An algebraic presentation of predicate logic
(extended abstract)

Sam Staton[⋆]

Computer Laboratory, University of Cambridge

Abstract. We present an algebraic theory for a fragment of predicate logic. The fragment has disjunction, existential quantification and equality. It is not an algebraic theory in the classical sense, but rather within a new framework that we call 'parameterized algebraic theories'.

We demonstrate the relevance of this algebraic presentation to computer science by identifying a programming language in which every type carries a model of the algebraic theory. The result is a simple functional logic programming language.

We provide a syntax-free representation theorem which places terms in bijection with sieves, a concept from category theory.

We study presentation-invariance for general parameterized algebraic theories by providing a theory of clones. We show that parameterized algebraic theories characterize a class of enriched monads.

1 Introduction

This paper is about the following fragment of predicate logic:

$$P, Q \ ::= \ \bot \mid P \vee Q \mid (t = u) \wedge P \mid \exists a.\, P(a) \mid x[t_1, \ldots, t_n]$$

where t, u range over the domain of discourse and x is an n-ary predicate symbol. We provide an algebraic presentation of logical equivalence using a new algebraic framework that we call 'parameterized algebraic theories'. This syntactic framework comes with a straightforward deduction system.

Having introduced the new algebraic framework and presented the theory of predicate logic, we make three further contributions.

1. We consider a programming language in which every type is equipped with the structure of a model of the theory. This yields a simple functional logic programming language (in the sense of [7]). In doing this, we add weight to the slogan of Plotkin and Power [35]: 'algebraic theories determine computational effects'. We demonstrate our language by providing a simple implementation[1].

2. We give a representation theorem for terms in our algebraic theory. There is nothing canonical about a presentation of an algebraic theory: which function

[⋆] Research partially supported by a grant from the Isaac Newton Trust and ERC Project ECSYM.

[1] The implementation is available at `http://www.cl.cam.ac.uk/users/ss368/flp`.

symbols should be used? which assortment of equations? We show that our algebraic presentation of predicate logic is correct by representing terms up-to equivalence as mathematical objects. More precisely, we show that terms up-to equivalence can be understood as *sieves*, a kind of generalized subset.

3. The idea of presentation-invariance of theories is an important one, and so we introduce a general notion of 'clone' for parameterized algebraic theories. Via enriched category theory we obtain a semantic status for the syntactic framework of parameterized algebraic theories. In particular, we show that the parameterized algebraic theories can be understood as a class of enriched monads.

This work thus provides a principled foundation for program equations, with anticipated consequences for program verification and compiler design (see [24]).

In many ways our algebraic presentation of predicate logic is an elaboration of the algebraic theory of semilattices. Recall that the theory of semilattices has a constant $\bot$ and a binary function symbol $\vee$, and the following equations

$$(x \vee y) \vee z \equiv x \vee (y \vee z) \qquad x \vee y \equiv y \vee x \qquad x \vee x \equiv x \qquad \bot \vee x \equiv x. \quad (1)$$

Predicate logic combines these equations with others, such as the equation $x[b] \vee \exists a.\, x[a] \equiv \exists a.\, x[a]$. We introduce the article by considering the three contributions from the simpler perspective of semilattices.

Extending the theory to a programming language (§3). To extend the theory of semilattices to a functional language, we add a constant $\bot$ and a binary operation $\vee$ at each type. The result is a language that is declarative in two ways: first as a functional language, and second in that the semilattice structure provides a kind of non-determinism.

The additional constructs of predicate logic provide further techniques for declarative programming: $(t = u) \wedge P$ can be understood as unification and $\exists a.\, P(a)$ can be understood as introducing a new logic variable. Thus the functional language gains an abstract type `param`, representing the domain of discourse. If the domain of discourse is the Peano natural numbers, we have expressions `z:param` and `s:param→param`. Consider the following recursive program `add` of type `param→param→param→unit`:

$$\mathtt{add}\, a\, b\, c \overset{\text{def}}{=} \big((a = \mathtt{z}) \wedge (b = c) \wedge ()\big) \vee \big(\exists a'.\, \exists b'.\, (a = \mathtt{s}(a')) \wedge (c = \mathtt{s}(c')) \wedge \mathtt{add}\, a'\, b\, c'\big)$$

The program returns if $a+b = c$, and fails if not. Thus functions into `unit` are like predicates. To experiment in more depth we provide a simple implementation in Standard ML.

Representation theorem (§4). Any presentation of an algebraic theory is somewhat arbitrary. We could have presented semilattices (1) using a ternary disjunction, or by replacing associativity with mediality $((v \vee x) \vee (y \vee z) \equiv (v \vee y) \vee (x \vee z))$. What really matters about the theory of semilattices is that to give a term is to give the set of variables that appear in it. For instance, $(x \vee z) \vee v$ and $v \vee ((z \vee x) \vee \bot)$ both use the same variables, and they are provably equal.

When we move to the algebraic theory of disjunctive predicate logic, a representation result is even more desirable, since the axiomatization is more complicated. We no longer have a characterization in terms of sets of variables. Rather, we generalize that analysis by understanding a set of variables as a kind of weakening principle: in any algebraic theory, any term involving those variables can also be considered as a term involving more variables. In the setting of parameterized algebraic theories, the notions of substitution and weakening are more sophisticated. Nonetheless our representation theorem characterizes terms of disjunctive predicate logic as classes of substitutions that satisfy a closure condition. We set this up using category theory, by defining a category whose objects are contexts and whose morphisms are substitutions, so that a class of substitutions is a 'sieve'.

This representation theorem shows that our algebraic theory of disjunctive predicate logic is not just an ad-hoc syntactic gadget: it is mathematically natural. This corroborates the following general hypothesis: algebraic theories for computational effects should have elegant characterizations of terms and free models [35, 30].

Clones (§5). In our third contribution we stay with the theme of presentation invariance, but we study it for parameterized algebraic theories in general. In classical universal algebra, presentation-invariance is studied via clones: closed sets of operations. Recall that an abstract clone is given by a set $T(n)$ for each natural number n, a tuple $\eta_n \in T(n)^n$ for each number n, and a family of functions $\{*_{m,n} : Tm \times (Tn)^m \to Tn\}_{m,n}$, all satisfying some conditions (e.g. [14, Ch. III]).

The terms in the theory of semilattices form a clone: $T(n)$ is the set of terms in n variables, η picks out the terms that are merely variables; and $*$ describes simultaneous substitution. Similarly the subsets form a clone: $T(n)$ is the set of subsets of n, η picks out the singleton subsets, and $*$ is a union construction. These two clones are isomorphic.

Abstract clones can be equivalently described as monoids in a suitable monoidal category, and moreover equivalently described as finitary monads on the category of sets. This provides the connection between Moggi's work on computational monads [32] and the assertion of Plotkin and Power [35] that computational effects determine algebraic theories.

In Section 5, we revisit this situation in the context of parameterized algebraic theories. We provide a general notion of enriched clone. Specialized to enrichment in a presheaf category, this provides a presentation-invariant description of parameterized algebraic theories. For general reasons, enriched clones can be equivalently described as sifted-colimit-preserving enriched monads on a presheaf category. Thus our syntactic framework of parameterized algebraic theories is given a canonical semantic status and a connection with Moggi's work [32].

Acknowledgements Thanks to reviewers for helpful feedback, and to M Fiore, M Hyland, O Kammar, A Kurz, P Levy, P-A Melliès, R Møgelberg, G Plotkin, J Power and T Uustalu for discussions.

2 Presentations of parameterized algebraic theories

We will present disjunctive predicate logic as an algebraic theory within a new framework of parameterized algebraic theories. It is an algebraic theory that takes parameters from another algebraic theory. When we write

$$((a = a) \wedge x) \;\equiv\; x \tag{2}$$

this is a judgement of equality between predicates, and while x is a variable standing for a predicate, a is not: it stands for a term (e.g. a natural number). This phenomenon is common across mathematics. For example, in linear algebra, when we write $a(x + y) \;\equiv\; ax + ay$ the variables x and y have a different status to the scalar parameter a.

