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.

$$(x \oplus y) \oplus z \qquad \textit{i.e.} \qquad \begin{tikzcd} & & \oplus \arrow[dl, "a"'] \arrow[dr] & \\ & \oplus \arrow[dl] \arrow[dr] & & z \\ x & & y & \end{tikzcd} \tag{1}$$

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 nondeterministic 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, sub and var. It involves a basic type ℓ, which might be thought of as a type of variables or indices, and the following term formation rules:

$$\frac{\Gamma, a : \ell \vdash t \quad \Gamma \vdash u}{\Gamma \vdash \mathsf{sub}(a.t, u)} \qquad \frac{\Gamma \vdash t : \ell}{\Gamma \vdash \mathsf{var}\, t}$$

Informally, we think of $\mathsf{sub}(a.t, u)$ as 'substitute u for each occurrence of $\mathsf{var}\, a$ in t'. We work up to α-renaming a in t and equations such as $\mathsf{sub}(a.\mathsf{var}\, a, u) \equiv u$. We have the following theorem (The-

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.

http://dx.doi.org/10.1145/2603088.2603163

orem 9): *To give an algebraic signature is to give a free algebra for the algebraic theory of substitution.*

Plotkin and Power [36] proposed to consider algebraic theories 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 ℓ is as a type of labels or code pointers, so that $\mathsf{sub}(a.t, u)$ can be read as 'create a new code pointer to u, bind it to a and continue as t', and $\mathsf{var}\, a$ 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 $\oplus$ 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 $\mathsf{sub}(a.\mathsf{return}\,(a, z), \mathsf{return}\,(x, y))$. 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 $(\mathsf{return\,tt} \oplus \mathsf{return\,ff}) : \mathsf{bool}$ which returns either tt or ff depending on the outcome of the virtual effect $\oplus$. This is translated into our metalanguage as

$$\mathsf{sub}(a.\mathsf{sub}(b.\mathsf{return\,inr}(a, b), \mathsf{return\,inl}(\mathsf{ff})), \mathsf{return\,inl}(\mathsf{tt})) : \mathsf{bool} + (\ell \times \ell)$$

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

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 β-equality of the untyped λ-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, ι and ℓ, an operation $\mathsf{sub} : (\ell \to \iota) \times \iota \to \iota$ and an operation $\mathsf{var} : \ell \to \iota$. The theory has four equations:

$$x : \iota \vdash \mathsf{sub}(\lambda a.\mathsf{var}(a), x) \equiv x$$

$$x : \iota, y : \iota \vdash \mathsf{sub}(\lambda a.x, y) \equiv x$$

$$a : \ell, x : \ell \to \iota \vdash \mathsf{sub}(\lambda b.x(b), \mathsf{var}(a)) \equiv x(a)$$

$$x : \ell{\times}\ell \to \iota, y : \ell \to \iota, z : \iota \vdash \mathsf{sub}(\lambda a.\mathsf{sub}(\lambda b.x(a, b), y(a)), z) \equiv \mathsf{sub}(\lambda b.\mathsf{sub}(\lambda a.x(a, b), z), \mathsf{sub}(\lambda a.y(a), z))$$

The idea is that ι is a type of terms, ℓ is a type of variables (or 'labels'), and var includes the variables in the terms and $\mathsf{sub}(\lambda a.t, u)$ substitutes u for a in t.

One can also read $\mathsf{sub}(\lambda a.t, u)$ as 'bind u to a in t', and $\mathsf{var}\, a$ 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

$$A, B ::= \ell \mid A \to_{\mathsf{s}} B \mid A_1 {*} \cdots {*} A_n \mid A_1 + \cdots + A_n \quad (n \in \mathbb{N}).$$

We have a special type ℓ 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 $(\to_{\mathsf{s}})$ 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 $(\vdash_{\mathsf{s}})$. The terms in context are defined as follows. Firstly we have rules for sums and products of pure computations:

$$\frac{-}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma \vdash t : A_1 {*} \ldots {*} A_n}{\Gamma \vdash \#i\, t : A_i}$$

$$\frac{\Gamma \vdash t : A_i}{\Gamma \vdash \mathsf{inj}_i t : A_1 + \cdots + A_n} \qquad \frac{\ldots\ \Gamma \vdash t_i : A_i\ \ldots}{\Gamma \vdash \langle t_1\ \ldots\ t_n \rangle : A_1 {*} \ldots {*} A_n}$$

$$\frac{\Gamma \vdash t : A_1 + \cdots + A_n \quad \ldots\ \Gamma, x_i : A_i \vdash u_i : B\ \ldots}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\ \mathsf{inj}_1(x_1) \Rightarrow u_1 \ldots \mathsf{inj}_n(x_n) \Rightarrow u_n : B}$$

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

$$\frac{\Gamma \vdash_{\mathsf{s}} t : A \quad \Gamma, x : A \vdash_{\mathsf{s}} u : B}{\Gamma \vdash_{\mathsf{s}} \mathsf{let\,val}\, x = t\, \mathsf{in}\, u : B} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash_{\mathsf{s}} \mathsf{return}\, t : A}$$

Thirdly the specific term formation rules for the metalanguage:

$$\frac{\Gamma \vdash t : \ell}{\Gamma \vdash_{\mathsf{s}} \mathsf{var}_A(t) : A} \qquad \frac{\Gamma, a : \ell \vdash_{\mathsf{s}} t : A \quad \Gamma \vdash_{\mathsf{s}} u : A}{\Gamma \vdash_{\mathsf{s}} \mathsf{sub}_A(a.t, u) : A}$$

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

$$\frac{\Gamma, x : A \vdash_{\mathsf{s}} t : B}{\Gamma \vdash \mathsf{fn}\, x \Rightarrow t : A \to_{\mathsf{s}} B} \qquad \frac{\Gamma \vdash t : A \to_{\mathsf{s}} B \quad \Gamma \vdash u : A}{\Gamma \vdash_{\mathsf{s}} t\, u : B}$$

We will use simple syntactic sugar. We write bool for the type $\mathsf{unit}{+}\mathsf{unit}$, and tt and ff for $\mathsf{inj}_1\langle\rangle$ and $\mathsf{inj}_2\langle\rangle$ respectively. We write $(\mathsf{if}\, t\, \mathsf{then}\, u_1\, \mathsf{else}\, u_2)$ for $(\mathsf{case}\, t\, \mathsf{of}\ \mathsf{inj}_1(\_) \Rightarrow u_1 \mid \mathsf{inj}_2(\_) \Rightarrow u_2)$, and we write $(t\, ;\, u)$ instead of $(\mathsf{let\,val}\, \_ = t\, \mathsf{in}\, u)$. 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 $(\mathsf{let\,val}\, f{=}(t\, u)\, \mathsf{in}\, (f v))$, and so on.

