Types as Theories

Joseph A. Goguen*
Programming Research Group, Oxford University
SRI International, Menlo Park CA 94025

12 September 1990

Abstract: There are many notions of type in computing. The most classical notion is "types as sets", which has been extended to cover many features of modern programming languages. This paper shows that such features are handled perhaps even more naturally by an extension of the "types as algebras" notion to a "types as theories" notion. This notion naturally supports object oriented concepts, including inheritance and local state, as well as generic modules and dependent types. Moreover, it explains why polymorphic operations are natural transformations and provides a systematic foundation for a powerful form of programming in the large. This paper also presents a new semantics for dependent types based on strict fibrations.

NOTE: This version of the paper contains some material that was inadvertently omitted from the version printed in Topology and Category Theory in Computer Science *(Oxford, 1991).*

---

1 Introduction

"Types" are used to classify programming entities. Because the entities involved in modern programming have many different kinds of structure and may be seen at many different levels of abstraction, it is not surprising that there are many different notions of type. For each notion of type, one should ask what it serves to classify. The two major established traditions are the "types as sets" tradition, and the "types as algebras" tradition; both these notions of type classify values. The first leads from classical programming languages to the λ-calculus and so-called "type theory," while the second leads from abstract data types into object oriented concepts and programming in the large. Brief introductions to these two traditions are given in the following two subsections, with some indication of how the second extends to the "types as theories" in which types classify models rather than values, and in particular, classify software modules and objects.

A basic distinction concerns the time at which types are checked against what they classify:

- Types that are checked at **compile time** are called "static types". One problem with static typing is that it can be too strong; for example, the expression "$(-24/-8)!$" would fail a strong static type check under the assumption that factorial (i.e., "!") is only defined on natural numbers, because computation is needed to determine whether the division yields a natural number. The polymorphic types of ML [48] can be checked at compile time.

*The research in this paper was supported in part by grants from the Science and Engineering Research Council, the National Science Foundation, and the System Development Foundation, plus contracts with the Office of Naval Research Contracts and the Fujitsu Corporation.

- Types checked at **run time** are called "dynamic types". Phenomena in this area include coercions (as in many conventional programming languages), retracts (as in OBJ [9, 28]), and some forms of polymorphism.

- Types checked at **design time** do not seem to have an established popular name: this paper calls them "theories". We believe that this kind of type is the most relevant to formal methods for program development, and to programming in the large.

This paper assumes familiarity with category, functor, natural transformation and colimit, for which see texts and papers such as [29], [38] and [4]; Lawvere theories and indexed categories are also mentioned but not assumed. Section 2.5 assumes adjoints, and some other advanced concepts are defined as they arise. The identity morphism at A is denoted 1_A, and composition is indicated by semicolon, so that for example, $(f;g)(x) = g(f(x))$.

This paper uses OBJ [9, 28] as a notation for examples. Much of this notation is hopefully self-evident because it is based directly on algebra, and the rest is explained where it arises. Indeed, this paper provides semantics for much of OBJ.

1.1 Types as Sets

The oldest notion of type may be summarised by the slogan "types as sets." Types in this sense classify values. [6] surveys this tradition (but despite its title, does not survey other approaches to types).

This notion is embodied in the most traditional programming languages, such as FORTRAN and BASIC, which have few types and no operations for combining types. More advanced languages may enrich this approach with some simple operations on sets, such as disjoint union and Cartesian product ($\times$ and $\uplus$, for records and variant records), and possibly inclusion ($\subseteq$, for a somewhat restricted notion of subtype). Many more or less modern programming languages embody this approach, including Algol68, Pascal and Modula-2. More recent work tends to take a higher order viewpoint, treating functions as values to be classified; ML and functional languages like Miranda embody this approach.

Theoretical work in this tradition, such as domain theory and Scott's "data types as lattices" [54, 55], tends to impose a partial ordering $\sqsubseteq$ on the entities in a type. Higher order functions arise through an exponentiation operation on types. A variant models types by "ideals" in domains [43]. A major benefit of this work is its support for the λ-calculus in programming, as anticipated by Landin [36, 37] and McCarthy [45]. An important recent advance is the second order polymorphic calculus, independently discovered by Reynolds [52] and Girard [10].

