natural to use a tree of evaluation frames rather than a stack. The idea of using a heap instead of a stack is certainly not a new one, e.g. [11].

Configurations Consider a finite set \(\{a_1, \ldots, a_n\}\) of labels. A code-heap labelled by \(\{a_1, \ldots, a_n\}\) is given by the following data:

- A finite forest \(G\), i.e. a set \(\mathcal{G}\) of nodes together with a partial function \(\text{succ} : G \rightarrow G\) whose graph is acyclic. We say \(g\) is a root if \(\text{succ}(g)\) is undefined. If \(\text{succ}(g) = g'\) then we say that \(g\) is a predecessor of \(g'\); we say \(g'\) is a leaf if it has no predecessors. Let \(\text{leaves}(G)\) be the set of leaves of \(G\).
- A choice of one leaf as the current node (i.e. the program counter), called now.
- An injection \(\text{label} : \text{leaves}(G) \rightarrow \{a_1, \ldots, a_n\}\).
- An assignment to each node \(g\) of the forest \(G\) either a pair of types \((A, B)\) or a single type \(B\). We write \(g : A \rightarrow B\). We write \(g : () \rightarrow B\) respectively.
  - if \(g\) is a leaf then \(g : () \rightarrow B\), otherwise \(g : A \rightarrow B\);
  - if \(g : A \rightarrow B\) or \(g : () \rightarrow B\) and \(\text{succ}(g)\) is defined then \(\text{succ}(g) : B \rightarrow C\) for some \(C\);
  - there is one fixed type \(B_G\) such that for all roots \(g\) either \(g : () \rightarrow B_G\) or \(g : A \rightarrow B_G\) for some \(A\).
- An assignment to each node \(g\) of the forest \(G\) a program expression \(g\), subject to the following conditions. Here \(\Gamma_G = \{a_1 : \ell, \ldots, a_n : \ell\}\).
  - if \(g : () \rightarrow B\) then \(\Gamma_G \vdash \lbrack g \rbrack : B\).
  - if \(g : A \rightarrow B\) then \(\Gamma_G, x : A \vdash \lbrack g \rbrack : B\).

The informal idea is that in normal behaviour the machine proceeds by running the program expression at the leaf node now, passing the result to \(\text{succ}(\text{now})\). The machine may need to add new nodes to operate, and sometimes control may jump to a different leaf node.

Evaluation of pure computations Among the well-typed pure expressions in context \((\Gamma_G \vdash t : A)\) we distinguish values, defined by the following grammar:

\[
\begin{align*}
v ::= \text{fn} x & \rightarrow t | (v_1 \ldots v_k) | \text{inj}_i v | a
\end{align*}
\]

where \(a\) ranges over labels, and we define a type-preserving evaluation function \(\ll\), taking terms to values:

\[
\begin{align*}
t \ll (v_1 \ldots v_k) & \rightarrow t_1 \ll v_1 , \ldots , t_k \ll v_k \\
\#i t \ll v_i & \rightarrow (t_1 \ldots t_k) \ll (v_1 \ldots v_k) \\
(t \ll \text{inj}_i(v)) & \rightarrow u_{i/\ell} \ll w \\
(t \ll \text{inj}_i(v)) & \rightarrow t \ll v
\end{align*}
\]

Small steps of the machine A code heap changes over time. We describe the next step of a code heap by describing how it is modified at each step. We write in imperative pseudocode since this is a simple way to describe graph manipulations. We write \(G \rightarrow G'\) if the code heap \(G\) becomes \(G'\) according to the following transformations.

1. If \([\text{now}] = \text{return } t\) and \(t \ll v\) then we proceed depending on whether \(\text{now}\) is a root.
   (a) If \(\text{now}\) is a root then set \([\text{now}] := v\) and stop: the machine has finished.
   (b) If \(\text{now}\) is not a root, if \(\text{succ}(\text{now}) = g : A \rightarrow B\) and \(g = (\Gamma_G, x : A \vdash u : B)\), then we set \([\text{now}] := u[\ell/\ell]\) \(\text{succ}(\text{now}) := \text{succ}(g)\).