Equational theory We have an equality judgement on pure typed terms, $\Gamma \vdash t \equiv u : A$, and an equality judgement on effectful terms, $\Gamma \vdash_{\mathsf{s}} t \equiv u : A$. The judgements are generated as follows:

- equality is reflexive, symmetric, transitive and substitutive;
- on pure terms we include the standard β/η laws:

$$\#i\, \langle t_1 \ldots t_n \rangle \equiv t_i \qquad t \equiv \langle \#1\, t \ldots \#n\, t \rangle$$

$$\mathsf{case}\, (\mathsf{inj}_i\, t)\, \mathsf{of} \ldots \mathsf{inj}_i(x_i) \Rightarrow u_i \ldots \equiv u_i[{}^{t}/_{x_i}] \qquad (2)$$

$$\mathsf{case}\, t\, \mathsf{of} \ldots \mathsf{inj}_i(x_i) \Rightarrow u[{}^{\mathsf{inj}_i(x_i)}/_{x}] \ldots \equiv u[{}^{t}/_{x}]$$

- on effectful terms we have the following standard laws [30, 34]:

$$\begin{aligned}
&\mathsf{let\,val}\,x = \mathsf{return}\,t\,\mathsf{in}\,u \equiv u[^t/_x] \quad t \equiv \mathsf{let\,val}\,x = t\,\mathsf{in\,return}\,x \\
&\mathsf{let\,val}\,x = t\,\mathsf{in}\,(\mathsf{let\,val}\,y = u\,\mathsf{in}\,w) \\
&\qquad\qquad \equiv \mathsf{let\,val}\,y = (\mathsf{let\,val}\,x = t\,\mathsf{in}\,u)\,\mathsf{in}\,w\ (y \notin \mathrm{fv}(t)) \\
&(\mathsf{fn}\,x \Rightarrow t)\,u \equiv t[^u/_x] \qquad t \equiv \mathsf{fn}\,x \Rightarrow (t\,x)\ \ (x \notin \mathrm{fv}(t))
\end{aligned} \tag{3}$$

- the judgement $\Gamma \vdash_{\mathsf{s}} t \equiv u : A$ includes all the equations for substitution (§2.1); together with two 'algebraicity' equations [40], which propagate the effects:

$$\begin{aligned}
&\mathsf{let\,val}\,x = \mathsf{sub}(a.t, u)\,\mathsf{in}\,w \qquad\qquad\qquad (a \notin \mathrm{fv}(w)) \\
&\qquad\quad \equiv\ \mathsf{sub}(a.(\mathsf{let\,val}\,x = t\,\mathsf{in}\,w), \mathsf{let\,val}\,x = u\,\mathsf{in}\,w) \\
&\mathsf{let\,val}\,x = \mathsf{var}(t)\,\mathsf{in}\,w\ \equiv\ \mathsf{var}(t)
\end{aligned}$$

Informal semantics We may think of ℓ as a type of code pointers or labels, so that $\mathsf{sub}(a.t, u)$ creates a new label a, and continues as t; if and when $\mathsf{var}(a)$ is called, the program jumps back to the point where a was created, and continues as u instead. The type of labels ℓ 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 $\mathsf{sub}(a.t, u)$ as installing a new statically-scoped exception a with handler u, and $\mathsf{var}(a)$ as raising the exception.

The reader should *not* think of $\mathsf{sub}(a.t, u)$ as textual substitution of u for a in t: that is a meta-operation that is typically nonsense in this context, and indeed it is inconsistent with the algebraicity equations. (The construct $\mathsf{sub}(a.t, u)$ 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 ℓ behaves like "$\neg\mathsf{unit}$". 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

$$\mathsf{fn}\,\_ \Rightarrow \mathsf{sub}(a.\mathsf{return\,inj}_1(a), \mathsf{return\,inj}_2\langle\rangle) : \mathsf{unit} \to_{\mathsf{s}} \ell + \mathsf{unit}$$

(NB we do not mean the *textual* substitution of a by $(\mathsf{return\,inj}_2\langle\rangle)$ in $(\mathsf{return\,inj}_1(a))$, which does not make sense.) This generic effect is like the law of the excluded middle $\neg\mathsf{unit} \vee \mathsf{unit}$. Recall a computational intuition for the excluded middle (e.g. [6],[20, §31.4]): it first introduces the left hand disjunct ($\neg\mathsf{unit}$); 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 ℓ (e.g. [41, §VI-C]). To resolve this we consider sets indexed by contexts of labels $(a_1{:}\ell \cdots a_n{:}\ell)$. 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 $P\colon \mathbf{Ctx} \to \mathbf{Set}$, where $\mathbf{Ctx}$ is the category of natural numbers (considered as sets) and functions between them.*

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

The objects n of $\mathbf{Ctx}$ 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 ℓ, to a model of intuitionistic higher-order logic. More precisely, the category $\mathbf{Set}^{\mathbf{Ctx}}$ of context-indexed-sets is the free cocompletion of $\mathbf{Ctx}^{\mathrm{op}}$, 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 $\mathrm{L}(n) = n$. The Yoneda lemma provides a natural family of bijections $P(n) \cong \mathbf{Set}^{\mathbf{Ctx}}(\mathrm{L}^n, P)$.
- The category is cartesian closed: there is a context-indexed-set Q^P of context-indexed-functions that satisfies
$$Q^P(n) \cong \mathbf{Set}^{\mathbf{Ctx}}(P \times \mathrm{L}^n, Q).$$
- In particular the function space $(Q^{(\mathrm{L}^n)})$ has a natural family of bijections $(Q^{(\mathrm{L}^n)})(m) \cong Q(m+n)$. 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 ℓ as L. In more detail, a *substitution algebra* is a context-indexed-set P equipped with a family of functions $\mathsf{sub}_n : P(n+1) \times P(n) \to P(n)$ and $\mathsf{var}_n : \mathrm{L}(n) \to P(n)$ 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* $\mathrm{S}P$, i.e. a substitution algebra $\mathrm{S}P$ together with a context-indexed-function $\eta : P \to \mathrm{S}P$ such that for any substitution algebra Q and any context-indexed-function $f : P \to Q$ there is a unique homomorphism $f^\sharp : \mathrm{S}P \to Q$ such that $f = f^\sharp \cdot \eta$. 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 ([12],[42, Cor. 1]): for any context-indexed-sets P and Q there is a context-indexed-function $\gg\!= : \mathrm{S}P \times (\mathrm{S}Q)^P \to \mathrm{S}Q$, satisfying the monad laws.

(We will use two other 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:

$$[\![\ell]\!] \stackrel{\text{def}}{=} \mathrm{L} \qquad [\![A_1 * \cdots * A_n]\!] \stackrel{\text{def}}{=} [\![A_1]\!] \times \cdots \times [\![A_n]\!]$$

$$[\![A \to_{\mathsf{s}} B]\!] \stackrel{\text{def}}{=} (\mathrm{S}[\![B]\!])^{[\![A]\!]} \qquad [\![A_1 + \cdots + A_n]\!] \stackrel{\text{def}}{=} [\![A_1]\!] \uplus \cdots \uplus [\![A_n]\!]$$

Pure terms ($\vdash$) and effectful terms ($\vdash_{\mathsf{s}}$) are interpreted as context-indexed-functions:

$$[\![\Gamma \vdash t : A]\!] : [\![\Gamma]\!] \to [\![A]\!] \qquad [\![\Gamma \vdash_{\mathsf{s}} u : A]\!] : [\![\Gamma]\!] \to \mathrm{S}[\![A]\!].$$

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

$$[\![\mathsf{let\,val}\; x = t \;\mathsf{in}\; u]\!](\rho) \stackrel{\text{def}}{=} [\![t]\!](\rho) \ggg \lambda x.\, [\![u]\!](\rho, x)$$

(in the internal language of $\mathbf{Set}^{\mathbf{Ctx}}$). We define $[\![\mathsf{var}(t)]\!]$ and $[\![\mathsf{sub}(a.t, u)]\!]$ using the substitution algebra structure of the monad S.

Proposition 2. *The semantics of the metalanguage is sound: if $\Gamma \vdash_{\mathsf{s}} t \equiv u : A$ is derivable in the equality theory, then*

$$[\![t]\!] = [\![u]\!] : [\![\Gamma]\!] \to \mathrm{S}[\![A]\!].$$

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. [1].)