Parameterized algebraic theories are not merely 2-sorted theories. In equation (2), the variable x stands for a predicate which might itself have parameters. For example, we can describe the substitutive nature of equality like this:

$$(a = b) \wedge x[a] \;\equiv\; (a = b) \wedge x[b]. \tag{3}$$

The substitution of predicates for variables is now quite elaborate. One instance of equation (3) is $(a = b) \wedge (a = a) \wedge y \;\equiv\; (a = b) \wedge (b = a) \wedge y$, under the assignment $x[c] \mapsto (c = a) \wedge y$.

Quantifiers bind parameters, requiring equations like this:

$$x[b] \vee \exists a.\, x[a] \;\equiv\; \exists a.\, x[a] \tag{4}$$

in which the parameter b is free while the parameter a is bound. We work up-to α-equivalence ($\exists a.\, x[a] = \exists b.\, x[b]$) and substitution must avoid variable capture: we must change bound variables before substituting $x[c] \mapsto (c = a) \wedge y$ in (4).

In this section we give a technical account of what constitutes a signature of parameters (§2.1) and an algebraic theory parameterized in that signature (§2.2). In the example of predicate logic, terms of the signature of parameters describe the domain of discourse. Terms of the parameterized theory are predicates over the domain of discourse, or alternatively simple logic programs over the domain of discourse.

The general framework is essentially a single-sorted version of the 'effect theories' developed in joint work with Møgelberg [31], based on proposals by Plotkin and Pretnar [34, 37].

2.1 Signatures of parameters

Recall the notion of signature used in universal algebra. A signature is given by a set of function symbols, $\mathrm{f}, \mathrm{g}, \ldots$, each equipped with an arity, which is a natural number specifying how many arguments it takes. From a signature we can build terms-in-context:

$$\frac{}{\vec{a} \vdash a_i}\;(i \leqslant |\vec{a}|) \qquad\qquad \frac{\vec{a} \vdash t_1 \;\ldots\; \vec{a} \vdash t_n}{\vec{a} \vdash \mathrm{f}(t_1, \ldots, t_n)}\;(\mathrm{f}\text{ has arity } n)$$

(We write $\vec{a}$ for a list of variables $a_1, a_2, \ldots$ and $|\vec{a}|$ for the length of the list.)

Here are some simple examples of signatures:

- The empty signature has no function symbols.
- The signature of natural numbers has a constant symbol z (i.e. a function symbol with arity 0) and a unary function symbol s.
- The domain of a database can be described by a signature with a constant symbol for each element of the domain.

2.2 Parameterized algebraic theories

Let $\mathbb{S}$ be a signature as in Sect. 2.1. A *signature* $\mathbb{T}$ *parameterized by* $\mathbb{S}$ is given by a set of function symbols $\mathrm{F}, \mathrm{G}, \ldots$ each equipped with an arity, written $\mathrm{F} : (p \mid \vec{n})$, where p is a natural number and $\vec{n}$ a list of natural numbers. We distinguish between parameters and arguments. The number p determines how many parameters (from $\mathbb{S}$) the function symbol takes. The length of the list $\vec{n}$ determines how many arguments (from $\mathbb{T}$) the function symbol takes, and each component of the list determines the valence or binding depth of that argument. For instance, the arity of binary join is $\vee : (0 \mid [0, 0])$, since it takes two arguments with no variable binding; the arity of the equality predicate $(a = b) \wedge x$ is $(2 \mid [0])$ as it takes two parameters (a and b) and one argument (x); the arity of the quantifier is $\exists : (0 \mid [1])$ since it takes one argument with a bound variable.

From a parameterized signature we can build parameterized terms in context. A context $\Gamma \mid \Delta$ for a parameterized term has two parts, comprising two kinds of variable. The first part Γ is a finite set of variables ranging over parameters. The second part Δ is a finite set of variables ranging over terms, each equipped with a natural number, which specifies how many parameters the variable takes. As usual, we write a set of variables as a list with the convention that all variables are different. The terms-in-context are built from variables, function symbols and terms from the signature of parameters, using the following two rules.

$$\frac{\Gamma \vdash t_1 \ \ldots\ \Gamma \vdash t_{n_i}}{\Gamma \mid x_1 \colon n_1 \ldots x_k \colon n_k \vdash x_i[t_1 \ldots t_{n_i}]}$$
$$\frac{\Gamma \vdash t_1 \ \ldots\ \Gamma \vdash t_p \quad \Gamma, a_{1,1} \ldots a_{1,n_1} \mid \Delta \vdash u_1 \ \ldots\ \Gamma, a_{k,1} \ldots a_{k,n_k} \mid \Delta \vdash u_k}{\Gamma \mid \Delta \vdash \mathrm{F}(t_1, \ldots, t_p, \vec{a}_1.\, u_1, \ldots, \vec{a}_k.\, u_k)} \tag{5}$$

where $\mathrm{F} : (p \mid [n_1 \ldots n_k])$. We work up-to α-renaming the binders $\vec{a}$. Notice our distinction between judgements of parameters ($\Gamma \vdash t$) and of terms ($\Gamma \mid \Delta \vdash t$).

Definition 1. *A presentation of a parameterized algebraic theory is a parameterized signature together with a collection of equations, where an equation is a pair of two terms in the same context.*

We define substitution for the two kinds of variable in a standard way, so as to give the following derived rules.

$$\frac{\Gamma \vdash t \quad \Gamma, a \mid \Delta \vdash u}{\Gamma \mid \Delta \vdash u[{}^{t}/_{a}]} \qquad \frac{\Gamma, a_1 \ldots a_n \mid \Delta \vdash t \quad \Gamma \mid \Delta, x \colon n \vdash u}{\Gamma \mid \Delta \vdash u[{}^{a_1 \ldots a_n . t}/_{x}]} \tag{6}$$

Recall that all variables in a context are distinct. Thus substitution is capture-avoiding unless the capture is explicit.

We form a deductive system from a presentation by combining all substitution-instances of the equations with the usual laws of reflexivity, symmetry, transitivity, and the following congruence rule:

$$\frac{\Gamma \vdash t_1 \;\dots\; \Gamma \vdash t_p \qquad \Gamma, \vec{a}_1 \mid \Delta \vdash u_1 \equiv u_1' \;\;\dots\;\; \Gamma, \vec{a}_k \mid \Delta \vdash u_k \equiv u_k'}{\Gamma \mid \Delta \vdash \mathrm{F}(t_1, \dots, t_p, \vec{a}_1.\, u_1, \dots, \vec{a}_k.\, u_k) \equiv \mathrm{F}(t_1, \dots, t_p, \vec{a}_1.\, u_1', \dots, \vec{a}_k.\, u_k')} \tag{7}$$

2.3 Presentation of predicate logic

We now describe disjunctive predicate logic as a parameterized algebraic theory. Predicates range over a domain of discourse, which is a signature of parameters. In this paper we only consider two signatures of parameters: (1) the empty one; (2) the signature of natural numbers; but other signatures can be accommodated straightforwardly.