   If \(g\) has no remaining predecessors then we delete \(g\) from the graph.

2. If \(\text{now}() \rightarrow B\) and \([\text{now}] := (\text{let } x = t \text{ in } u)\) then we add a node \(g : A \rightarrow B\). We set \([g] := (\Gamma_G, x : A \vdash u : B)\) \(\text{now} := (\Gamma_G \vdash t : A)\) \(\text{succ}(g) := \text{succ}(\text{now})\) \(\text{succ}(g) := g\).

3. If \([\text{now}] := (\text{sub}(a.t, u))\) then we add a leaf node \(g\). We assume the binder \(a\) is different from the labels already in the machine, renaming it if necessary. We set \([\text{label}(g) := a]\) \([g] := u\) \([\text{now}] := t\) \(\text{succ}(g) := \text{succ}(\text{now})\).

4. If \([\text{now}] := (\text{var } a)\) and \(t \ll a\) then we proceed depending on whether \(a\) is in the image of the label function. If \(\text{label}(g) = a\), we set \([\text{now}] := g\). If \(a \not\in \text{im}(\text{label})\), we stop.

5. If \([\text{now}] := (v.w)\) and \(v \ll (\text{fn } x \rightarrow t)\) then we set \([\text{now}] := (t[\ell/\ell])\).

By construction, this stepping transformation preserves the well-formedness constraints of code heaps.

Some of the steps of the machine are illustrated as follows. We use a double edge to indicate the current node, now.
We say that a label $a$ is an entry point of a node $g$ if $a$ labels a leaf with a path to $g$ and there is a node $g'$ that is not a predecessor of $g$ but where $[g']$ mentions $a$. In what follows it will be convenient to transform a code heap into one where every node has at most one entry point.

If a branch node has two or more predecessors, it can be split in two by duplicating and then garbage collecting. This transformation is essentially the algebraicity law for sub,\[
\text{let } \text{val } x = \text{sub}(a, t, u) \text{ in } w \quad (a \not\in \text{fv}(w))
\]  
\[
\overset{\text{Proposition 3.}}{\equiv} \text{sub}(a, (\text{let } \text{val } x = t \text{ in } w), \text{let } \text{val } x = u \text{ in } w)
\]

In this way, any code heap can be transformed into one where each node has at most one entry point.

From a code heap to a typed term We assign a term
\[
b_1 \ldots b_m : \ell \vdash_s \text{Tm}(G) : B_G
\]
to every dependency-acyclic code heap $G$ of type $B_G$, where \(\{b_1 \ldots b_m\}\) are the dangling pointers.

We do this by induction on the height of $G$. Either $G$ is a forest (with many roots) or a tree (with one root).

- If $G$ is a forest, we duplicate branches and arrange the labels so that every node has exactly one entry point. Let $G_1 \ldots G_l$ be the trees comprising $G$, such that the order $G_1 < \cdots < G_l$ respects the dependency graph, and let $a_2 \ldots a_l$ be the entry points of $G_2 \ldots G_l$. We let
  \[
  \text{Tm}(G) \overset{\text{Definition}}{=} \text{sub}(a_1 \ldots \text{sub}(a_2, \text{Tm}(G_1), \text{Tm}(G_2)) \ldots G_l).
  \]
  
  There may be some choice about how many branches to split, but the algebraicity law says that this doesn’t matter.

- Suppose $G$ is a tree with root $g$. If $g$ is also a leaf then let
  \[
  \text{Tm}(G) \overset{\text{Definition}}{=} [g].
  \]
  Otherwise, let $G'$ be the forest of predecessors of $g$, so that $G'$ is the result of removing $g$ from $G$, and let
  \[
  \text{Tm}(G) \overset{\text{Definition}}{=} \text{let } \text{val } x = \text{Tm}(G') \text{ in } [g].
  \]

Proposition 4. If $G \rightsquigarrow G'$ then $\text{Tm}(G) \equiv \text{Tm}(G')$ is derivable from the equational theory in Section 2.