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

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

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 $succ(now)$. 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:

$$v ::= \mathsf{fn}\, x \;\Rightarrow\; t \mid \langle v_1 \dots v_k \rangle \mid \mathsf{inj}_i\, v \mid a$$

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

$$\frac{t \Downarrow \langle v_1 \dots v_k \rangle}{\#i\, t \Downarrow v_i} \qquad \frac{t_1 \Downarrow v_1 \;\dots\; t_k \Downarrow v_k}{\langle t_1 \dots t_k \rangle \Downarrow \langle v_1 \dots v_k \rangle}$$

$$\frac{t \Downarrow \mathsf{inj}_i(v) \quad u_i[{}^{v}/_{x_i}] \Downarrow w}{(\mathsf{case}\, t\, \mathsf{of} \dots \mathsf{inj}_i(x_i) \Rightarrow u_i \dots) \Downarrow w} \qquad \frac{t \Downarrow v}{\mathsf{inj}_i(t) \Downarrow \mathsf{inj}_i(v)}$$

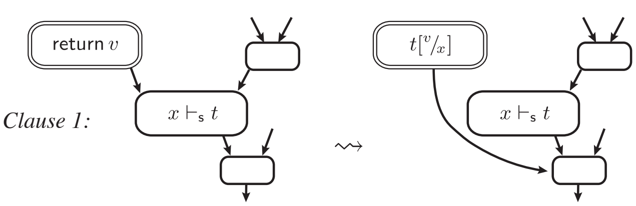
Small steps of the machine A code heap changes over time. We describe the next step of a code heap by describing how it is modified at each step. We write in an imperative pseudocode since this is a simple way to describe graph manipulations. We write $G \rightsquigarrow G'$ if the code heap G becomes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

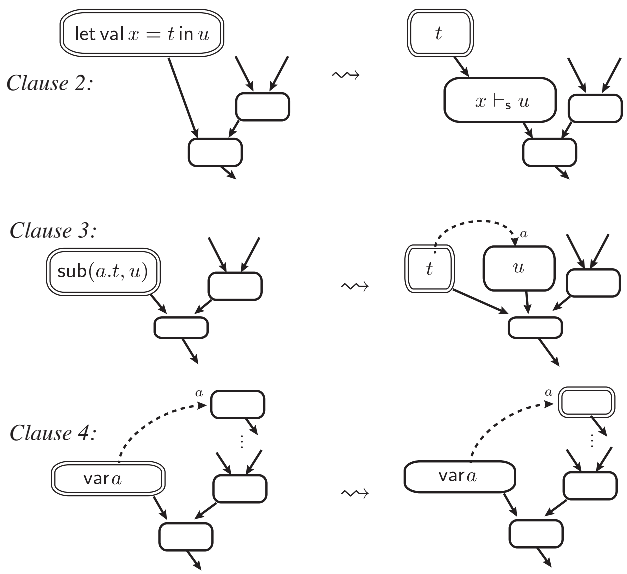
1. If $[now] = \mathsf{return}\, t$ and $t \Downarrow v$ then we proceed depending on whether now is a root.
   (a) If now is a root then set $[now] := v$ and stop: the machine has finished.
   (b) If now is not a root, if $succ(now) = g : A \to B$ and $[g] = (\Gamma_G, x : A \vdash_{\mathsf{s}} u : B)$, then we set
   $$[now] := u[{}^{v}/_{x}] \qquad succ(now) := succ(g).$$
   If g has no remaining predecessors then we delete g from the graph.
2. If $now : () \to B$ and $[now] = (\mathsf{let\,val}\, x : A = t\, \mathsf{in}\, u)$ then we add a node $g : A \to B$. We set
   $$\begin{array}{ll} [g] := (\Gamma_G, x : A \vdash_{\mathsf{s}} u : B) & [now] := (\Gamma_G \vdash_{\mathsf{s}} t : A) \\ succ(g) := succ(now) & succ(now) := g. \end{array}$$
3. If $[now] = (\mathsf{sub}(a.t, u))$ 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
   $$\begin{array}{ll} label(g) := a & [g] := u \\ [now] := t & succ(g) := succ(now). \end{array}$$
4. If $[now] = (\mathsf{var}\, t)$ and $t \Downarrow a$ then we proceed depending on whether a is in the image of the $label$ function. If $label(g) = a$, we set $now := g$. If $a \notin \mathrm{im}(label)$, we stop.
5. If $[now] = (v\, w)$ and $v \Downarrow (\mathsf{fn}\, x \;\Rightarrow\; t)$ then we set $[now] := (t[{}^{w}/_{x}])$.

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.

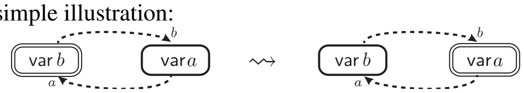
Clause 1:





3.3 Analysis of the abstract machine

Dependency graphs and garbage collection The *dependency graph* of a code heap is given by adding to the graph an edge $g \to g'$ whenever $[g]$ mentions $label(g')$. A cycle in the dependency graph indicates that the code heap might run forever, as in this simple illustration:



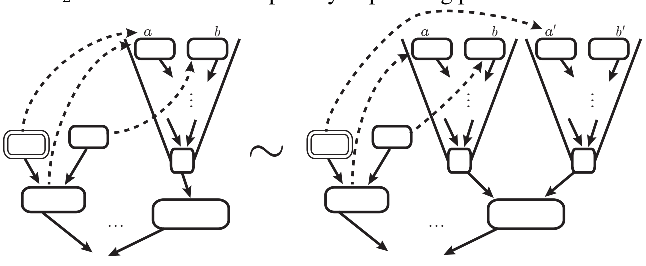
We say a code heap is *dependency-acyclic* when its dependency graph is acyclic. The stepping transformation preserves dependency acyclicity. In particular code heaps that arise from running programs written in the metalanguage are dependency-acyclic.

If a node is not reachable in the dependency graph, it plays no role in the behaviour of the machine. We can 'garbage collect' those nodes: given a code heap G, let $\mathrm{gc}(G)$ be the code heap obtained from G by removing all nodes that are not reachable from now in the dependency graph.

Proposition 3. *If* $\mathrm{gc}(G_1) = \mathrm{gc}(G_2)$ *and* $G_1 \rightsquigarrow G_1'$ *then there is* G_2' *such that* $G_2 \rightsquigarrow G_2'$ *and* $\mathrm{gc}(G_1') = \mathrm{gc}(G_2')$.

Garbage collection is justified by the second equation in our presentation of the theory of substitution (§2.1). The equational theory of substitution also suggests other sound optimizations that we could perform in the machine. For instance, the third equation describes a way of removing extraneous links.

Duplicating branches and entry points Any part of a code heap can be duplicated without changing the behaviour of the machine. If there are labels in the part that is duplicated, they are also duplicated. Labels pointing into that part of the code heap can then be arranged to point to either copy. We write $G_1 \sim G_2$ if G_1 and G_2 can be made isomorphic by duplicating parts. For instance:



Duplication preserves the steps of the machine.