The parameterized algebraic theory is given in Figure 1. It has a constant symbol $\bot$ and a binary function symbol $\vee$. There is a unary function symbol $(=:=)$ which takes two parameters, and a function symbol $\exists$ which binds a parameter. We use an infix notation for $\vee$ and $=:=$. Term formation (5) yields

$$\frac{}{\Gamma|\Delta \vdash \bot} \qquad \frac{\Gamma|\Delta \vdash t \quad \Gamma|\Delta \vdash u}{\Gamma|\Delta \vdash t \vee u} \qquad \frac{\Gamma \vdash t_1 \quad \Gamma \vdash t_2 \quad \Gamma|\Delta \vdash u}{\Gamma|\Delta \vdash (t_1 =:= t_2)u} \qquad \frac{\Gamma, a \mid \Delta \vdash u}{\Gamma \mid \Delta \vdash \exists a.\, u}$$

The string $(a =:= b)x[]$ can be thought of as the predicate $(a = b) \wedge x[]$, or as the logic program 'unify a and b, and then continue as x'. Note that we do not have arbitrary conjunctions in our algebraic theory. However, if we understand predicate variables as continuation variables, then substitution behaves like conjunction: e.g. given $\vec{a} \mid x : 0 \vdash t$ and $\vec{a} \mid - \vdash u$, the expression $t[{}^{u}/_{x}]$ can be understood as $t \wedge u$. We return to the idea of 'conjunction as sequential composition' in Section 3.

Laws 1–4 are the laws of semilattices. Laws 5–7 are basic axioms for equality. If we write '$t \leqslant u$' for '$t \vee u \equiv u$', then Laws 8, 9, 10 can be written $\exists a.\, x[] \leqslant x[]$, $x[b] \leqslant \exists a.\, x[a]$, $(a =:= b)x[] \leqslant x[]$. Laws 11 and 12 say that $\vee$ commutes over $=:=$ and $\exists$. In fact, all the operations commute over each other. For instance,

$$a, b \mid - \vdash (a =:= b)\bot \overset{4}{\equiv} \bot \vee (a =:= b)\bot \overset{2}{\equiv} (a =:= b)\bot \vee \bot \overset{10}{\equiv} \bot.$$

Laws 13 and 14 are axioms of Peano arithmetic. Law 15 is 'occurs check'.

We can derive $a, b \mid y : 0 \vdash (a =:= b)y[] \equiv (b =:= a)y[]$. First, notice that $a, b \mid y \colon 0 \vdash (a =:= b)(a =:= a)y[] \equiv (a =:= b)(b =:= a)y[]$ is an instance of Law 6, under the substitution $[{}^{c.((c=:=a)y[])}/_{x}]$. Now,

$$\begin{aligned} a, b \mid y : 0 \vdash (a =:= b)y &\overset{5}{\equiv} (a =:= b)(a =:= a)y[] \overset{6}{\equiv} (a =:= b)(b =:= a)y[] \\ &\overset{7}{\equiv} (b =:= a)(a =:= b)y[] \overset{6}{\equiv} (b =:= a)(b =:= b)y[] \overset{5}{\equiv} (b =:= a)y[]. \end{aligned}$$

Signature:		$\bot : (0 \mid [])$ $\quad \vee : (0 \mid [0,0])$ $\quad (=:=) : (2 \mid [0])$ $\quad \exists : (0 \mid [1])$
Equations:	1.	$- \mid x, y, z : 0 \vdash (x[] \vee y[]) \vee z[] \equiv x[] \vee (y[] \vee z[])$
	2.	$- \mid x, y : 0 \vdash x[] \vee y[] \equiv y[] \vee x[]$
	3.	$- \mid x : 0 \vdash x[] \vee x[] \equiv x[]$
	4.	$- \mid x : 0 \vdash \bot \vee x[] \equiv x[]$
	5.	$a \mid x : 0 \vdash (a =:= a)x[] \equiv x[]$
	6.	$a, b \mid x : 1 \vdash (a =:= b)x[a] \equiv (a =:= b)x[b]$
	7.	$a, b, c, d \mid x : 0 \vdash (a =:= b)(c =:= d)x \equiv (c =:= d)(a =:= b)x$
	8.	$- \mid x : 0 \vdash (\exists a.\, x[]) \vee x[] \equiv x[]$
	9.	$b \mid x : 1 \vdash x[b] \vee \exists a.\, x[a] \equiv \exists a.\, x[a]$
	10.	$a, b \mid x : 0 \vdash ((a =:= b)x[]) \vee x[] \equiv x[]$
	11.	$a, b \mid x, y : 0 \vdash (a =:= b)(x[] \vee y[]) \equiv ((a =:= b)x[]) \vee ((a =:= b)y[])$
	12.	$- \mid x, y : 1 \vdash \exists a.\, (x[a] \vee y[a]) \equiv \exists a.\, x[a] \vee \exists a.\, y[a]$
Additional equation schemes when the signature of parameters is natural numbers:		
	13.	$a \mid x : 0 \vdash (\mathrm{z} =:= \mathrm{s}(a))x[] \equiv \bot$
	14.	$a, b \mid x : 0 \vdash (\mathrm{s}(a) =:= \mathrm{s}(b))x[] \equiv (a =:= b)x[]$
	15.	$a \mid x : 0 \vdash (a =:= \mathrm{s}^n(a))x[] \equiv \bot \quad \forall n > 0$, where $\mathrm{s}^2(a) = \mathrm{s}(\mathrm{s}(a))$, etc.

Fig. 1. A presentation of the parameterized theory of disjunctive predicate logic.

A subtle point is that the context cannot be omitted. When the signature of parameters is empty, the equation $- \mid x : 0 \vdash \exists a.\, x[] \equiv x[]$ is not derivable, although we do have $a \mid x : 0 \vdash \exists a.\, x[] \overset{9}{\equiv} x[] \vee \exists a.\, x[] \overset{2}{\equiv} (\exists a.\, x[]) \vee x[] \overset{8}{\equiv} x[]$. (To instantiate law 9 we applied the substitution $[{}^{a.x[]}/_{x}]$.)

2.4 Other examples of parameterized algebraic theories

Any classical algebraic theory can be understood as a parameterized one in which the function symbols take no parameters and all the valences are 0. A slightly more elaborate example is the 2-sorted theory of modules over an unspecified ring, which is an algebraic theory parameterized in the signature of rings.

For any signature we have the following theory of computations over a memory cell. There are two function symbols in the parameterized algebraic theory: $\mathsf{w} : (1 \mid [0])$ and $\mathsf{r} : (0 \mid [1])$. The intuition is that $\mathsf{w}(a, x[])$ writes a to memory and continues as x, while $\mathsf{r}(a.\, x[a])$ reads from memory, binds the result to a, and continues as x. The presentation has three equations (c.f. [35]):

$$x[] \equiv \mathsf{r}(a.\, \mathsf{w}(a, x[])) \qquad \mathsf{w}(a, \mathsf{w}(b, x[])) \equiv \mathsf{w}(b, x[]) \qquad \mathsf{w}(a, \mathsf{r}(b.\, x[b])) \equiv \mathsf{w}(a, x[a])$$

The first equation says that if you read a, then write a, then continue as x, then you may as well just run x. The second equation says that if you write to memory twice then it is the second write that counts. The third equation says that if you read b after a write a, then b will be a and the read is unnecessary.

If the parameterizing signature is the signature of natural numbers then there is an expression $\mathsf{r}(a.\,\mathsf{w}(\mathrm{s}(a), x))$, which increments the memory and continues as x.

2.5 Set-theoretic models

We briefly discuss models of parameterized algebraic theories. As we will see, the set-theoretic notion of model is not a complete way to understand theories, but it is sound and so we are able to use it to verify consistency.