Proof notes: By case analysis on how a step can be made.

Garbage collection Given a code heap $G$, let $\text{gc}(G)$ be the code heap obtained from $G$ by removing all nodes that are not reachable from now in the dependency graph.

Proposition 5. 1. If $\text{gc}(G_1) = \text{gc}(G_2)$ and $G_1 \rightsquigarrow G'_1$ then there is $G'_2$ such that $G_2 \rightsquigarrow G'_2$ and $\text{gc}(G'_1) = \text{gc}(G'_2)$.

2. Let $G$ be a dependency-acyclic code heap. The code heap $\text{gc}(G)$ is also dependency-acyclic and $\text{Tm}(G) \equiv \text{Tm}(\text{gc}(G))$ is derivable.

Adequacy Theorem 6. If $\bar{a} : \ell \vdash_s t : \text{bool}$ and $[t] = [\text{return } v]$ then $t \rightsquigarrow^* G$ and $\text{gc}(G) = \text{return } v$ (for $v \in \{\text{tt}, \text{ff}\}$).

We follow the proof scheme for adequacy and algebraic effects in [35]: adequacy is a consequence of termination, together with Propositions 2 and 5. The machine can stop in various ways: either with $\text{var}(a)$ where $a$ is dangling, or with a root $\text{return}(v)$, possibly with some other reachable leaves.

We prove termination by defining computability predicates on typed expressions with free labels, also following [35]. There are two kinds of computability predicate, $R$ and $R_\ell$. The relation on pure expressions, $R(a_1, \ldots, a_n : \ell \vdash t : A)$, is defined by induction on the structure of types, as usual. For instance,
All our developments can be extended to cater for this, but we omit that allows a compound term to be assigned to an operation symbol.

If we run the code heap

Aside: comparison with a stack machine

We have taken measures to keep things semantically simple. If we can compose functions with different effects, say \( f : A \rightarrow E B \) and \( g : B \rightarrow E' C \), the language doesn't allow us to merely compose them, since \( x : A \vdash g(f(x)) : C \) is not well-formed. Instead, we must pick a signature \( E' \) that subsumes \( E \) and \( E' \) (for instance \( E' \supseteq E \cup E' \)) so that there are morphisms of signatures \( E \Rightarrow E' \Rightarrow E' \) and we can write

\[
    x : A \vdash E' g \cdot y = (f x) \cdot (g y) : C
\]

Equality

We thus have an equality judgement \( \Gamma \vdash t \equiv u : A \) on pure typed terms and an equality judgement \( \Gamma \vdash E t \equiv E u : A \) on effectful terms for each finite signature \( E \). Equality is generated as in §2.2, by reflexivity, symmetry, transitivity and substitutivity; the \( \beta - \eta \) laws for sums products and functions (2), the associativity and substitution laws for let (3); and additionally:

- the equality judgement \( \vdash E t \equiv \) includes equations of the form
  \[
  \phi(t_{1}, \ldots, t_{n}) \equiv \phi(op(t_{1}), \ldots, op(t_{n}) \phi)
  \]
  for signature morphisms \( \phi : E \rightarrow E' \) and each operation \( op \in E \), along with (return) \( \phi t = return t \)
  and (let val) \( \phi x = t u \cdot \phi = let \val x \equiv t \cdot \phi u \phi
  \)
- the equality judgement \( \vdash E t \equiv \) includes algebraic equation for each \( n \)-ary operation \( op \in E \), to propagate the effects:
  \[
  \equiv op(let \val x \equiv t_{1} \cdot \ldots \cdot \val x \equiv t_{n} \cdot \val u)
  \]

4.3 Denotational semantics

Each algebraic signature \( E \) determines a monad \( T_{E} \) on the category of sets. The set \( T_{E} X \) is the set of terms in the signature with variables in \( X \). The functions \( T_{E} X : X \rightarrow T_{E} X \) include the variables among the terms, and the functions \( \Rightarrow : T_{E} X \times (T_{E} Y) X \rightarrow T_{E} Y \) perform substitution of terms for variables. We use these monads to give a set-theoretic denotational semantics for our programming language.