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,

$$\mathsf{let\,val}\, x = \mathsf{sub}(a.t, u)\, \mathsf{in}\, w \qquad\qquad (a \notin \mathrm{fv}(w))$$
$$\equiv\ \mathsf{sub}(a.(\mathsf{let\,val}\, x = t\, \mathsf{in}\, w), \mathsf{let\,val}\, x = u\, \mathsf{in}\, w)$$

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

$$b_1 \dots b_m : \ell \vdash_{\mathsf{s}} \mathrm{Tm}(G) : B_G$$

to every dependency-acyclic code heap G of type B_G, where $\{b_1 \dots b_m\}$ are the dangling pointers

$$\{b_1 \dots b_m\} = \{a \mid \neg\exists g.\, label(g) = a\}.$$

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 $G_1 \dots G_l$ be the trees comprising G, such that the order $G_1 < \dots < G_l$ respects the dependency graph, and let $a_2 \dots a_l$ be the entry points of $G_2 \dots G_l$. We let

  $$\mathrm{Tm}(G) \stackrel{\mathrm{def}}{=} \mathsf{sub}(a_l \dots \mathsf{sub}(a_2.\mathrm{Tm}(G_1), \mathrm{Tm}(G_2)) \dots G_l).$$

  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 $\mathrm{Tm}(G) \stackrel{\mathrm{def}}{=} [g]$. Otherwise, let G' be the forest of predecessors of g, so that G' is the result of removing g from G, and let $\mathrm{Tm}(G) \stackrel{\mathrm{def}}{=} \mathsf{let\,val}\, x = \mathrm{Tm}(G')\, \mathsf{in}\, [g]$.

Proposition 4. *If* $G \rightsquigarrow G'$ *then* $\mathrm{Tm}(G) \equiv \mathrm{Tm}(G')$ *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 $\mathrm{gc}(G)$ 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* $\mathrm{gc}(G_1) = \mathrm{gc}(G_2)$ *and* $G_1 \rightsquigarrow G_1'$ *then there is* G_2' *such that* $G_2 \rightsquigarrow G_2'$ *and* $\mathrm{gc}(G_1') = \mathrm{gc}(G_2')$.
2. Let G *be a dependency-acyclic code heap. The code heap* $\mathrm{gc}(G)$ *is also dependency-acyclic and* $\mathrm{Tm}(G) \equiv \mathrm{Tm}(\mathrm{gc}(G))$ *is derivable.*

Adequacy

Theorem 6. *If* $\vec{a} : \ell \vdash_{\mathsf{s}} t : \mathsf{bool}$ *and* $[\![t]\!] = [\![\mathsf{return}\, v]\!]$ *then* $t \rightsquigarrow^* G$ *and* $\mathrm{gc}(G) = \mathsf{return}\, v$ *(for* $v \in \{\mathsf{tt}, \mathsf{ff}\}$*).*

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 $\mathsf{var}(a)$ where a is dangling, or with a root $\mathsf{return}(v)$, 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_{s}. The relation on pure expressions, $\mathrm{R}(a_1, \dots, a_n : \ell \vdash t : A)$, is defined by induction on the structure of types, as usual. For instance,

$\mathrm{R}(\vec{a} \vdash t : A \to_{\mathsf{s}} B)$ if $t \Downarrow (\mathsf{fn}\, x \Rightarrow u)$ and for all v with $\mathrm{R}(\vec{a} \vdash v : A)$ we have $\mathrm{R}_{\mathsf{s}}(\vec{a} \vdash_{\mathsf{s}} u[{}^{v}/_{x}] : B)$. The predicate on effectful expressions, $\mathrm{R}_{\mathsf{s}}(a_1, \ldots, a_n : \ell \vdash_{\mathsf{s}} t : A)$, is the least predicate closed under the following rules:

$$(t \rightsquigarrow^* G \not\rightsquigarrow \text{ and } [now] = \mathsf{var}(a)) \implies \mathrm{R}_{\mathsf{s}}(t)$$
$$(t \rightsquigarrow^* G \not\rightsquigarrow \text{ and } \mathrm{gc}(G) = \mathsf{return}(v) \text{ and } \mathrm{R}(v)) \implies \mathrm{R}_{\mathsf{s}}(t)$$
$$(t \rightsquigarrow^* G \not\rightsquigarrow \text{ and } [now] = \mathsf{return}(v) \text{ and } \mathrm{R}(v) \text{ and}$$
$$\forall g \in \mathit{leaves}(\mathrm{gc}(G)), g \neq \mathit{now} \Rightarrow \mathrm{R}_{\mathsf{s}}(\mathrm{Tm}(G^{\dagger g}))) \implies \mathrm{R}_{\mathsf{s}}(t)$$

where $G^{\dagger g}$ is the code heap G but with $now := g$. We use these predicates to show, via the usual 'fundamental lemma', that every term terminates when run in the machine.

Aside: comparison with a stack machine If we run the code heap machine over a term without sub or var, then every node will have at most one predecessor. We thus have a linked-list, i.e. a stack: let pushes onto the stack and return pops.

It is possible to modify the code heap machine to maintain this linked-list structure, by modifying the cases for sub and var so that they copy/replace the entire continuation, instead of manipulating pointers. This is the standard CK machine semantics for a language with unit continuations (e.g. [7, 8, 29]).

4. Algebraic signatures and virtual effects

We now introduce a programming language, inspired by Filinski's multimonadic metalanguage [10] and related developments (e.g. [24, 46]). The typing judgements are annotated by finite algebraic signatures which describe the effects that may occur in running a program.

The main goal of this section is merely to set the scene for the next section (§5), where we show how to translate the language with virtual effects into the metalanguage with substitution.

4.1 Algebraic signatures

Definition 7. *A* finite algebraic signature *comprises a finite set of operation symbols each equipped with an arity, which is a natural number (possibly zero). We write* $\mathrm{op} : n$ *if the operation symbol* op *has arity* n*.*

A morphism of algebraic signatures $E \to E'$ *assigns to each operation symbol in* E *an operation symbol in* E' *with the same arity.*

For a simple example, consider the signature with a single operation $\oplus : 2$, which we discussed in the introduction.

(There is another, more general notion of signature morphism that allows a compound term to be assigned to an operation symbol. All our developments can be extended to cater for this, but we omit the details.)

4.2 A programming language with effects and annotations

We consider a language with the following types:

$$A, B \ ::= A_1 * \ldots * A_n \mid A_1 + \cdots + A_n \mid A \to_E B$$

Here E ranges over finite algebraic signatures. Informally, the function types $A \to_E B$ are annotated by the effects E that may occur when the function is called.

There is a typing judgement $\vdash$ for pure computations, and there is a judgement $\vdash_E$ for each finite algebraic signature E: it is a judgement of typed computations involving effects in that signature. The judgements are defined by the standard rules (§2.2) for pure sums and products and sequencing of computations for each judgement $\vdash_E$, together with the following specific rules for the operations in the signature:

$$\frac{\Gamma \vdash_E t_1 : A \quad \ldots \quad \Gamma \vdash_E t_n : A}{\Gamma \vdash_E \mathrm{op}(t_1, \ldots, t_n) : A}\ (\mathrm{op} : n) \in E$$

$$\frac{\Gamma \vdash_E t : A}{\Gamma \vdash_{E'} (t)_\phi : A}\ \phi : E \to E' \text{ is a morphism of signatures}$$

and the following rules for functions:

$$\frac{\Gamma, x : A \vdash_E t : B}{\Gamma \vdash \mathsf{fn}\, x \Rightarrow t : A \to_E B} \qquad \frac{\Gamma \vdash t : A \to_E B \quad \Gamma \vdash u : A}{\Gamma \vdash_E t\, u : B}$$

We have taken measures to keep things semantically simple. If we have composable functions with different effects, say $f : A \to_E B$ and $g : B \to_{E'} C$, 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 $E'' = E \cup E'$) so that there are morphisms of signatures $E \xrightarrow{\phi} E'' \xleftarrow{\phi'} E'$ and we can write

$$x : A \vdash_{E''} \mathsf{let\,val}\, y = (f\,x)_\phi\ \mathsf{in}\ (g\,y)_{\phi'} : C$$