In the "types as predicates" variant of the "types as sets" approach, types are taken to be predicates, which therefore denote sets (or some variant thereof, such as domains). However, many advocates of this view are more proof theoretically inclined, and hence might resist such denotations. Perhaps the best known work along this line is Martin-Löf's "type theory" [44], which also provides dependent types, as implemented in Pebble [5] and other languages. (Note that "type theory" is not a general theory of types, but rather a specific intuitionistic logic which provides one specific notion of type).

Although much effort has been put into the "types as sets" tradition, perhaps because it seems easier to understand, it nonetheless has some deficiencies: it does not take sufficient account of the operations associated with types, and in particular, selectors (for constructors) are not treated in a natural way (see [57] for a treatment of some special cases, and [22] for a general discussion of the problem); it does not deal with the correctness of implementations; it has a difficult logic, so that it is hard to prove properties of types and operations;

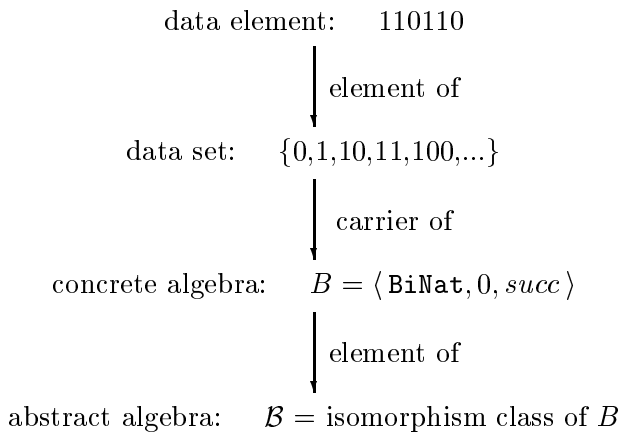


Figure 1: Levels of Abstraction of Data

and it is difficult to apply to modules, exceptions, states, object oriented concepts, and so on, often requiring complex *ad hoc* encodings to achieve such ends.

1.2 Types as Algebras

The essential insight of the "types as algebras" notion is that the *operations* associated with data are at least as important as the values. Thus, this approach generalises from sets to *algebras*, which are just sets with some given operations. Early work anticipating or advocating this approach includes lectures by Goguen at ETH Zürich [12], Hoare's work on data representation [34], and work by Zilles and Liskov [61, 42], ADJ [26, 25], and Guttag [31] on data abstraction.

The "initial algebra semantics" of ADJ says that a concrete data representation is a concrete (many sorted) algebra, and an abstract data type is an abstract algebra, where as usual in algebra, "abstract" means "unique up-to-isomorphism." A convenient way to describe (or "present") an abstract algebra is as "the" initial algebra in the variety of all algebras that satisfy some equations; that is, one gives a signature Σ (of sorts and operation symbols) and set E of (possibly conditional) Σ-equations, and then defines the class of the initial models of $\langle \Sigma, E \rangle$ to *be* the abstract data type. This works because any two such initial algebras are necessarily isomorphic. Such a pair $\langle \Sigma, E \rangle$ is called a *theory*, and serves to classify algebras by whether or not they satisfy it; thus theories are types.

For example, consider the natural numbers, where Σ has the one sort `Nat`, and the following two operations,

```
0 : -> Nat
succ : Nat -> Nat
```

There are no equations, so $E = \emptyset$. Now let B be the concrete initial Σ-algebra of natural numbers represented in binary positional notation. Finally, let $\mathcal{B}$ be the class of all initial Σ-algebras, i.e., of all Σ-algebras isomorphic to B. Thus, the intended denotation of the theory $\mathtt{NAT} = \langle \Sigma, \emptyset \rangle$ is $\mathcal{B}$. This way of characterising the natural numbers is due to Lawvere [41], and is shown equivalent to the Peano axioms in [39].