### Interpretation of types

Product types are interpreted as the product of sets; sum types are interpreted as disjoint unions. The function type as functions into the free algebra \( T_{E} \), i.e. the space of Kleisli morphisms. In summary:

\[
\begin{align*}
    [A_{1} \times \ldots \times A_{n}]_{\text{set}} & \overset{\text{def}}{=} [A_{1}]_{\text{set}} \times \ldots \times [A_{n}]_{\text{set}} \\
    [A_{1} + \ldots + A_{n}]_{\text{set}} & \overset{\text{def}}{=} [A_{1}]_{\text{set}} \uplus \ldots \uplus [A_{n}]_{\text{set}} \\
    [A \rightarrow E B]_{\text{set}} & \overset{\text{def}}{=} [A]_{\text{set}} \Rightarrow T_{E}([A]_{\text{set}})
\end{align*}
\]

A context \( \Gamma \equiv \{ x_{1} : A_{1} \ldots ; x_{n} : A_{n} \} \) is interpreted as a set too: \( [\Gamma]_{\text{set}} = [A_{1}]_{\text{set}} \times \ldots \times [A_{n}]_{\text{set}} \).

### Interpretation of terms

A term in context \( \Gamma \vdash t : A \) is interpreted as a function \( [\Gamma]_{\text{set}} \rightarrow [A]_{\text{set}} \), and an effectful term in context \( \Gamma \vdash t : A \) is interpreted as a function \( [\Gamma]_{\text{set}} \rightarrow T_{E}([A]_{\text{set}}) \).
This interpretation is defined by induction on the structure of typing derivations. For instance:

\[ [\text{op}(t_1, \ldots, t_n)]_{\text{Set}}(\rho) \overset{\text{def}}{=} \text{op}([t_1]_{\text{Set}}(\rho), \ldots, [t_n]_{\text{Set}}(\rho)) \]

**Proposition 8.** The semantics in sets is sound: if \( \Gamma \vdash_E t \equiv u : A \) is derivable in the equality theory, then

\[ [t]_{\text{Set}} = [u]_{\text{Set}} : [\Gamma]_{\text{Set}} \rightarrow \text{TE}([A]_{\text{Set}}). \]

### 5. Representing virtual effects using actual code pointers

#### 5.1 An alternative denotational semantics for the language with virtual effects

A finite algebraic signature \( E = \{ \text{op}_1 : n_1 \ldots \text{op}_k : n_k \} \) can be thought of as a context-indexed-set,

\[ P_E \overset{\text{def}}{=} L^{n_1} + \cdots + L^{n_k}. \tag{4} \]

The terms of a signature also form a context-indexed-set, as a restriction of the monad \( \text{TE} \) in §4.3; let \( \text{TE}(n) \) be the terms in \( n \) fixed variables. Since \( P_E \) can be thought of as the terms involving exactly one operation, we can think of \( P_E \) as a sub-context-indexed-set of \( \text{TE} \). Indeed the full terms can be built from \( P_E \) by freely substituting:

**Theorem 9.** Let \( E \) be a finite algebraic signature. The context-indexed-set of terms over \( E \) is a free substitution algebra on the signature considered as a context-indexed-set \( (P_E) \).

**Remark:** This result entirely determines the monad \( S \) (§3.1); let \( \text{Sig} \) be the full subcategory of \( \text{Set}^{\text{Ctx}} \) whose objects are of the form \( P_E \); then \( S : \text{Set}^{\text{Ctx}} \rightarrow \text{Set}^{\text{Ctx}} \) is a left Kan extension of the terms-for-a-signature construction \( \text{Sig} \rightarrow \text{Set}^{\text{Ctx}} \) along the embedding \( \text{Sig} \rightarrow \text{Set}^{\text{Ctx}} \) (e.g. [42, Prop. 2], [41, §VILA]; more broadly [4, 32]).