A set-theoretic structure for a parameterized algebraic theory is given by two sets: a set π of parameters, and a set M which is the carrier. For each n-ary function symbol f in the signature of parameters, a function $\mathrm{f} : \pi^n \to \pi$ must be given. For each function symbol $\mathrm{F} : (p \mid \vec{n})$ in the theory, a function $\mathrm{F} : \pi^p \times \prod_{j=1}^{|\vec{n}|} M^{(\pi^{n_j})} \to M$ must be given. Here $M^{(\pi^{n_j})}$ is the set of functions from (π^{n_j}) to M. It is routine to extend this to all terms, interpreting a term-in-context $\vec{a} \mid \vec{x} : \vec{n} \vdash t$ as a function $[\![t]\!] : \pi^{|\vec{a}|} \times \prod_{j=1}^{|\vec{n}|} M^{(\pi^{n_j})} \to M$. We say that a structure is a *model* when for each equation $\Gamma \mid \Delta \vdash t \equiv u$ in the theory the corresponding functions are equal: $[\![t]\!] = [\![u]\!]$. This interpretation is sound:

Proposition 1. *If $\vec{a} \mid \vec{x} : \vec{n} \vdash t \equiv u$ is derivable in a parameterized algebraic theory, then $[\![t]\!] = [\![u]\!]$ in all models.*

Consider the theory of disjunctive predicate logic over the signature for natural numbers. We can let π be the set $\mathbb{N}$ of natural numbers, and then we must provide a set M together with an element $\bot$ and three functions: $\vee : M \times M \to M$, $(\text{=:=}) : \mathbb{N} \times \mathbb{N} \times M \to M$, and $\exists : M^{\mathbb{N}} \to M$. In fact, this forms a model if and only if M is a countable semilattice, with $\bot$ and $\vee$ supplying the finite joins and $\exists$ supplying the countably infinite joins. By the soundness result, the consistency of our theory is witnessed by giving a non-trivial countable-join-semilattice.

There are two things that are unsatisfactory about set-theoretic models of disjunctive predicate logic. First, it is often best not to think of $\exists$ as a countable join: in logic programming it is better to think of $\exists$ as introducing a free logic variable. Second, the interpretation of (=:=) is necessarily fixed, as we now explain.

2.6 Incompleteness of set-theoretic models

Classical universal algebra is complete for set-theoretic models: if an equation is true in all algebras, then it is derivable. However, set-theoretic models are *not* complete for parameterized algebraic theories: some equations are true in all set-theoretic models but not derivable. In any set-theoretic structure for disjunctive predicate logic, the three equations

$$(a \text{ =:= } a)x[\,] \equiv x[\,] \qquad (a \text{ =:= } b)\bot \equiv \bot \qquad (a \text{ =:= } b)x[a] \equiv (a \text{ =:= } b)x[b] \tag{8}$$

entirely determine (=:=). This is because two elements a, b of π are either equal or not equal. In any structure satisfying the three equations we *must* have

$(a =:= b)(x) = x$ when $a = b$ and $(a =:= b)(x) = \bot$ when $a \neq b$. The first case, when $a = b$, is the first equation in (8). To establish the second case, when $a \neq b$, we define a function $\delta^a_x : \pi \to M$ by setting $\delta^a_x(c) \overset{\text{def}}{=} x$ if $c = a$ and $\delta^a_x(c) \overset{\text{def}}{=} \bot$ if $c \neq a$, so that $(a =:= b)(x) = (a =:= b)\delta^a_x(a) = (a =:= b)\delta^a_x(b) = (a =:= b)\bot = \bot$.

Thus any set-theoretic structure satisfying the laws in (8) will also satisfy law 7 in Figure 1: $(a =:= b)(c =:= d)x[] \equiv (c =:= d)(a =:= b)x[]$. But that is not derivable from (8). To resolve this incompleteness we must move to a constructive set theory in which equality is not two-valued, which is the essence of Section 5.

2.7 Other equational approaches to logic

The syntax for parameterized algebraic theories is reminiscent of logical frameworks such as Type Framework [4] although it is much simpler.

Our syntax is also similar to Aczel's syntax [1] which forms the basis of the second-order algebraic theories of Fiore et al. [18, 19]. The key difference is that we do not allow second-order variables to take second-order terms as parameters, e.g. $x[y[] \vee z[]]$. This restriction allows us to make a connection with programming languages (§3) and a simple categorical model theory (§5).

By far the most studied equational approach to logic is Tarski's cylindric algebra. Cylindric algebra encodes the binding structures of predicate logic into classical universal algebra. Although cylindric algebra can provide a foundation for concurrent constraint programming [39], it does not extend easily to higher typed programming languages. Our parameterized algebraic theory does (§3).

There have been several proposals for 'nominal algebra' [13, 20]. Gabbay and Matthijssen used this to describe first-order logic [20] and the author has earlier developed a nominal-style presentation of semilattices and equality [41, §6]. However, it is unclear how to combine a theory of nominal algebra with programming language primitives in a canonical way. Although the free model construction [13] yields a monad on the category of nominal sets, it does not yield a strength for the monad, and so Moggi's framework [32] does not apply to nominal algebraic theories. Moggi's framework *does* apply to parameterized algebraic theories (§5).

Kurz and Petrişan [28] have shown how to understand cylindric algebra and nominal algebra within the framework of presheaf categories. In Section 5 we will demonstrate that parameterized algebraic theories can be understood as algebraic theories enriched in a presheaf category.

Bronsard and Reddy [12] axiomatized a theory of disjunction, conjunction, existentials and if-then-else, and they gave a completeness result for domain theoretic models, providing a more axiomatic account than earlier domain theoretic models of logic programming (e.g. [21, 33, 38]). Our work strengthens that early development by moving away from concrete models, which are not a complete way to study algebraic theories with binding (see §2.6). This allows us to give a canonical status to our algebraic theory (Theorem 1).

3 Extending the algebraic theory to a programming language

Plotkin and Power have proposed that algebraic theories determine notions of computational effect [35]. One can build a higher-order functional programming language in which each type is a model of the algebraic theory, so that the algebraic structure provides the impurities in the functional programming language. We demonstrate this by picking out a fragment of Standard ML, eliciting the algebraic structure of each type by identifying suitable generic effects [36]. For instance, the generic effect for disjunction, $\vee$, is an impure function $\underline{\texttt{choose}}$: `unit → bool`, to be thought of as returning an undetermined boolean. Our implementation is thus a structure for the following ML signature.

```
infix 3 =:=
sig
  (* SIGNATURE OF PARAMETERS *)           (* MAIN SIGNATURE *)
  (* param is the domain                  (* Presented using
              of discourse *)                generic effects *)
  type param                              val choose : unit → bool
  val succ : param → param                val fail : unit → 'a
  val zero : param                        val =:= : param * param → unit
                                          val free : unit → param end
```

The algebraic operations at each type can be recovered from the generic effects:

$$\texttt{t} \vee \texttt{u} \overset{\text{def}}{=} \texttt{if } \underline{\texttt{choose}}\texttt{() then t else u} \qquad \bot \overset{\text{def}}{=} \underline{\texttt{fail}}\texttt{()}$$

$$(\texttt{a =:= b})(\texttt{t}) \overset{\text{def}}{=} \texttt{a=:=b ; t} \qquad \exists \texttt{a.t} \overset{\text{def}}{=} \texttt{let val a=}\underline{\texttt{free}}\texttt{() in t end}$$

In this ML signature, there are two ways to define addition, firstly as a function:

```
- fun add(a,b) = if choose () then a =:= zero ; b
                 else let val a' = free()
                      in  a =:= succ a' ; add(a',succ(b)) end
val add = fn : param * param → param
```

and secondly as a predicate:

```
- fun add'(a,b,c) = if choose () then a =:= zero ; b =:= c
                    else let val a' = free() val c' = free() in
                         a =:= succ a' ; c =:= succ c' ; add'(a',b,c') end
val add' = fn : param * param * param → unit
```

This demonstrates a type isomorphism:

```
- fun iso f a = let val b = free() in (b,f(a,b)) end
val iso = fn : ('a * param → 'c) → ('a → param * 'c)
- fun inverse g a = let val (b',c) = g(a) in b =:= b' ; c end
val inverse = fn : ('a → param * 'c) → ('a * param → 'c)
```

Notice that sequencing (semicolon) is like conjunction.