We can extend this approach in many different ways. Section 2 develops it into the "types as theories" approach, in which the theories themselves, and various operations

upon them, come to play a dominant role. Figure 1 gives an overview of the various levels of abstraction that are involved. From this, we can see that the three approaches to type have a natural hierarchical order: the "types as sets" approach is the most concrete, the "types as algebras" approach is a modest abstraction of it, and the "types as theories" approach supports genuine independence of representation, while still encompassing the other approaches.

Another natural step extends from many sorted algebra to order sorted algebra [24], in which a partial order is given on the set of sorts, interpreted semantically as subset inclusion. This permits a nice treatment of exception handling, and of partial and overloaded operations, as illustrated later in this paper. See also the "unified algebra" of Mosses [50]. Another extension admits relations as well as operations, leading to various forms of first order logic. Horn clause logic with equality is especially convenient, because it still has initial models for all theories, as shown in work on the semantics of the language Eqlog, which combines functional and logic programming [21, 20].

Yet another extension considers "machines" that may have internal states, identifying two such machines iff their behaviour is equivalent [19, 46]. This gives a semantics for abstract objects which generalises that of abstract data types, as in the semantics of FOOPS [23]. An early attempt to handle state appears in Guttag's thesis [31], later formalised by Wand [60] using final algebras. However, approaches which admit a class of models, e.g., those models that are behaviourally equivalent, seem more satisfactory. Section 3.2 gives another such approach, based on the satisfaction of equations up to observability.

The existence of applications using many different logical systems suggests generalising to an arbitrary logical system. This requires formalising the notion of logical system. Fortunately, such a formalisation is available in the form of *institutions* [18], which arose in the semantics of the specification language Clear [2, 3]. The framework of institutions is reviewed in Section 2.1, and then systematically used thereafter.

Acknowledgements

The approach in this paper evolved out of work on Clear [2, 3, 18] and its semantics; this joint research with Rod Burstall includes the concept of institution and the putting together of theories by colimits to form larger specifications. These ideas were further developed in joint work with José Meseguer on various topics, including OBJ [9, 28], which can be seen as an implementation of Clear for the order sorted (conditional) equational institution, as well as FOOPS [23], which extends these ideas to object oriented programming. Special thanks to Don Sannella and Andrzej Tarlecki for spotting some bugs in a draft of this paper. My debt to the Venerable Chögyam Trungpa Rinpoche is not expressible in words, and his inspiration is pervasive.

2 Types as Theories

This section first gives a brief introduction to institutions, and then uses them to explicate theories, parameterised (also called "dependent") theories, and parameterised theory application. Although these concepts are syntactic in nature, they have formal denotations using constructions on classes of models. Some novel ideas about fibrations and indexed categories arise in this connection, and many examples are given.

2.1 Institutions

The formal development in this section assumes a fixed but arbitrary logical system, having *signatures* denoted Σ, Σ', etc., having Σ-*models* denoted M, M', etc., having Σ-*sentences* denoted P, Q, etc., and having Σ-*satisfaction* relations, denoted

$M \models_\Sigma P$

and indicating that the Σ-sentence P is satisfied by the Σ-model M.

For many purposes, we also need signature morphisms, denoted $\phi\colon\ \Sigma \to \Sigma'$, and we need to know how these morphisms act on sentences and on models. Thus, if P is a Σ-sentence, we let $\phi(P)$ denote the *translation* of P by ϕ, a Σ'-sentence (generally obtained by substituting $\phi(\sigma)$ into P for each $\sigma \in \Sigma$), and if M' is a Σ'-model, we let $\phi(M')$ denote the *translation* (or *reduct*) of M' by ϕ, a Σ-model (generally obtained by letting $\phi(\sigma)$ be the interpretation in M' for each $\sigma \in \Sigma$).