Recall that for any object \( A \) on any category with sums, we have a monad \((-)-\setminus A\), sometimes called an ‘exceptions monad’, and each monad \( M \) extends to a monad \( M((-)-\setminus A) \), called the ‘exceptions monad transformer’. Roughly speaking, the exceptions monad transformer for the context-indexed-set \( P_E \) induces the monad \( \text{TE} \) on \( \text{Set} \) that was used for the denotational semantics of the language with virtual effects. To be precise, note that ‘evaluation at \( 0 \)’ functor \( (-)_0 : \text{Set}^{\text{Ctx}} \rightarrow \text{Set} \) has a left adjoint \( K : \text{Set} \rightarrow \text{Set}^{\text{Ctx}} \), which associates to each set \( X \) the context-indexed-set \( KX \) that is constantly \( X \).

**Proposition 10.** For any signature \( E \) we have an isomorphism of monads on \( \text{Set} \): for any set \( X \), \( (S(KX + P_E))_0 \cong \text{TE}X \).

### 5.2 Syntactic translation

The analysis above suggests a semantics for the programming language using context-indexed-sets instead of sets, and using monads of the form \( S((-)-\setminus P_E) \) instead of \( \text{TE}((-)-\setminus) \). This alternative semantics can be factored through the metalanguage with substitution. We now directly describe a translation from the programming language into the metalanguage.

**Interpretation of types** We translate a type of the programming language into a type of the metalanguage as follows. First, given a finite signature \( E = \{ \text{op}_1 : n_1 \ldots \text{op}_k : n_k \} \), we define a type \( \Sigma E \) in the metalanguage:

\[ \Sigma E \overset{\text{def}}{=} \ell^{n_1} + \cdots + \ell^{n_k} \] so that \([\Sigma E] = P_E\).

We adopt the following convention: when working with a type of the form \( A + \Sigma E \), rather than indexing the injections \( 1 \ldots (k + 1) \), we use an index 0 for the first summand \( (A) \) and use the names of the operations in \( E \) to index the second summand.

We now define a translation from types \( A \) of the programming language to types \( [A] \) of the metalanguage:

\[ [A_1 + \cdots + A_k] \overset{\text{def}}{=} [A_1] + \cdots + [A_k] \]

\[ [A \rightarrow_E B] \overset{\text{def}}{=} [A] \rightarrow_E ([B] + \Sigma E) \]

We translate a context \( \Gamma = (x_1 : A_1, \ldots, x_n : A_n) \) into a context \( [\Gamma] \overset{\text{def}}{=} (x_1 : [A_1], \ldots, x_n : [A_n]) \). We translate a pure judgement \( \Gamma \vdash t : A \) in the programming language to a pure judgement \( [\Gamma] \vdash [t] : [A] \) in the metalanguage, and an effectful judgement \( \Gamma \vdash_E t : A \) in the programming language to an effectful judgement \( [\Gamma] \vdash_E [t] : [A] + \Sigma E \) in the metalanguage. To do this, we first introduce a derived construction in the metalanguage: let

\[ \text{op}(t_1, \ldots, t_n) \overset{\text{def}}{=} \text{sub}(x_1, \ldots, \text{sub}(x_n, \text{return inj}_{\text{op}}(x_1 \ldots x_n), t_n), \ldots, t_1) \]

yielding the following derived rule:

\[ \Gamma \vdash_E t_1 : A + \Sigma E \quad \ldots \quad \Gamma \vdash_E t_n : A + \Sigma E \]

\[ \Gamma \vdash_E \text{op}(t_1, \ldots, t_n) : A + \Sigma E \]