Our implementation of the ML signature (Appendix A) uses references and `callcc`. One way to view the laws in Figure 1 is as an axiomatic account of which optimizations are allowed [24]. Our implementation certainly doesn't implement Law 2 (commutativity), which would need parallel execution. Does our implementation capture the other laws? A proper answer to this question would need a theory of observational equivalence for this fragment of ML, for instance extending [22] to parameterized algebraic theories, which we leave for future work.

Our lightweight approach to implementation is partly inspired by Eff [9], a new language for algebraic effects. A related approach is to use monads in Haskell, following [11, 40], since in our language the type construction `unit → (-)` is equipped with the structure of a monad.

4 Representation of terms

In Figure 1 we have presented an algebraic theory of disjunctive predicate logic. In Section 3 we have seen that the theory provides a reasonable account of the basic phenomena in logic programming. However, the choice of the presentation, which operations and which equations, is somewhat arbitrary. We now justify the theory by giving a canonical representation of the terms modulo the equations.

As a first step, we consider the theory of semilattices, the fragment of disjunctive predicate logic restricted to $\vee$ and $\perp$. A term built from $\vee$ and $\perp$ is determined by the variables that appear in the term. For instance, the terms $- \mid v, x, y, z : 0 \vdash (x[] \vee z[]) \vee v[]$ and $- \mid v, x, y, z : 0 \vdash v[] \vee ((z[] \vee x[]) \vee \perp)$ both contain the same variables $\{v, x, z\}$, and they are equal. We are able to get a similar result for full disjunctive predicate logic, but it is more complicated. For instance, consider the following term.

$$- \mid x : 2 \vdash \exists a.\, x[a, a] \tag{9}$$

To understand this term as a 'subset' of $\{x\}$, we make the following observation. The subset $\{v, x, z\}$ of $\{v, x, y, z\}$ provides a weakening principle: for any algebraic theory, any term in context $\{v, x, z\}$ can also be understood as a term in context $\{v, x, y, z\}$. The term (9) also describes a weakening principle: in any parameterized algebraic theory, a term in context $(x : 1)$ can be understood as a term in context $(x : 2)$, by substituting every occurrence of $x[t]$ by $x[t, t]$.

This motivates us to define a category whose objects are contexts and whose morphisms are substitutions. We investigate sieves, which are sets of substitutions subject to a closure condition. Our representation theorem provides a correspondence between terms of disjunctive predicate logic and sieves.

Subsets and sieves. The concept of 'subset' is not a priori a category-theoretic notion because it is defined in terms of elements of sets rather than morphisms. One category-theoretic notion of 'subset' is the notion of sieve.

Definition 2. *Let $\mathcal{C}$ be a category, and let X be an object in $\mathcal{C}$. A* sieve *S on X is a class of morphisms with codomain X which is closed under precomposition: $f \in S \implies fg \in S$. Every morphism $f\colon Y \to X$ generates a sieve on X as follows: $[f] \overset{\text{def}}{=} \{g\colon Z \to X \mid \exists h : Z \to Y.\, g = fh\}$. A sieve S on X is* singleton-generated *if it is of this form.*

For two morphisms $f\colon Y \to X$ and $f'\colon Y' \to X$, the following are equivalent: (i) they generate the same sieve ($[f] = [f']$); (ii) $f \in [f']$ and $f' \in [f]$; (iii) there are morphisms $g\colon Y \to Y'$ and $g'\colon Y' \to Y$ such that $f = f'g$ and $f' = fg'$.

Singleton generated sieves can be understood as a category-theoretic version of subset. For instance, in the category of sets, a function $f\colon X \to Y$ generates the same sieve of Y as its image $f(X) \rightarrowtail Y$ (assuming choice). More generally, a singleton-generated sieve on an object X of a category $\mathcal{C}$ is a subobject of X in the regularization of $\mathcal{C}$ [23, A1.3.10(d)]. The importance of sieves and presheaves to logic programming has been observed earlier [16, 26, 27].

A category of contexts. We will describe a correspondence between terms of disjunctive predicate logic and sieves in a category whose objects are contexts and whose morphisms are substitutions.

In a parameterized algebraic theory, the context has two components $(\Gamma|\Delta)$: variables in Γ ranging over parameters and variables in Δ ranging over terms. The objects of our category focus on the second component Δ. Since the names of variables are irrelevant, we represent a context by a list of numbers.

The morphisms of our category are simultaneous substitutions. To motivate, consider the following derived typing rule for substituting variables for variables.

$$\frac{\vec{a} \vdash t_1 \;\dots\; \vec{a} \vdash t_n \qquad - \mid \vec{x} : \vec{m}, y : |\vec{a}| \vdash u}{- \mid \vec{x} : \vec{m}, z : n \vdash u[{}^{\vec{a}.z[t_1,\dots,t_n]}/_{y}]} \tag{10}$$

Definition 3. *Let $\mathbb{S}$ be a signature (of parameters, as in §2.1). The objects of the category $\mathbf{Ctx}(\mathbb{S})$ are lists of natural numbers. A morphism $\vec{m} \to \vec{n}$ comprises a function $f\colon |\vec{m}| \to |\vec{n}|$ together with, for $1 \leqslant i \leqslant |\vec{m}|$ and $1 \leqslant j \leqslant n_{f(i)}$, a term $a_1, \dots a_{m_i} \vdash t_{i,j}$ in the signature of parameters. Morphisms compose by composing functions and substituting terms for variables. The identity morphism is built from variables.*

The category theorist will recognize $\mathbf{Ctx}(\mathbb{S})$ as the free finite coproduct completion of the Lawvere theory for the signature $\mathbb{S}$. Lawvere theories are widely regarded as important to the foundations of logic programming (e.g. [8, 26, 6, 27]).

For any term $- \mid \vec{x}\colon \vec{m} \vdash u$ in any parameterized algebraic theory, and any morphism $(f, \vec{t}) : \vec{m} \to \vec{n}$ in $\mathbf{Ctx}(\mathbb{S})$, notice that we can build a term by substitution, following (10),

$$- \mid \vec{y}\colon \vec{n} \vdash (f, \vec{t}) \bullet u \;\stackrel{\text{def}}{=}\; u[{}^{a_1 \dots a_{m_1}.y_{f(1)}[\vec{t}_1]}/_{x_1}] \dots [{}^{a_1 \dots a_{m_{|\vec{m}|}}.y_{f(|\vec{m}|)}[\vec{t}_{|\vec{m}|}]}/_{x_{|\vec{m}|}}]$$

so that substitution respects composition: $(g, \vec{v}) \bullet ((f, \vec{u}) \bullet t) = ((g, \vec{v}) \cdot (f, \vec{u})) \bullet t$.

Representation theorem. We now state our representation theorem for the theory of disjunctive predicate logic. We focus on terms with no free parameters, returning to this point later.

Given a morphism $(f, \{b_1, \dots, b_{m_i} \vdash t_{i,j}\}_{i \leqslant |\vec{m}|, j \leqslant n_{f(i)}}) : \vec{m} \to \vec{n}$ in $\mathbf{Ctx}(\mathbb{S})$ we define the following term:

$$\wr f, \vec{t} \int \;\stackrel{\text{def}}{=}\; - \mid \vec{x} : \vec{n} \vdash \bigvee_{i=1}^{|\vec{m}|} \exists b_1. \dots \exists b_{m_i}. x_{f(i)}[t_{i,1} \dots t_{i,n_{f(i)}}] \tag{11}$$

Theorem 1. *Let $\mathbb{S}$ be either the empty signature or the signature for natural numbers. Let $\vec{n}$ be a list of numbers. The construction $\wr - \int$ induces a bijective correspondence between:*

- *terms in context, $- \mid \vec{x} : \vec{n} \vdash t$, modulo the equivalence relation in Figure 1;*
- *singleton-generated sieves on $\vec{n}$ in the category* $\mathbf{Ctx}(\mathbb{S})$.