Imposing some simple conditions on this data leads to the notion of an *institution* [18], details of which are omitted here. The main technical condition, called the **Satisfaction Condition**, expresses the invariance of satisfaction under change of notation,

$\phi(M') \models_\Sigma P$ iff $M' \models_{\Sigma'} \phi(P)$

where $\phi\colon\ \Sigma \to \Sigma'$ is a signature morphism, P is a Σ-sentence and M' is a Σ'-model. In addition, signatures and their morphisms form a category, denoted **Sign**; if $Sen(\Sigma)$ denotes the class of all Σ-sentences, then $Sen\colon\ \mathbf{Sign} \to \mathbf{SET}$ is a functor; and if $Mod(\Sigma)$ denotes the category of all Σ-models, then $Mod\colon\ \mathbf{Sign} \to \mathbf{CAT}^{op}$ is a functor. Example institutions include many sorted equational logic, Horn clause logic, order sorted equational logic, and first order logic, as shown in [18].

The illustrations in this paper mainly use many sorted equational logic. In this institution, a signature is a pair $\langle S, \Sigma\rangle$ where S is a *sort set* and Σ is an $S^* \times S$-indexed family of sets. A signature morphism $\phi\colon\ \Sigma \to \Sigma'$ consists of a function $f\colon\ S \to S'$ and an $S^* \times S$-indexed family of functions $g_{w,s}\colon\ \Sigma_{w,s} \to \Sigma'_{f(w),f(s)}$. Σ-models are Σ-algebras, Σ-sentences are (conditional) Σ-equations, and Σ-satisfaction is as usual. (Actually, the main examples will use order sorted equational logic [24].)

2.2 Theories

This subsection develops theories over an arbitrary institution, following the semantics of Clear [3, 18].

Definition 1 A **theory** consists of a signature Σ and a set E of Σ-sentences, i.e., it is a pair $\langle \Sigma, E\rangle$. We may also call $\langle \Sigma, E\rangle$ a Σ-theory.

Given a theory $T = \langle \Sigma, E\rangle$ and a Σ-model M, we say that M is a T-**model**, or that M **satisfies** T iff $M \models_\Sigma P$ for each $P \in E$, written $M \models_\Sigma E$.

A theory $\langle \Sigma, E\rangle$ is **closed** iff

$M \models_\Sigma E$ implies ($M \models_\Sigma P$ implies $P \in E$).

(Sometimes theories are called "presentations" to emphasise that closure is not required.) □

Thus, a Σ-theory classifies Σ-models by whether or not they satisfy it.

Definition 2 Given a set T of Σ-sentences and a class $\mathcal{M}$ of Σ-models, let

$T^* = \{M \mid M \models_\Sigma T\}$, the **variety** or **denotation** of T,

and let

$\mathcal{M}^* = \{e \mid M \models_\Sigma e, \text{ for all } M \in \mathcal{M}\}$, the **theory** of $\mathcal{M}$, also denoted $Th(\mathcal{M})$.

Because an institution provides the category $Mod(\Sigma)$ of all Σ-models, we can let T^* be a full subcategory of $Mod(\Sigma)$. $\Box$

Notice that satisfaction induces a duality (or more precisely, a Galois connection) between

- sets of sentences (i.e., theories), and
- sets (or classes, to be more precise) of models,

by the operations denoted * in the above definition. You might think that because of this duality we could just as well take model classes as basic, instead of theories. However, theories have the advantage over model classes that they are "more presentable": we can often find a *finite* set of (finite) sentences that describes what we are interested in, whereas the model classes (as well as their elements) are much less likely to be finite.

Fact 3 Given a theory $T = \langle \Sigma, E \rangle$, there is a least closed set $\overline{E}$ of Σ-sentences containing E, defined by

$P \in \overline{E}$ iff $M \models_\Sigma P$ for all M such that $M \models_\Sigma E$,

or equivalently, by

$\overline{E} = E^{**}$.