Equality We thus have an equality judgement $\Gamma \vdash t \equiv u : A$ on pure typed terms and an equality judgement $\Gamma \vdash_E t \equiv u : A$ 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 $(\vdash_{E'} \equiv)$ includes equations of the form
$$(\mathrm{op}(t_1, \ldots, t_n))_\phi \equiv (\phi\,\mathrm{op})((t_1)_\phi, \ldots, (t_n)_\phi)$$
for signature morphisms $\phi : E \to E'$ and each operation $\mathrm{op} \in E$, along with $(\mathsf{return}\, t)_\phi \equiv \mathsf{return}\, t$ and
$$(\mathsf{let\,val}\, x = t\ \mathsf{in}\ u)_\phi \equiv \mathsf{let\,val}\, x = (t)_\phi\ \mathsf{in}\ (u)_\phi$$
- the equality judgement $(\vdash_E \equiv)$ includes algebraicity equations for each n-ary operation $\mathrm{op} \in E$, to propagate the effects:
$$\begin{aligned}&\mathsf{let\,val}\, x = \mathrm{op}(t_1, \ldots, t_n)\ \mathsf{in}\ u \\ &\qquad \equiv \mathrm{op}(\mathsf{let\,val}\, x = t_1\ \mathsf{in}\ u, \ldots, \mathsf{let\,val}\, x = t_n\ \mathsf{in}\ u)\end{aligned}$$

4.3 Denotational semantics

Each algebraic signature E determines a monad T_E on the category of sets. The set $\mathrm{T}_E X$ is the set of terms in the signature with variables in X. The functions $\eta_X : X \to \mathrm{T}_E X$ include the variables among the terms, and the functions $\gg\!= : \mathrm{T}_E X \times (\mathrm{T}_E Y)^X \to \mathrm{T}_E Y$ 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:

$$[\![A_1 * \ldots * A_n]\!]_{\mathbf{Set}} \stackrel{\text{def}}{=} [\![A_1]\!]_{\mathbf{Set}} \times \cdots \times [\![A_n]\!]_{\mathbf{Set}}$$
$$[\![A_1 + \cdots + A_n]\!]_{\mathbf{Set}} \stackrel{\text{def}}{=} [\![A_1]\!]_{\mathbf{Set}} \uplus \cdots \uplus [\![A_n]\!]_{\mathbf{Set}}$$
$$[\![A \to_E B]\!]_{\mathbf{Set}} \stackrel{\text{def}}{=} ([\![A]\!]_{\mathbf{Set}} \to \mathrm{T}_E([\![B]\!]_{\mathbf{Set}}))$$

A context $\Gamma = (x_1 : A_1 \ldots x_n : A_n)$ is interpreted as a set too: $[\![\Gamma]\!]_{\mathbf{Set}} = [\![A_1]\!]_{\mathbf{Set}} \times \cdots \times [\![A_n]\!]_{\mathbf{Set}}$.

Interpretation of terms A term in context $\Gamma \vdash t : A$ is interpreted as a function $[\![\Gamma]\!]_{\mathbf{Set}} \to [\![A]\!]_{\mathbf{Set}}$, and an effectful term in context $\Gamma \vdash_E t : A$ is interpreted as a function $[\![\Gamma]\!]_{\mathbf{Set}} \to \mathrm{T}_E([\![A]\!]_{\mathbf{Set}})$.

This interpretation is defined by induction on the structure of typing derivations. For instance:

$$[\![\mathrm{op}(t_1,\ldots,t_n)]\!]_{\mathbf{Set}}(\rho) \stackrel{\text{def}}{=} \mathrm{op}([\![t_1]\!]_{\mathbf{Set}}(\rho),\ldots,[\![t_n]\!]_{\mathbf{Set}}(\rho))$$

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

$$[\![t]\!]_{\mathbf{Set}} = [\![u]\!]_{\mathbf{Set}} : [\![\Gamma]\!]_{\mathbf{Set}} \to \mathrm{T}_E([\![A]\!]_{\mathbf{Set}}).$$

5. Representing virtual effects using actual code pointers

5.1 An alternative denotational semantics for the language with virtual effects

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

$$\mathrm{P}_E \stackrel{\text{def}}{=} \mathrm{L}^{n_1} + \cdots + \mathrm{L}^{n_k}. \tag{4}$$

The terms of a signature also form a context-indexed-set, as a restriction of the monad T_E in §4.3: let $\mathrm{T}_E(n)$ 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 T_E. 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 $\mathbf{Sig}$ be the full subcategory of $\mathbf{Set}^{\mathbf{Ctx}}$ whose objects are of the form P_E; then $\mathrm{S} : \mathbf{Set}^{\mathbf{Ctx}} \to \mathbf{Set}^{\mathbf{Ctx}}$ is a left Kan extension of the terms-for-a-signature construction $\mathbf{Sig} \to \mathbf{Set}^{\mathbf{Ctx}}$ along the embedding $\mathbf{Sig} \to \mathbf{Set}^{\mathbf{Ctx}}$ (e.g. [42, Prop. 2], [41, §VII.A]; more broadly [4, 32]).

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

Proposition 10. *For any signature E we have an isomorphism of monads on $\mathbf{Set}$: for any set X, $(\mathrm{S}(\mathrm{K}X + \mathrm{P}_E))_0 \cong \mathrm{T}_E X$.*

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 $\mathrm{S}((-)+\mathrm{P}_E)$ instead of $\mathrm{T}_E(-)$. 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 to a type of the metalanguage as follows. First, given a finite signature $E = \{\mathrm{op}_1 : n_1 \ldots \mathrm{op}_k : n_k\}$, we define a type ΣE in the metalanguage:

$$\Sigma E \stackrel{\text{def}}{=} \ell^{n_1} + \cdots + \ell^{n_k} \qquad \text{so that } [\![\Sigma E]\!] = \mathrm{P}_E.$$

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 $\lfloor A \rfloor$ of the metalanguage.

$$\begin{aligned}
\lfloor A_1 * \ldots * A_k \rfloor &\stackrel{\text{def}}{=} \lfloor A_1 \rfloor * \ldots * \lfloor A_k \rfloor \\
\lfloor A_1 + \cdots + A_k \rfloor &\stackrel{\text{def}}{=} \lfloor A_1 \rfloor + \cdots + \lfloor A_k \rfloor \\
\lfloor A \to_E B \rfloor &\stackrel{\text{def}}{=} \lfloor A \rfloor \to_{\mathsf{s}} (\lfloor B \rfloor + \Sigma E)
\end{aligned}$$

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

$$\mathrm{op}(t_1 \cdots t_n) \stackrel{\text{def}}{=} \mathsf{sub}(x_1.\cdots \mathsf{sub}(x_n.\mathsf{return}\ \mathsf{inj}_{\mathrm{op}}(x_1 \cdots x_n), t_n) \cdots, t_1)$$

yielding the following derived rule:

$$\frac{\Gamma \vdash_{\mathsf{s}} t_1 : A + \Sigma E \quad \ldots \quad \Gamma \vdash_{\mathsf{s}} t_n : A + \Sigma E}{\Gamma \vdash_{\mathsf{s}} \mathrm{op}(t_1, \ldots, t_n) : A + \Sigma E} \tag{5}$$