An \( n \)-ary operation is thus implemented as a computation that returns \( n \) labels pointing to the remainder of the computation, as discussed in the introduction. This allows us to make the translation from a typed term \( t \) in the programming language to a typed term \([t] \) in the metalanguage, by induction on the structure of the syntax:

\[ \begin{align*}
\text{fn} x & \Rightarrow [t] \overset{\text{def}}{=} \text{fn} x \Rightarrow [t] \\
[t \ u] & \overset{\text{def}}{=} [t] \ [u] \\
\text{return } [t] & \overset{\text{def}}{=} \text{return } ([\text{inj}_0] [t]) \\
\text{let } \text{val } x = t \text{ in } u & \overset{\text{def}}{=} \text{case } [t] \text{ of } \text{inj}_0(x) \Rightarrow [u] \quad \ldots \quad \text{inj}_\text{op}(x) \Rightarrow \text{return } \text{inj}_{\text{op}}(x) \quad \ldots \\
\text{op}(t_1, \ldots, t_n) & \overset{\text{def}}{=} \text{op}([t_1], \ldots, [t_n]) \quad \text{(using 5)} \\
\text{let } \phi \text{ in } u & \overset{\text{def}}{=} \text{case } [t] \text{ of } \text{inj}_0(x) \Rightarrow \text{return } \text{inj}_0(x) \quad \ldots \quad \text{inj}_\text{op}(x) \Rightarrow \text{return } \text{inj}_{\phi(x)} \quad \ldots \\
\end{align*} \]

(The first four clauses are standard for the exceptions monad.)

**Proposition 11.** The translation is faithful: For terms \( t \) and \( u \) of the programming language, we have \( \Gamma \vdash_E t \equiv u : A \) if and only if \( \Gamma \vdash_E [t] \equiv [u] : [A] + \Sigma E \).

### 6. Handlers of virtual effects

In the final two sections of this paper we investigate some additional features that can be added to our metalanguage. In this section we consider the possibility of setting handlers for effects. Informally, a handler will capture an effect tree and deal with it. This section is inspired by recent developments on programming with handlers of effects [2, 5, 25, 31], although that line of work can be traced back to the earlier work on delimited control (notably [18]) and monadic reflection [9]. The important thing to note is that our denotational semantics (§3.1) already supports these constructs, and our abstract machine (§3.2) is easily adapted to accommodate them.

In the metalanguage, first-order types (types without \( \rightarrow \)) can be thought of as signatures, since they are all isomorphic to types of the form \( \Sigma E \) for a signature \( E \). When \( B \) is a first order type and \( A \) is any type, we can define \( B \bullet A \) (the depth 1 \( B \)-terms in \( A \)) by...
induction on $B$:

$$\ell \cdot A \equiv A \quad (B_1 \cdots \cdots B_n) \cdot A \equiv (B_1 \cdots \cdots B_n) \cdot A \quad \cdots \quad (B_1 \cdots \cdots B_n) \cdot A \equiv (B_1 \cdots \cdots B_n) \cdot A \quad$$

We consider the following term formation rule, which we add to the metamodel.

$$\Gamma \vdash t : A + B \quad \Gamma, x : B \cdot (\text{unit} \rightarrow_s A) \vdash u : A \quad (B \text{ is first order})$$

Informally, the program ($\text{handle } t \text{ with } x \cdot u : A$) will first run $t$. If $t$ returns normally, i.e. some $\text{inl}(v)$, then handle $t$ with $x \cdot u$ will return $v$ in other words:

$$\text{handle} \ (\text{return} \ \text{inl}(v)) \ \text{with} \ x \cdot u \equiv \text{return} v$$

If, on the other hand, $t$ returns some exceptional value $\text{inr}(v)$ containing labels for the continuation of the effect tree, then handle $t$ with $x \cdot u$ will ‘handle’ these labels by recursing through the effect tree; for instance

$$\text{handle sub}(a, \text{return}(\text{inr} \ a), t) \ \text{with} \ x \cdot u \equiv u [\ell] \Rightarrow \text{handle} \ t \ \text{with} \ x \cdot u [\ell]$$

We extend the translation from machines to programs ($\cdot \text{unit} \rightarrow_s A$) fixes the answer type as $\text{bool}$.

We did not include this monoidal structure as a first-class type constructor in the metamodel because there does not appear to be a good term syntax for it. However, it is lurking: for a first-order type $A$ and any type $B$, $[A \cdot B] \equiv [\text{unit}] \cdot [B]$.