We will call $\overline{E}$ the **closure** of E, and $\langle \Sigma, \overline{E} \rangle$ the **closure** of T, written $\overline{T}$. $\Box$

Let us now consider some examples, using the syntax of OBJ. The simplest non-void theory is

```
th TRIV is
  sort Elt .
endth
```

Notice that TRIV* = **SET**, the category of all sets. Next, we consider monoids and groups, two simple theories from abstract algebra:

```
th MONOID is
  sort Elt .
  op e : -> Elt .
  op _·_ : Elt Elt -> Elt .
  vars A B C : Elt .
  eq A · e = A .
  eq e · A = A .
  eq A · (B · C) = (A · B) · C .
endth

th GROUP is
  sort Elt .
  op e : -> Elt .
  op _·_ : Elt Elt -> Elt .
  op _-1 : Elt -> Elt .
  vars A B C : Elt .
  eq A · e = A .
  eq e · A = A .
  eq A · (B · C) = (A · B) · C .
  eq A · (A -1) = e .
  eq (A -1) · A = e .
endth
```

These are theories over the institution $\mathcal{EQ}$ of (conditional, many sorted) equational logic. The denotation MONOID* of the theory MONOID of monoids is the variety (or category) **MONOID** of all monoids; similarly, GROUP* = **GROUP**.

2.3 Dependent Theories

In order to talk about relationships between theories – for example, inheritance, or that a certain theory serves as interface for a certain generic module – we need the following:

Definition 4 A **view** or **theory morphism** $\Phi\colon \langle \Sigma, E\rangle \to \langle \Sigma', E'\rangle$ is a signature morphism $\Phi\colon \Sigma \to \Sigma'$ such that

$P \in E$ implies $\Phi(P) \in \overline{E'}$.

This gives a category **Theo**$(\mathcal{I})$ of theories over an institution $\mathcal{I}$. □

Fact 5 $\Phi\colon \Sigma \to \Sigma'$ is a theory morphism $\Phi\colon \langle \Sigma, E\rangle \to \langle \Sigma', E'\rangle$ iff

$M' \models_{\Sigma'} E'$ implies $M' \models_{\Sigma'} \Phi(E)$

for all Σ'-models M', where $\Phi(E) = \{\Phi(P) \mid P \in E\}$. □

A **subtheory** is a view $\langle \Sigma, E\rangle \to \langle \Sigma', E'\rangle$, where $\Sigma \subseteq \Sigma'$ and $E \subseteq E'$; although this only makes sense for institutions where the notion of subsignature is well-defined, most cases of practical interest are included. For example, MONOID ⊆ GROUP.

Definition 6 The **denotation** of a theory morphism $\Phi\colon T \to T'$ is its **reduct** or **forgetful functor**, $\Phi^*\colon T'^* \to T^*$, defined to send M' to $\Phi(M')$, which is a T-model by Fact 5 plus the Satisfaction Condition, and to send $f\colon M'_0 \to M'_1$ to $\Phi(f)$. □

Because TRIV is a subtheory of every theory (that has at least one sort), we have

TRIV ⊆ GROUP

and

TRIV ⊆ MONOID,

which induce the familiar forgetful functors,

GROUP* $\longrightarrow$ **SET**

and

MONOID* $\longrightarrow$ **SET**.

For another example, the inclusion MONOID ⊆ GROUP induces the forgetful functor

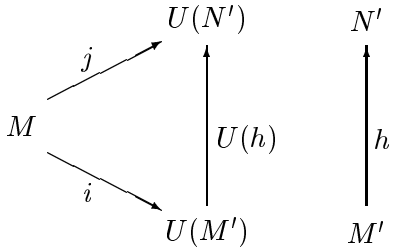
GROUP* $\longrightarrow$ MONOID*.

Many institutions are "liberal" in the sense that all their forgetful functors have "left adjoints" which give the "free things generated by". It is possible to define this without a lot of category theory:

Definition 7 Given a theory morphism $\Phi\colon T \to T'$, let

$U = \Phi^*\colon T'^* \to T^*$

be its forgetful functor, and let M be a model in T^*. Then a model M' in T'^* is **free** (with respect to U) **over** M iff there is some morphism $i\colon M \to U(M')$ such that given any $j\colon M \to U(N')$, there is a unique morphism $h\colon M' \to N'$ such that $i;U(h) = j$, i.e., such that



commutes, where i, j are Σ-morphisms and h is a Σ'-morphism. We may write $j^\#$ for h, because it is uniquely determined by j.