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 $\lfloor t \rfloor$ in the metalanguage, by induction on the structure of the syntax:

$$\begin{aligned}
\lfloor \mathsf{fn}\, x \Rightarrow t \rfloor &\stackrel{\text{def}}{=} \mathsf{fn}\, x \Rightarrow \lfloor t \rfloor \\
\lfloor t\, u \rfloor &\stackrel{\text{def}}{=} \lfloor t \rfloor \lfloor u \rfloor \\
\lfloor \mathsf{return}\, t \rfloor &\stackrel{\text{def}}{=} \mathsf{return}(\mathsf{inj}_0 \lfloor t \rfloor) \\
\lfloor \mathsf{let\,val}\, x = t\, \mathsf{in}\, u \rfloor &\stackrel{\text{def}}{=} \mathsf{case}\, \lfloor t \rfloor\, \mathsf{of}\, \mathsf{inj}_0(x) \Rightarrow \lfloor u \rfloor \mid \\
&\qquad \ldots \mid \mathsf{inj}_{\mathrm{op}}(x) \Rightarrow \mathsf{return}\, \mathsf{inj}_{\mathrm{op}}(x) \mid \ldots \\
\lfloor \mathrm{op}(t_1, \ldots, t_n) \rfloor &\stackrel{\text{def}}{=} \mathrm{op}(\lfloor t_1 \rfloor, \ldots, \lfloor t_n \rfloor) \quad \text{(using (5))} \\
\lfloor (t)_\phi \rfloor &\stackrel{\text{def}}{=} \mathsf{case}\, \lfloor t \rfloor\, \mathsf{of}\, \mathsf{inj}_0(x) \Rightarrow \mathsf{return}\, \mathsf{inj}_0(x) \mid \\
&\qquad \ldots \mid \mathsf{inj}_{\mathrm{op}}(x) \Rightarrow \mathsf{return}\, \mathsf{inj}_{(\phi\, \mathrm{op})}(x) \mid \ldots
\end{aligned}$$

(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 $\lfloor \Gamma \rfloor \vdash_{\mathsf{s}} \lfloor t \rfloor \equiv \lfloor u \rfloor : \lfloor A \rfloor + \Sigma E$.*

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 $\to_{\mathsf{s}}$) can be thought of as signatures, since they are all isomorphic to types of the form ΣE 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:

$$\ell \bullet A \stackrel{\text{def}}{=} A \qquad (B_1 * \cdots * B_n) \bullet A \stackrel{\text{def}}{=} (B_1 \bullet A) * \cdots * (B_n \bullet A)$$

$$(B_1 + \cdots + B_n) \bullet A \stackrel{\text{def}}{=} (B_1 \bullet A) + \cdots + (B_n \bullet A)$$

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

$$\frac{\Gamma \vdash_{\mathsf{s}} t : A + B \quad \Gamma, x : B \bullet (\mathsf{unit} \to_{\mathsf{s}} A) \vdash_{\mathsf{s}} u : A}{\Gamma \vdash_{\mathsf{s}} \mathsf{handle}\, t\, \mathsf{with}\, x.\, u : A} \ (B \text{ is first-order})$$

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

$$\mathsf{handle}\,(\mathsf{return}\,(\mathsf{inl}\, v))\,\mathsf{with}\, x.\, u \equiv \mathsf{return}\, v$$

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

$$\mathsf{handle}\,\mathsf{sub}(a.\mathsf{return}(\mathsf{inr}\, a), t)\,\mathsf{with}\, x.\, u \equiv u[^{\mathsf{fn}\langle\rangle \Rightarrow \mathsf{handle}\, t\, \mathsf{with}\, x.\, u}/_x]$$

$$\mathsf{handle}\,(\mathsf{return}(\mathsf{inr}\, a))\,\mathsf{with}\, x.\, u \equiv u[^{(\mathsf{fn}\langle\rangle \Rightarrow \mathsf{var}\, a)}/_x]$$

If we think of ℓ as a type of unit continuations, then, roughly, $B \bullet (\mathsf{unit} \to_{\mathsf{s}} A)$ fixes the answer type as A.

In the simple case where $B = \mathsf{unit}$ then handling reduces to case analysis just as in a hand-coded implementation of exception handling.

For an example, consider a program using virtual effects in the signature with a binary operation, $t : A{+}\ell{*}\ell$. We will use handlers to interpret the virtual operation as test-and-flip on a single bit of memory (aka fetch-and-complement), transforming the program t with virtual effects into a state-passing program of the type $\mathsf{bool} \to_{\mathsf{s}} \mathsf{bool}{*}A$. We first define an auxiliary term

$$t' \stackrel{\text{def}}{=} \mathsf{case}\, t\, \mathsf{of}\, \mathsf{inl}(x) \Rightarrow \mathsf{return}\, \mathsf{inl}(\mathsf{fn}\, s \Rightarrow \langle s, x\rangle)$$
$$\mid \mathsf{inr}(x) \Rightarrow \mathsf{return}\, \mathsf{inr}(x) \ : (\mathsf{bool} \to_{\mathsf{s}} \mathsf{bool}{*}A) + \ell{*}\ell$$

and now we consider the following term:

$$\mathsf{handle}\, t'\, \mathsf{with}\, k.\, \mathsf{return}(\mathsf{fn}\, s \Rightarrow$$
$$\mathsf{if}\, s\, \mathsf{then}\, (\#1\, k)\langle\rangle\, \mathsf{ff}\, \mathsf{else}\, (\#2\, k)\langle\rangle\, \mathsf{tt}) \quad : \mathsf{bool} \to_{\mathsf{s}} \mathsf{bool}{*}A$$

which runs t by state-passing: c.f. [2, §6.3], [25, §2.1]. (The example demonstrates that some more elaborate syntax would be useful [3]).

6.1 Accommodating handlers in the abstract machine

We accommodate handlers in the abstract machine by adding a special kind of branch node denoted $g : A \xrightarrow{\text{handle}} B$. We require that $\Gamma_G, x : A \bullet (\mathsf{unit}{\to_{\mathsf{s}}}B) \vdash_{\mathsf{s}} [g] : B$ and that if $succ(g') : A \xrightarrow{\text{handle}} B$ then either $g' : C \to (B + A)$ or $g' : () \to (B + A)$ or $g' : C \xrightarrow{\text{handle}} (B + A)$.

We extend the translation from machines to programs (§3.3) as follows: if G is a tree with root $g : B \xrightarrow{\text{handle}} C$ and G' is the forest of predecessors of g, then let

$$\mathrm{Tm}(G) \stackrel{\text{def}}{=} \mathsf{handle}\, \mathrm{Tm}(G')\, \mathsf{with}\, x.\, [g].$$

Before we extend the transition behaviour of the machine we first make the following definition. If a is a label for a machine and $g : A \xrightarrow{\text{handle}} B$ is a branch node, then we define a program $\Gamma_G \vdash (a{-}g) : B$ as follows. First, let $G{\downarrow_{\bullet g}}$ be the tree of nodes with a path to g (including g), and let $(a-g) \stackrel{\text{def}}{=} \mathrm{Tm}(\mathsf{var}(a), G{\downarrow_{\bullet g}})$. For example, if g is not reachable from a then $(a - g) \equiv \mathsf{var} a$.