(The Theorem can be established for a different signature of parameters by finding appropriate analogues of Laws 13–15.)

Outline proof of Theorem 1. To prove the representation theorem, we first characterize morphisms into $\vec{n}$ as terms modulo a fragment of the theory. We then show that two morphisms generate the same sieve if and only if the corresponding terms are equal in the full theory.

For the first step, we show that the construction $\wr - \int$ determines a bijective correspondence between morphisms into $\vec{n}$ and terms-in-context modulo a fragment of the theory in Figure 1. The fragment is given by laws 1, 4, 5, 6, 7, 11 and 12 from Figure 1, and schemes 13–15 where relevant, together with the following five laws, which are derivable from laws 2, 3 and 8–10 in Figure 1 but not from the other laws.

$$- \mid x\colon 0 \vdash \bot \vee x \equiv x \qquad a, b \mid - \vdash (a \mathrel{=:=} b)\bot \equiv \bot$$

$$- \mid - \vdash \exists a.\, \bot \equiv \bot \qquad b, c \mid x\colon 1 \vdash (b \mathrel{=:=} c)\exists a.\, x[a] \equiv \exists a.\, (b \mathrel{=:=} c)x[a]$$

$$b \mid x\colon 1 + n \vdash \exists c_1. \dots \exists c_n.\, x[b, \vec{c}] \equiv \exists a.\, \exists c_1. \dots \exists c_n.\, (a \mathrel{=:=} b)x[a, \vec{c}]$$

The first four laws are commutativity conditions; the last one is roughly introduction and elimination for $\exists$.

Note that every term can be rewritten to the form in (11) using the laws in this fragment of the theory. We first pull the disjunctions to the front, then the existentials, and then we use the remaining axioms to rearrange and eliminate the equality tests. We thus have a bijective correspondence between terms in context $\vec{n}$ modulo this fragment of the theory, and morphisms into $\vec{n}$ in $\mathbf{Ctx}(\mathbb{S})$.

The second step is to show that that two morphisms in $\mathbf{Ctx}(\mathbb{S})$ determine the same sieve if and only if the corresponding terms (11) can be proven equal using the laws in Figure 1. We show that $[f, \vec{t}] \subseteq [g, \vec{u}]$ if and only if $\wr f, \vec{t} \int \leqslant \wr g, \vec{u} \int$.

Terms with free parameters. Let $T^{\exists\vee}(p|\vec{n})$ be the set of terms in the context $(a_1, \dots, a_p|\vec{x} : \vec{n})$, modulo the equivalence relation. If we write $\mathrm{Sieves}_1(\vec{n})$ for the set of singleton-generated sieves on $\vec{n}$ in $\mathbf{Ctx}(\mathbb{S})$, then Theorem 1 provides a natural bijection $T^{\exists\vee}(0|\vec{n}) \cong \mathrm{Sieves}_1(\vec{n})$.

We now briefly consider the situation where the parameter context p is non-empty, by exhibiting a bijection $T^{\exists\vee}(p|[n_1, \dots, n_k]) \cong T^{\exists\vee}(0|[p + n_1, \dots, p + n_k])$.

To go from left to right we substitute $\vec{a}, \vec{b} \mid x_i\colon p + n_i \vdash x_i[b_1 \dots b_p, a_1 \dots a_{n_i}]$ for each variable $x_i[a_1 \dots a_{n_i}]$ $(i \leqslant k)$, and then existentially quantify all the free variables $\vec{b}$. From right to left we substitute

$$a_1 \mathinner{..} a_{n_i} \mathinner{..} a_{p+n_i+n}, b_1 \mathinner{..} b_p \mid x_i\colon m_i \vdash (a_{n_i+1} \mathrel{=:=} b_1) \dots (a_{n_i+p} \mathrel{=:=} b_p)x_i[a_1 \dots a_{n_i}]$$

for each variable x_i, yielding a term with free variables $\vec{b}$ (c.f. `iso` in §3).

5 Enriched clones

We conclude by giving our general syntactic framework of parameterized algebraic theories a canonical status by reference to enriched category theory. The importance of enriched monads for programming language semantics has long been recognized [32]. We show that parameterized algebraic theories characterize a class of enriched monads.

In the previous section we described a bijection between the set of terms of disjunctive predicate logic and the set of sieves in a category of contexts. However, the set of terms is not a mere set: it also has a substitution structure. We characterize this abstractly by introducing enriched clones.

Definition 4. *Let $(\mathcal{V}, \otimes, I)$ be a symmetric monoidal category. Let $\mathcal{C}$ be a $\mathcal{V}$-enriched category, and $J\colon \mathcal{A} \subseteq \mathcal{C}$ be a full sub-$\mathcal{V}$-category. An* enriched clone *is given by*
1. For each $A \in \mathcal{A}$, an object TA in $\mathcal{C}$;
2. A morphism $\eta_A : I \to \mathcal{C}(JA, TA)$ in $\mathcal{V}$ for all $A \in \mathcal{A}$;
3. A morphism $*_{A,B} : \mathcal{C}(JA, TB) \to \mathcal{C}(TA, TB)$ in $\mathcal{V}$ for all $A, B \in \mathcal{A}$
such that the following diagrams commute:

$$\begin{tikzcd}
I \otimes \mathcal{C}(JA,TB) \arrow[r, "\eta\otimes *"] \arrow[dr, "\lambda"'] & \mathcal{C}(JA,TA) \otimes \mathcal{C}(TA,TB) \arrow[d, "composition"] \\
 & \mathcal{C}(JA,TB)
\end{tikzcd}
\qquad
\begin{tikzcd}
I \arrow[r, "\eta_A"] \arrow[dr, "\mathrm{id}_{TA}"'] & \mathcal{C}(JA,TA) \arrow[d, "*"] \\
 & \mathcal{C}(TA,TA)
\end{tikzcd}$$

$$\begin{tikzcd}
\mathcal{C}(JA,TB) \otimes \mathcal{C}(JB,TC) \arrow[r, "*"] \arrow[d, "*\otimes *"'] & \mathcal{C}(JA,TB) \otimes \mathcal{C}(TB,TC) \arrow[r, "comp"] & \mathcal{C}(JA,TC) \arrow[d, "*"] \\
\mathcal{C}(TA,TB) \otimes \mathcal{C}(TB,TC) \arrow[rr, "composition"'] & & \mathcal{C}(TA,TC)
\end{tikzcd}$$

The original notion of abstract clone (e.g. [14, Ch. III]) appears when $\mathcal{V} = \mathcal{C} = \mathcal{S}et$ and $\mathcal{A}$ comprises natural numbers considered as sets. When $\mathcal{V} = \mathcal{S}et$ then enriched clones have been called 'Kleisli structures' (e.g. [15, §7]) and 'relative monads' [5].

We now turn to parameterized algebraic theories. The signature of parameters induces a Lawvere theory $\mathbb{S}$ which is a category whose objects are natural numbers and where a morphism $m \to n$ is a family of n terms over m parameter variables. We are interested in the category $\hat{\mathbb{S}}$ of presheaves on the Lawvere theory $\mathbb{S}$, that is, the category of contravariant functors $\mathbb{S}^{\mathrm{op}} \to \mathcal{S}et$ and natural transformations between them. As we will see shortly, a presheaf can be understood as a set with substitution structure (see also [26, 17]). Notice that $\mathbf{Ctx}(\mathbb{S})$ (Def. 3) can be understood as a full subcategory of $\hat{\mathbb{S}}$ once we understand a context $[n_1, \dots, n_k]$ as the presheaf $\coprod_{i=1}^{k} \mathbb{S}(-, n_i)$. We let $J\colon \mathbf{Ctx}(\mathbb{S}) \subseteq \hat{\mathbb{S}}$ be the embedding. To put it another way, $\mathbf{Ctx}(\mathbb{S})$ is the completion of $\mathbb{S}$ under coproducts, and $\hat{\mathbb{S}}$ is the completion under all colimits. We consider $\hat{\mathbb{S}}$ as a cartesian closed category, i.e. self-enriched. The function space $\hat{\mathbb{S}}(J\vec{n}, F)$ is given by context extension: $\hat{\mathbb{S}}(J\vec{n}, F)(p) = \prod_{i=1}^{|\vec{n}|} F(p + n_i)$.