An institution is **liberal** iff for every such theory morphism Φ and model M, there is some model M' that is free over M with respect to Φ^*. □

Intuitively, M' is the best T'^*-model that "extends" M. Examples include the following:

- The free group on a set of X of generators.
- The free monoid on a set of X of generators.
- The free group over a monoid M.

Liberal institutions include equational logic (in all the variations that we have mentioned), Horn clause logic, Horn clause logic with equality, and many others – but not first order logic.

Proposition 8 Given a theory morphism $\Phi\colon T \to T'$, the function $M \mapsto M'$ for $M \in T^*$ where M' is free (with respect to Φ^*) over M, extends uniquely to a functor, denoted

$$\Phi^\$\colon T^* \to T'^*$$

and called a **free functor**. It is left adjoint to the forgetful functor Φ^*. Such functors always exist for liberal institutions. □

For the "types as theories" point of view to include the "types as models" point of view, it is convenient to introduce a new kind of sentence which says that an *initial* model is intended for a theory (or more generally, for part of a theory). The theory of institutions provides such sentences, called "constraints" [18]; see also [51]. In order to define these, we first formalise the idea that a model is free over part of a theory. Instead of just considering a theory inclusion, we generalise to an arbitrary theory morphism.

Definition 9 Given a theory morphism $\Phi\colon T \to T'$, we say that a T'-model M' is Φ-**free** iff M' is free over $\Phi^*(M')$ and the morphism $(1_{\Phi^*(M')})^\#\colon \Phi^\$(\Phi^*(M')) \longrightarrow M'$, called the **counit** morphism at M' and denoted $\epsilon_{M'}$, is an isomorphism. □

We now define the new kind of sentence:

Definition 10 Given a signature Σ, then a Σ-**constraint** is a pair

$$\langle \Phi\colon T'' \to T',\ \theta\colon Sign(T') \to \Sigma \rangle,$$

where T', T'' are theories and $Sign(T')$ is the signature of T'. Given a Σ-constraint $c = \langle \Phi\colon T'' \to T',\ \theta\colon Sign(T') \to \Sigma \rangle$ and a Σ-model M, we say that M **satisfies** c iff $\theta(M)$ satisfies T' and is Φ-free. Given signature morphism $\phi\colon \Sigma \to \Sigma'$, then the ϕ-**translation** of $c = \langle \Phi,\ \theta \rangle$ is the constraint $\langle \Phi,\ \theta; \phi \rangle$, denoted $\phi(c)$. □

Proposition 11 Given an institution $\mathcal{I}$, a new institution denoted $\mathcal{C}(\mathcal{I})$ is obtained from $\mathcal{I}$ as follows: its signatures are the same as those of $\mathcal{I}$; its sentences include those of $\mathcal{I}$, plus constraints as new sentences, with translation and satisfaction as given in Definition 10. □

See [18] for more detail, including a proof. In $\mathcal{C}(\mathcal{I})$, a Σ-theory contains both Σ-sentences from $\mathcal{I}$ and Σ-constraints, and is satisfied by a model M iff M satisfies both all the sentences and all the constraints. It is worth noting that some $\mathcal{C}(\mathcal{I})$ theories may be satisfied by *no* models, even when $\mathcal{I}$ is liberal, because of the possibility of inconsistent constraints.

Let us now consider some examples, to show how these ideas are used in practice. Again we use OBJ as a notation. For the unparameterised case, the keyword "`th`" indicates that any model is acceptable, whereas the keyword "`obj`" indicates that the only acceptable models are Φ-free with respect to the theory inclusion $\Phi\colon\ \emptyset \to T$, where $\emptyset$ denotes the empty theory (with no sorts) and T is the body of the theory in question. In this way, unparameterised theories are a special case of parameterised theories. (Note that $\emptyset$ has just one model, having no carriers and no operations.) For example, the natural numbers may be specified as follows[1]:

```
obj NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat [prec 5].
  op _+_ : Nat Nat -> Nat .
  vars M N : Nat .
  eq 0 + N = N .
  eq s M + N = s(M + N).
endo
```