Indeed, recall that every signature $E$ induces a context-indexed-set $\text{P}_{E}$. To give a context-indexed-function $\text{P}_{E} : Q \rightarrow Q$ is to give a context-indexed-function $Q' \rightarrow Q$ for each operation ($\cdot n$) in $E$, i.e., an algebra for the signature. This is the structure that appears in the second premise of the term formation rule for handlers, matching up with the motto ‘handlers are algebras’ [38].

**Characterization of free algebraic theories**

The context-indexed-set $\text{P}_{E} : Q \rightarrow Q$ can be thought of as depth-1 $E$-terms in $Q$. Thus we can build all $E$-terms by iterating the construction. In other words, $\text{SP}$ is the free $\cdot$-monoid on $P$ (e.g. [12, 19, 26, 28])

$$\text{SP} = \mu M. L + P \cdot M. \quad (6)$$

This gives us a structural recursion principle for eliminating $\text{SP}$. We use this to define the denotational semantics of handlers.

**Denotational semantics for handlers**

For simplicity we only give an interpretation to terms

$$− \vdash \text{handle } t \text{ with } x \cdot u : A$$

where the ambient context is empty. (To accommodate a non-empty context, one uses the fact that the constructions involved are strong.)

First, we use the initiality (6) of $\text{S} [A + B]$ to define a context-indexed-function $\text{S} [A + B] \rightarrow \text{S} [A]$. To this end we must define context-indexed-functions

$$L \rightarrow \text{S} [A] \quad [A + B] \cdot \text{S} [A] \rightarrow \text{S} [A].$$

The left-hand context-indexed-function is the var operation. The right-hand context-indexed-function can be equivalently given by two context-indexed-functions, $[A] \cdot \text{S} [A] \rightarrow \text{S} [A]$, which arises
from the initiality property (6) of $S[A]$, and $[B] \bullet S[A] \rightarrow S[A]$, which is the denotational semantics of the term $u$.

We now compose this context-indexed-function $S[A + B] \rightarrow S[S[A]]$ with the denotational semantics of handle $t$ with $x, u$.

**Adequacy**

**Theorem 12.** If $\bar{a} : \ell \vdash_\ell t : \text{bool}$ and $[\bar{a}] = [\text{return } v]$ then $\vdash G$ and $gc(G) = \text{return } v$.

7. Further actual effects

The metalanguage in Section 2 has two actual effects, sub and var. We demonstrated that virtual effects can be encoded into this metalanguage. We now explain how to incorporate further actual effects.

7.1 Example: a bit of memory

We begin by considering how the theory of accessing a single bit of memory can be accommodated into the metalanguage. We begin by considering how the theory of accessing a single bit of memory can be accommodated into the metalanguage. We extend the transition behaviour with the following clauses:

$\Gamma \vdash t : \ell \quad \Gamma \vdash u : A \quad \Gamma, x : \ell \vdash t : A$

$\Gamma \vdash \text{pop}(t,u) : A$

$\Gamma \vdash \text{push}(t,u) : A$

with corresponding 'generic effects':

$\text{Pop} \equiv \text{fn } x : \ell \Rightarrow \text{pop}(x, \text{return } \langle \rangle ) : \ell \rightarrow s \text{ unit}$

$\text{Push} \equiv \text{fn } x : \ell \Rightarrow \text{push}(x, \text{return } \langle \rangle ) : \ell \rightarrow s \text{ unit}$

The first equation can be written $\text{Push}(a) : \text{Pop}(a) \equiv \text{return } a$.

**Abstract machine**

We can also accommodate this extra effect in our code heap machine by defining a configuration to be a pair $(G, stk)$ of a code heap and a list $stk$ of labels, considered as a stack. We extend the transition behaviour with the following clauses:

$6'$ If $[\text{now}] = (\text{push}(t,u))$ and $t \nmid a$ then we push $a$ onto the stack $stk$ (i.e. $stk ::= a :: stk$). We add a new leaf $g$ and a new label $a$ with $\text{label}(g) = a$, and set $[g] := u$ and $\text{succ}(g) := \text{succ}(\text{now})$ now $:= g$.