Every presentation of a parameterized algebraic theory gives rise to an enriched clone for $\mathcal{V} = \mathcal{C} = \hat{\mathbb{S}}$ and $\mathcal{A} = \mathbf{Ctx}(\mathbb{S})$.

1. The data $T(\vec{n})$ assigns a presheaf to each context $\vec{n}$. The set $(T(\vec{n}))(p)$ is the set of terms in context $(p|\vec{n})$ modulo the equations. The functorial action of $T(\vec{n})$ corresponds to the left-hand substitution structure in (6).
2. The data $\eta_{\vec{n}}$ provides an element of the set $\hat{\mathbb{S}}(\vec{n}, P)(0)$, which identifies the variables among the terms.
3. The right-hand substitution structure in (6) gives natural transformations $T\vec{m} \times \hat{\mathbb{S}}(J\vec{m}, T\vec{n}) \to T\vec{n}$. By currying, this supplies the data $*_{\vec{m},\vec{n}}$.

The three commuting diagrams are easy substitution lemmas.

Theorem 2. *Every enriched clone for $\mathcal{V} = \mathcal{C} = \hat{\mathbb{S}}$ and $\mathcal{A} = \mathbf{Ctx}(\mathbb{S})$ arises from a presentation of a parameterized algebraic theory.*

In general, when J is dense, enriched clones can be understood as monoids in the multicategory whose objects are $\mathcal{V}$-functors $\mathcal{A} \to \mathcal{C}$, and where an n-ary morphism $F_1, \ldots, F_n \to G$ between $\mathcal{V}$-functors is an extra-natural family of morphisms $\mathcal{C}(JA_0, F_1A_1) \otimes \cdots \otimes \mathcal{C}(JA_{n-1}, F_nA_n) \to \mathcal{C}(JA_0, GA_n)$ in $\mathcal{V}$. When this multicategory has tensors then we arrive at the situation considered by Kelly and Power [25, §5]. They focus on the situation where $\mathcal{A}$ comprises the finitely presentable objects of $\mathcal{C}$, but it seems reasonable to replace 'finitely presentable' with another well-behaved notion of finiteness [29, 2, 10, 42]. We consider sifted colimits [2, 3, 29], i.e. colimits that commute with products in the category of sets, which leads us to the notions of strongly finitely presentable object and strongly accessible category [2] (aka generalized variety [3]).

Proposition 2 (c.f. [25], §5). *Let $\mathcal{V}$ be strongly finitely accessible as a closed category. Let $J\colon \mathcal{A}\subseteq\mathcal{V}$ comprise the strongly finitely presentable objects. Let $T\colon \mathcal{A}\to\mathcal{V}$ be a $\mathcal{V}$-functor. To equip T with the structure of an enriched clone is to equip the left Kan extension of T along J with the structure of an enriched monad.*

Parameterized algebraic theories fit the premises of this proposition. The presheaf category $\hat{\mathbb{S}}$ is strongly finitely accessible as a closed category, and $\mathbf{Ctx}(\mathbb{S})$ comprises the strongly finitely presentable objects (up to splitting idempotents).

Corollary 1. *To give a parameterized algebraic theory is to give a sifted-colimit-preserving enriched monad on $\hat{\mathbb{S}}$.*

Summary. We have shown that our framework for parameterized algebraic theories (§2) is a syntactic formalism for enriched clones (§5). For our theory of disjunctive predicate logic, which has applications to logic programming (§3), the clones can be represented abstractly as sieves (§4).

References

1. P. Aczel. A general Church-Rosser theorem. 1978.
2. J. Adámek, F. Borceux, S. Lack, and J. Rosický. A classification of accessible categories. *J. Pure Appl. Algebra*, 175(1–3):7–30, 2002.
3. J. Adámek and J. Rosický. On sifted colimits and generalized varieties. *Theory Appl. Categ.*, 8(3):33–53, 2001.
4. R. Adams. Lambda-free logical frameworks. *Ann. Pure Appl. Logic*. To appear.
5. T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. In *FOSSACS'10*, pages 297–311, 2010.
6. G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, 410(46):4626–4671, 2009.
7. S. Antoy and M. Hanus. Functional logic programming. *C. ACM*, 53(4):74–85, 2010.
8. A. Asperti and S. Martini. Projections instead of variables: A category theoretic interpretation of logic programs. In *Proc. ICLP'89*, 1989.
9. A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. arXiv:1203.1539v1.
10. C. Berger, P.-A. Melliès, and M. Weber. Monads with arities and their associated theories. *J. Pure Appl. Algebra*, 216(8–9):2029–2048, 2012.
11. B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming functional logic programs into monadic functional programs. In *Proc. WFLP 2010*. Springer, 2011.
12. F. Bronsard and U. S. Reddy. Axiomatization of a functional logic language. In *Proc. ALP'90*, 1990.
13. R. A. Clouston and A. M. Pitts. Nominal equational logic. In *Computation, Meaning, and Logic*. Elsevier, 2007.
14. P. M. Cohn. *Universal algebra*. D Reidel, 2nd edition, 1981.
15. P.-L. Curien. Operads, clones and distributive laws. In *Operads and Universal Algebra*. World Scientific, 2012.
16. S. E. Finkelstein, P. J. Freyd, and J. Lipton. Logic programming in tau categories. In *CSL'94*.
17. M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proc. LICS'99*, 1999.
18. M. P. Fiore and C.-K. Hur. Second-order equational logic. In *Proc. CSL'10*, 2010.
19. M. P. Fiore and O. Mahmoud. Second-order algebraic theories. In *Proc. MFCS 2010*.
20. M. J. Gabbay and A. Mathijssen. One and a halfth order logic. *J. Logic Comput.*, 18, 2008.
21. R. Jagadeesan, P. Panangaden, and K. Pingali. A fully abstract semantics for a functional language with logic variables. In *LICS'89*, 1989.
22. P. Johann, A. Simpson, and J. Voigtländer. A generic operational metatheory for algebraic effects. In *LICS 2010*, 2010.
23. P. T. Johnstone. *Sketches of an Elephant*. OUP, 2002.
24. O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *Proc. POPL'12*, 2012.
25. G. M. Kelly and A. J. Power. Adjunctions whose counits are coequalisers. *J. Pure Appl. Algebra*, 89:163–179, 1993.
26. Y. Kinoshita and A. J. Power. A fibrational semantics for logic programs. In *Proc. ELP'96*.
27. E. Komendantskaya and J. Power. Coalgebraic semantics for derivations in logic programming. In *Proc. CALCO*, 2011.
28. A. Kurz and D. Petrişan. Presenting functors on many-sorted varieties and applications. *Inform. Comput.*, 208(12):1421–1446, 2010.
29. S. Lack and J. Rosický. Notions of Lawvere theory. *Appl. Categ. Structures*, 19(1), 2011.
30. P.-A. Melliès. Segal condition meets computational effects. In *Proc. LICS 2010*, 2010.
31. R. E. Møgelberg and S. Staton. Linearly-used state in models of call-by-value. In *Proc. CALCO 2011*, 2011.
32. E. Moggi. Notions of computation and monads. *Inform. Comput.*, 93(1), 1991.
33. J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates. *J. Log. Program.*, 12(3&4):191–223, 1992.
34. G. Plotkin. Some varieties of equational logic. In *Algebra, meaning and computation*. Springer, 2006.
35. G. D. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FOSSACS'02*.
36. G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Categ. Structures*, 11(1):69–94, 2003.
37. G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *Proc. ESOP'09*, 2009.
38. U. S. Reddy. Functional logic languages, part I. In *Graph Reduction*, pages 401–425, 1986.
39. V. A. Saraswat, M. C. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. POPL'91*, pages 333–352, 1991.
40. T. Schrijvers, P. J. Stuckey, and P. Wadler. Monadic constraint programming. *J. Funct. Program.*, 19(6), 2009.
41. S. Staton. Relating coalgebraic notions of bisimulation. In *Proc. CALCO*, 2009.
42. J. Velebil and A. Kurz. Equational presentations of functors and monads. *Math. Struct. in Comp. Science*, 21, 2011.