We now define the machine by adding the following case:

6) If $now : () \to B$ and $[now] = \mathsf{handle}\, t\, \mathsf{with}\, x.\, u$ then we add a new node $g : A \xrightarrow{\text{handle}} B$ and set

$$[now] := t \qquad [g] := u$$
$$succ(g) := succ(now) \qquad succ(now) := g$$

and by extending case 1 with the following clauses:

1c) If $succ(now) : A \xrightarrow{\text{handle}} B$ then we proceed as follows.

i) If $v = \mathsf{inl}(w)$ then we set $[now] := \mathsf{return}\, w$ and $succ(now) := succ(succ(now))$.

ii) If $v = \mathsf{inr}(w)$ then let $g = succ(now)$, and suppose $[g] = \Gamma_G, x : A \bullet (\mathsf{unit}{\to_{\mathsf{s}}}B) \vdash_{\mathsf{s}} t : B$. Notice that w has type A and, if we substitute $(\mathsf{fn}\,\langle\rangle \Rightarrow (a - g))$ for each label a in w, we obtain a term $w[^{\mathsf{fn}\langle\rangle \Rightarrow (a-g)}/_a]$ of type $(A \bullet (\mathsf{unit}{\to_{\mathsf{s}}}B))$. We set

$$[now] := t[^{w[^{\mathsf{fn}\langle\rangle \Rightarrow (a-g)}/_a]}/_x]$$
$$succ(now) := succ(g).$$

6.2 Denotational semantics of handlers

Substitution monoidal product We have given a denotational semantics for the metalanguage by interpreting types as context-indexed-sets. The category of context-indexed-sets has a monoidal structure $\bullet$ (discussed in [15, 26, 39, 43] and elsewhere). One way to explain it is as follows: to give a context-indexed-set is to give a functor $\mathbf{Set} \to \mathbf{Set}$ that preserves filtered colimits, and the monoidal structure corresponds to the composition of functors. The unit for the monoidal structure is L. The monoidal structure $\bullet$ is such that $(- \bullet P)$ preserves colimits and finite products.

We did not include this monoidal structure as a first-class type constructor in the metalanguage 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, $[\![A \bullet B]\!] \cong [\![A]\!] \bullet [\![B]\!]$.

Indeed, recall that every signature E induces a context-indexed-set P_E (4). To give a context-indexed-function $\mathrm{P}_E \bullet Q \to Q$ is to give a context-indexed-function $Q^n \to Q$ for each operation $(\mathrm{op} : n)$ 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 $\mathrm{P}_E \bullet Q$ 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, $\mathrm{S}P$ is the free $\bullet$-monoid on P (e.g. [12, 19, 26, 28]):

$$\mathrm{S}P = \mu M.\, \mathrm{L} + P \bullet M. \tag{6}$$

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

Denotational semantics for handlers For simplicity we only give an interpretation to terms

$$- \vdash_{\mathsf{s}} \mathsf{handle}\, t\, \mathsf{with}\, x.\, u : A$$

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 $\mathrm{S}[\![A + B]\!]$ to define a context-indexed-function $\mathrm{S}[\![A + B]\!] \to \mathrm{S}[\![A]\!]$. To this end we must define context-indexed-functions

$$\mathrm{L} \to \mathrm{S}[\![A]\!] \qquad [\![A + B]\!] \bullet \mathrm{S}[\![A]\!] \to \mathrm{S}[\![A]\!].$$

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, $[\![A]\!] \bullet \mathrm{S}[\![A]\!] \to \mathrm{S}[\![A]\!]$, which arises from the initiality property (6) of $\mathrm{S}[\![A]\!]$, and $[\![B]\!] \bullet \mathrm{S}[\![A]\!] \to \mathrm{S}[\![A]\!]$, which is the denotational semantics of the term u.

We now compose this context-indexed-function $\mathrm{S}[\![A+B]\!] \to \mathrm{S}[\![A]\!]$ with the denotational semantics of t, to obtain the denotational semantics of $\mathsf{handle}\, t \,\mathsf{with}\, x.\, u$.

Adequacy

Theorem 12. *If* $\vec{a} : \ell \vdash_{\mathsf{s}} t : \mathsf{bool}$ *and* $[\![t]\!] = [\![\mathsf{return}\, v]\!]$ *then* $t \rightsquigarrow^* G$ *and* $\mathrm{gc}(G) = \mathsf{return}\, v$.

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 extend the theory of substitution (§2.1) with three function symbols:

$$\mathsf{rd} : \iota \times \iota \to \iota \qquad \mathsf{wr}_{\mathsf{tt}} : \iota \to \iota \qquad \mathsf{wr}_{\mathsf{ff}} : \iota \to \iota$$

and add the equations for a bit of memory (e.g. [36], [41, IIIA]) together with the following equations which explain how substitution propagates over reading/writing:

$$\mathsf{sub}(a.\mathsf{rd}(x(a), y(a)), z) \equiv \mathsf{rd}(\mathsf{sub}(a.x(a), z), \mathsf{sub}(a.y(a), z))$$
$$\mathsf{sub}(a.\mathsf{wr}_i(x(a)), z) \equiv \mathsf{wr}_i(\mathsf{sub}(a.x(a), z))$$

We note that these kinds of equation are discussed in a different context in [16] and they are implicit in [13, 14].

Metalanguage We accommodate reading/writing in the metalanguage using the following term formation rules:

$$\frac{\Gamma \vdash_{\mathsf{s}} t : A \quad \Gamma \vdash_{\mathsf{s}} u : A}{\Gamma \vdash_{\mathsf{s}} \mathsf{rd}(t, u) : A} \qquad \frac{\Gamma \vdash_{\mathsf{s}} t : A}{\Gamma \vdash_{\mathsf{s}} \mathsf{wr}_b(t) : A}\, b = \mathsf{tt}, \mathsf{ff}$$

Abstract machine We can accommodate the extra effect in the code heap machine by redefining a configuration to be a pair (G, s) of a code heap G and a bit s. We then add clauses to the transition behaviour such as

- If $[now] = (\mathsf{rd}(t, u))$ and $s = \mathsf{tt}$ then we add a new leaf g and a new label a with $label(g) := a$, and set
$$[g] := t \quad succ(g) := succ(now) \quad now := g.$$

7.2 Example: Stacks of code pointers and untyped λ-calculus

Our second example concerns the effect of having a stack of labels alongside the code heap. This is an old idea (see e.g. [7, §2]) but we can shed new light on it: it amounts to the algebraic theory of the untyped λ-calculus. We accommodate it by extending the theory of substitution (§2.1) with two function symbols:

$$\mathsf{push} : \ell \times \iota \to \iota \qquad \mathsf{pop} : (\ell \to \iota) \to \iota$$

subject to the following equations.

$$a : \ell, x : \ell \to \iota \vdash \mathsf{push}(a, \mathsf{pop}(b.x(b))) \equiv x(a)$$
$$b : \ell, x : \ell \to \iota, z : \iota \vdash \mathsf{sub}(a.\mathsf{push}(b, x(a)), z) \equiv \mathsf{push}(b, \mathsf{sub}(a.x(a), z))$$
$$x : \ell \to \iota, z : \iota \vdash \mathsf{sub}(a.\mathsf{push}(a, x(a)), z) \equiv \mathsf{sub}(a.\mathsf{push}(a, \mathsf{sub}(a.x(a), z)), z)$$
$$x : \ell \times \ell \to \iota, z : \iota \vdash \mathsf{sub}(a.\mathsf{pop}(b.x(a, b)), z) \equiv \mathsf{pop}(b.\mathsf{sub}(a.x(a, b), z))$$

Metalanguage We accommodate the stack in the metalanguage by adding the following term formation rules:

$$\frac{\Gamma \vdash t : \ell \qquad \Gamma \vdash_{\mathsf{s}} u : A}{\Gamma \vdash_{\mathsf{s}} \mathsf{push}(t, u) : A} \qquad \frac{\Gamma, x : \ell \vdash_{\mathsf{s}} t : A}{\Gamma \vdash_{\mathsf{s}} \mathsf{pop}(x.t) : A}$$

with corresponding 'generic effects':

$$\mathsf{Push} \stackrel{\text{def}}{=} \mathsf{fn}\, x : \ell \Rightarrow \mathsf{push}(x, \mathsf{return}\, \langle\rangle) : \ell \to_{\mathsf{s}} \mathsf{unit}$$
$$\mathsf{Pop} \stackrel{\text{def}}{=} \mathsf{fn}\, \_ : \mathsf{unit} \Rightarrow \mathsf{pop}(x.\mathsf{return}\, x) : \mathsf{unit} \to_{\mathsf{s}} \ell$$

The first equation can be written $\mathsf{Push}(a) \,;\, \mathsf{Pop}\langle\rangle \equiv \mathsf{return}\, a$.