$7'$ If $[\text{now}] = (\text{pop}(x,t))$ then if the stack $stk$ is empty, we stop. If the stack is not empty, we pop the top element $a$. We add a new leaf $g$ and a new label $a$ with $\text{label}(g) = a$, and set $[g] := t[a] \Rightarrow \text{succ}(g) := \text{succ}(\text{now})$ now $:= g$.

**Encoding recursion**

We can define the following derived term, when $\ell$ has a free variable $a$ of type $\ell$ and $b$ is fresh.

$$\text{mk-loop}(a,t) \equiv \text{sub}(b, \text{push}(b, \text{var } b), \text{pop}(b, \text{sub}(a.t, \text{push}(b, \text{var } b))))$$

This term has the property that

$$\text{mk-loop}(a,t) = \text{sub}(a.t, \text{mk-loop}(a,t))$$

Its generic effect

$$\text{PC} \equiv \text{fn } x : \ell \Rightarrow \text{mk-loop}(a.t, \text{return } a) : \text{unit } \rightarrow s \ell$$

can be thought of as a command which returns the current program counter.

If we combine the theory of a stack of pointers with the theory of store then we have generic effects

$$\text{Wr}_{tt} \equiv \text{fn } x : \ell \Rightarrow \text{wr}_{t}(\text{return } \langle \rangle ) \quad (i \in \{tt, ff\})$$

can run the following program

$$\text{let val } x = \text{PC}(i) \text{ in } \text{Wr}_{tt}(i) \cdot \text{Wr}_{ff}(i) \cdot \text{var } x$$

and in the machine it runs forever, continually flipping the bit in the store.

**Connection with the lambda calculus**

We conclude by explaining that that the algebraic theory for a stack of pointers is the equational theory of $\beta$-equality in the untyped $\lambda$-calculus. To see this, let $\text{lam}(a.t) \equiv \text{pop}(a.t)$ and let $\text{app}(t,u) \equiv \text{sub}(b, \text{push}(b, t, u))$ (b fresh). This algebraic theory is thus essentially the one in [11, §B, 13, Ex. 3]; its models are the semi-closed algebraic theories [23] (see also [22]). The derived term $\text{mk-loop}$ is essentially Curry’s $Y$ combinator.

We could also consider the $\eta$-law, $\text{pop}(a.\text{push}(a,x)) \equiv x$. This says that the stack is never empty.

8. Summary

We have presented a foundational analysis of the principles of programming with algebraic effects.
We have shown that concepts such as labels and jumps are not merely implementation details for virtual algebraic effects. Rather, they arise already from a mathematical result about algebraic signatures; an algebraic signature is a free model of the theory of substitution.

We have demonstrated this by designing a typed metalanguage based around the theory of substitution and jumps (§2). We solidified the computational intuitions about the relationship between substitution, labels and jumps by giving an abstract machine (§3). We gave a sound interpretation of an effectful programming language into this metalanguage (§5).

In the final sections we sketched how handlers for effects can be understood as an extension of this metalanguage (§6) We also considered some extensions of the theory of substitution, most notably β-equality for the untyped lambda calculus, which, we argue, describes the computational effects associated with a stack of code pointers (§7).

We have brought together several lines of work but some links remain to be made. We are now investigating whether other aspects of the denotational model have an elegant syntactic counterpart. For example, there is an isomorphism \( S_0 \cong L \) in the model that is not definable in the metalanguage; it suggests a new language construct that takes an effectful computation of type 0, which must eventually jump (\( \text{var } a \)), and traps that jump, returning the pure label (\( a \)). Our work also suggests new styles of programming with jumps, which we are developing.

Acknowledgements It has been helpful to discuss this line of work with many people, including Daniel Ahman, Marco Ferreira Devesas Campos, Hugo Herbelin, Ohad Kammar, Paul Levy, Sam Lindley, Gordon Plotkin and Noam Zeilberger. Our research was partly supported by ERC Projects ECSYM and QCLS.

References