A Implementation in Standard ML

This appendix is included to aid the referee. We elaborate on Section 3 by providing the source code of our simple implementation of the parameterized algebraic theory of disjunctive predicate logic in Standard ML.

In the body of this paper we focused on two parameterizing signatures: the empty signature, and the signature of natural numbers. In our implementation we consider a general signature with a function symbol (cons str) of every arity and for each string str. We demonstrate our implementation with the simple arithmetic examples from Section 3 and also a simple example of proof search in natural deduction.

Full source code is at www.cl.cam.ac.uk/~ss368/flp.

A.1 Signature

```
infix 3 =:=

signature DISJPREDLOGIC = sig
   (* SIGNATURE OF PARAMETERS *)
   (* param is the domain of discourse *)
   type param
   (* cons: takes a label and a list of arguments *)
   val cons : string * param list → param

   (* MAIN SIGNATURE *)
   val choose : unit → bool
   val fail : unit → 'a
   val free : unit → param
   val =:= : param * param → unit

   (* Auxilliary constructs, for testing *)
   exception Cancel
   exception Rollback
   val toString : param → string
end
```

A.2 Structure

```
structure DisjPredLogic :> DISJPREDLOGIC = struct
   open SMLofNJ.Cont
   exception Cancel
   exception Rollback

   (* A param is either something built from a constructor or a free variable
      (a pointer to NONE) or a pointer to another param (a pointer to SOME).
      The second argument of Var is for pretty printing. *)
   datatype param = Cons of string * param list
                  | Var of param option ref * int
```

```
  val cons = Cons

  fun toString ...

  (* equals : param * param → bool
     equals (a,b) returns true if a and b are known to be equal, false otherwise.
     equals will traverse chains of pointers. NB it is local to the structure. *)
  fun equals ...

  local val counter_ref = ref 0 in
     fun counter () = (counter_ref := !counter_ref + 1 ; !counter_ref)
  end

  (* NB capture/escape is like callcc/throw
     except the exception context is not captured. *)
  fun choose () =
      capture (fn k ⇒ ((escape k true) handle Rollback ⇒ escape k false))

  fun fail() = raise Rollback ;

  fun free() = Var ((ref NONE),counter())

  (* occurs i t checks whether i occurs in param t *)
  fun occurs i (Cons(s,es)) = app (occurs i) es
    | occurs i (Var (j,l)) = if i = j then fail()
                              else case !j of NONE ⇒ () | (SOME a) ⇒ occurs i a

  (* implementation of unification *)
  fun (Cons(s,es)) =:= (Cons(s',es')) =
        if s=s' then ListPair.appEq (op=:=) (es,es')
                    handle UnequalLengths ⇒ fail()
        else fail()
    | (Var (i,l)) =:= b =
        if equals (Var (i,l),b) then ()
        else
        (case !i of NONE ⇒ (occurs i b ; i := SOME b ;
           capture (fn k ⇒ ((escape k ())
                          handle Rollback ⇒ (i := NONE ; raise Rollback)
                               | Cancel ⇒ (i := NONE ; raise Cancel))))
          | SOME a ⇒ a =:= b)
    | a =:= (Var (j,l)) = (Var (j,l)) =:= a
end
```

A.3 Demonstration of arithmetic

```
(* Simple definitions *)
val zero = cons (''zero'',[])
fun succ x = cons (''succ'',[x])
fun fromInt n = if n=0 then zero else succ (fromInt (n-1))
```

```
(* Demonstrations from Sec 3 *)

fun add(a,b) = if choose() then (a =:= zero ; b)
             else let val a' = free() val c' = free()
                  in a =:= succ a' ; add(a',succ(b)) end

fun add'(a,b,c) = if choose() then (a =:= zero ; b =:= c)
                else let val a' = free() val c' = free()
                     in a =:= succ a' ; c =:= succ c' ; add'(a',b,c') end

fun iso f a = let val b = free() in f(a,b) end
fun isoinv g (a,b) = let val (b',c) = g(a) in b =:= b' ; c end
```

A.4 Demonstration of proof search in natural deduction

Here is a slightly more elaborate demonstration.

```
(* impl : param * param → param *)
fun impl (a,b) = cons (''⇒'',[a,b])

(* member : (param list) → param → unit *)
fun member [] p = fail ()
  | member (p :: ps) q = if choose () then p =:= q else member ps q

(* (pick n) returns a number in [1,n] *)
fun pick n = if n = 0 then fail () else if choose () then 1 else 1 + pick (n-1)

(* ded : (param list) * param → unit
   ded(ps,p) succeeds if p follows from ps
   There are three cases: axiom; impl introduction and impl elimination *)
fun ded(ps,p) = case pick 3 of
    1 ⇒ member ps p
  | 2 ⇒ let val q = free () val r = free () in
        p =:= (impl (q,r)) ; ded ((q :: ps),r) end
  | 3 ⇒ let val q = free () in
        ded (ps,(impl (q,p))) ; ded (ps,q) end
```

A.5 Sample toplevel transcript.

The following function is useful in testing.

```
(* get_all_interactive : ('a → 'b) → 'a → param list → unit
   (get_all_interactive f x vars) will run (f x)
   and print a possible instantiation for each of the params in vars
   press return to get next instantiation, press n to cancel *)
fun get_all_interactive ...
```

Here is a transcript from the SML toplevel.

```
$ sml flp.sml
Standard ML of New Jersey v110.72 [built: ...
...
- toString(add(fromInt 2,fromInt 1));
val it = "succ(succ(succ(zero)))" : string
- let val a=free() in add'(fromInt 2,fromInt 1,a) ; add'(a,a,fromInt 6) end;
val it = () : unit
- let val a=free() in add'(fromInt 2,fromInt 1,a) ; add'(a,a,fromInt 5) end;
uncaught exception Rollback...
- val a = free ();
val a = - : param
- val b = free ();
val b = - : param
- get_all_interactive add' (a,b,fromInt 4) [a,b];
zero,succ(succ(succ(succ(zero))))
succ(zero),succ(succ(succ(zero)))
succ(succ(zero)),succ(succ(zero))
succ(succ(succ(zero))),succ(zero)
succ(succ(succ(succ(zero)))),zero
Exhausted.
val it = () : unit
- get_all_interactive add' (fromInt 4,a,fromInt 6) [a];
succ(succ(zero))
Exhausted.
val it = () : unit
- get_all_interactive add'(a,a,fromInt 4) [a];
succ(succ(zero))
Exhausted.
val it = () : unit

- val p = cons("p",[]) val q = cons("q",[]) val r = cons("r",[]);
val p = - : param
val q = - : param
val r = - : param
- ded([],(impl (p,p)));
val it = () : unit
- ded([],(impl (p,q)));
^C
Interrupt
- ded([impl(p,q),impl(q,r)],(impl (p,r)));
val it = () : unit
- get_all_interactive ded ([a],(impl(p,q))) [a];
=>(p,q)
q
=>(p,q)
=>(=>(var279384,var279384),q)
=>(=>(var279383,p),q)
n
Cancelled.
```