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 $[now] = (\mathsf{push}(t, u))$ and $t \Downarrow a$ 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 $label(g) = a$, and set
$$[g] := u \quad succ(g) := succ(now) \quad now := g.$$

7′) If $[now] = (\mathsf{pop}(x.t))$ 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 $label(g) = a$, and set
$$[g] := t[^a/_x] \quad succ(g) := succ(now) \quad now := g.$$

Encoding recursion We can define the following derived term, when t has a free variable a of type ℓ and b is fresh.

$$\mathsf{mk\text{-}loop}(a.t) \stackrel{\text{def}}{=} \mathsf{sub}(b.\mathsf{push}(b, \mathsf{var}\, b), \mathsf{pop}(b.\mathsf{sub}(a.t, \mathsf{push}(b, \mathsf{var}\, b))))$$

This term has the property that

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

Its generic effect

$$\mathsf{PC} \stackrel{\text{def}}{=} \mathsf{fn}\, \_ : \mathsf{unit} \Rightarrow \mathsf{mk\text{-}loop}(a.\mathsf{return}\, a) : \mathsf{unit} \to_{\mathsf{s}} \ell$$

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

$$\mathsf{Wr}_i \stackrel{\text{def}}{=} \mathsf{fn}\, \_ : \mathsf{unit} \Rightarrow \mathsf{wr}_i(\mathsf{return}\, \langle\rangle) \qquad (i \in \{\mathsf{tt}, \mathsf{ff}\})$$

can run the following program

$$\mathsf{let\, val}\, x = \mathsf{PC}\langle\rangle\, \mathsf{in}\, \mathsf{Wr}_{\mathsf{tt}}\langle\rangle \,;\, \mathsf{Wr}_{\mathsf{ff}}\langle\rangle \,;\, \mathsf{var}\, x$$

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 β-equality in the untyped λ-calculus. To see this, let $\mathsf{lam}(a.t) \stackrel{\text{def}}{=} \mathsf{pop}(a.t)$ and let $\mathsf{app}(t, u) \stackrel{\text{def}}{=} \mathsf{sub}(b.\mathsf{push}(b, t), u)$ (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 $\mathsf{mk\text{-}loop}$ is essentially Curry's Y combinator.

We could also consider the η-law, $\mathsf{pop}(a.\mathsf{push}(a, x)) \equiv x$. 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 immediately 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 $\mathrm{S0} \cong \mathrm{L}$ 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 ($\mathsf{var}\ a$), 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 Danel 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

[1] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proc. POPL 1989*, pages 293–302, 1989.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. arXiv:1203.1539, 2012.

[3] N. Benton and A. Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, 2001.

[4] C. Berger, P.-A. Melliès, and M. Weber. Monads with arities and their associated theories. *J. Pure Appl. Algebra*, 216, 2012.

[5] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proc. ICFP 2013*, 2013.

[6] P. de Groote. A simple calculus of exception handling. In *TLCA 1995*, pages 201–215, 1995.

[7] B. F. Duba, R. Harper, and D. B. MacQueen. Typing first-class continuations in ML. In *Proc. POPL 1991*, pages 163–173, 1991.

[8] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts*, pages 193–217. North Holland, 1986.

[9] A. Filinski. Representing monads. In *Proc. POPL'94*, 1994.

[10] A. Filinski. On the relations between monadic semantics. *Theor. Comput. Sci.*, 375(1–3):41–75, 2007.

[11] M. P. Fiore. Mathematical models of computational and combinatorial structures. In *FOSSACS 2005*, 2005.

[12] M. P. Fiore. Second-order and dependently-sorted abstract syntax. In *LICS 2008*, 2008.

[13] M. P. Fiore and C.-K. Hur. Second-order equational logic. In *CSL 2010*, pages 320–335, 2010.

[14] M. P. Fiore and O. Mahmoud. Second-order algebraic theories. In *Proc. MFCS 2010*, pages 368–380, 2010.

[15] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. LICS 1999*, pages 193–202, 1999.

[16] M. J. Gabbay and A. Mathijssen. Capture-avoiding substitution as a nominal algebra. *Formal Asp. Comput.*, 2008.

[17] J. Gibbons. Unifying theories of programming with monads. In *Proc. UTP 2012*, pages 23–67, 2012.

[18] C. A. Gunter, R. Didier, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proc. FPCA 1995*, pages 12–23. ACM, 1995.

[19] M. Hamana. Free Σ-monoids: A higher-order syntax with metavariables. In *APLAS 2004*, pages 348–363, 2004.

[20] R. Harper. *Practical Foundations for Programming Languages*. CUP, 2012.

[21] H. Herbelin. An intuitionistic logic that proves Markov's principle. In *Proc. LICS 2010*, pages 50–56, 2010.

[22] A. Hirschowitz and M. Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010.

[23] J. M. E. Hyland. Classical lambda calculus in modern dress. *Math. Struct. Comput. Sci.*, 2014. To appear.

[24] O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *POPL 2012*, pages 349–360, 2012.

[25] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proc. ICFP 2013*, 2013.

[26] G. M. Kelly and A. J. Power. Adjunctions whose counits are coequalisers. *J. Pure Appl. Algebra*, 89:163–179, 1993.

[27] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Haskell 2013*, pages 59–70, 2013.

[28] S. Lack. On the monadicity of finitary monads. *J. Pure Appl. Algebra*, 140:65–73, 1999.

[29] P. B. Levy. *Call-by-push-value. A functional/imperative synthesis*. Springer, 2004.

[30] P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Inform. and Comput.*, 2003.

[31] C. McBride. The Frank manual. `tinyurl.com/frank-manual`, 2012.

[32] P.-A. Melliès. Segal condition meets computational effects. In *Proc. LICS 2010*, pages 150–159, 2010.

[33] R. E. Møgelberg and S. Staton. Linearly-used state in models of call-by-value. In *CALCO 2011*, 2011.

[34] E. Moggi. Notions of computation and monads. *Inform. and Comput.*, 1991.

[35] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In *Proc. FOSSACS 2001*, pages 1–24, 2001.

[36] G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FOSSACS'02*, pages 342–356, 2002.

[37] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[38] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *Proc. ESOP'09*, pages 80–94, 2009.

[39] J. Power. Abstract syntax: Substitution and binders. In *MFPS 2007*, 2007.

[40] M. Pretnar. *The Logic and Handling of Algebraic Effects*. PhD thesis, School of Informatics, University of Edinburgh, 2010.

[41] S. Staton. Instances of computational effects. In *LICS 2013*, 2013.

[42] S. Staton. An algebraic presentation of predicate logic. In *FOSSACS 2013*, pages 401–417, 2013.

[43] M. Tanaka and J. Power. Pseudo-distributive laws and axiomatics for variable binding. *Higher-Order and Symbolic Computation*, 2006.

[44] H. Thielecke. *Categorical structure of continuation passing style*. PhD thesis, Univ. Edinburgh, 1997.

[45] J. Voigtländer. Asymptotic improvement of computations over free monads. In *MPC 2008*, volume 5133 of *LNCS*, pages 388–403, 2008.

[46] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003.