This theory denotes the class of all initial algebras over the theory $T = \langle \Sigma, E \rangle$ where Σ has $S = \{\texttt{Nat}\}$ and the three operation symbols `0`, `s_`, `_+_`, and where E contains the two equations given above, according to the following result:

Proposition 12 Given an $\mathcal{I}$-theory $T = \langle \Sigma, E \rangle$, a model M is an initial model of T iff it satisfies the $\mathcal{C}(\mathcal{I})$-theory $T' = \langle \Sigma, E' \rangle$, where E' is E plus the constraint $\langle \emptyset \to T, 1_\Sigma \rangle$. We may call constraints of this form **initiality constraints**. □

Our machinery also provides a natural semantics for parameterised theories, such as the following theory of actions of monoids on sets of states; intuitively, $A \cdot S$ is the new state that results from taking action A when in state S.

```
th ACT[M :: MONOID] is
  sort State .
  op _·_ : Elt State -> State .
  var A A' : Elt .
  var S : State .
  eq A · (A' · S) = (A · A') · S .
  eq e · S = S .
endth
```

We take the forgetful functor $\Phi^*\colon\ \texttt{ACT}^* \longrightarrow \texttt{MONOID}^*$ to be the denotation of this parameterised theory, where `ACT` includes not only the operations and equations shown above, but also those in `MONOID`.

In much the same way, we can give a denotation for a constrained parameterised theory. It is traditional to illustrate this with the following example, which also makes good use of order sorted algebra (for more information on order sorted algebra, see [24, 22]):

[1] The attribute `[prec 5]` of the prefix operation `s` indicates that it is more tightly binding than `+`, which OBJ3 gives the default precedence for binary infix operations of 41; see [28].

```
obj STACK[X :: TRIV] is
   sorts Stack NeStack .
   subsort NeStack < Stack .
   op push : Elt Stack -> NeStack .
   op empty : -> Stack .
   op pop_ : NeStack -> Stack .
   op top_ : NeStack -> Elt .
   var E : Elt .
   var S : Stack .
   eq pop push(E,S) = S .
   eq top push(E,S) = E .
endo
```

This denotes the $\mathcal{C}(\mathcal{EQ})$-theory with all the sorts, operations and equations stated above, plus `Elt` from `TRIV` and the constraint

$\langle \texttt{TRIV} \hookrightarrow \texttt{STACK}, 1_{Sign(\texttt{STACK})} \rangle$.

Letting $\Phi\colon Sign(\texttt{TRIV}) \to Sign(\texttt{STACK})$ be the signature inclusion and letting $\texttt{STACK}^* = \mathbf{STACK}$, then $\Phi^*\colon \mathbf{STACK} \longrightarrow \mathbf{SET}$ is the denotation for the parameterised theory `STACK`, whereas $\Phi^{\$}\colon \mathbf{SET} \longrightarrow \texttt{STACKEQ}^*$ would be its denotation in the style of Clear [3, 18], where `STACKEQ` is the purely equational part of the theory `STACK`.

The following example, which given integers a, b, c, returns the "type" of functions $f\colon \texttt{Int} \to \texttt{Int}$ which always lie within c of $ax + b$, is more similar to examples of dependent types in the usual literature on that subject:

```
th 3INT is
  pr INT .
  ops a b c : -> Int .
endth
```

(The line "`pr INT .`" indicates that the theory `INT` of integers is being imported, i.e., "inherited", in the sense explained in Section 3.1 below.)

```
th WLIN[P :: 3INT] is
  op f : Int -> Int .
  var X : Int .
  eq f(X) - c < (a * X) + b = true .
  eq (a * X) + b < f(X) + c = true .
endth
```

These examples motivate the following:

Definition 13 A **parameterised** (or **dependent**) **theory** is a theory morphism

$\Phi\colon T \to T'$.

The **denotation** of $\Phi\colon T \to T'$ is its forgetful functor $\Phi^*\colon T'^* \to T^*$, as in Definition 6. □

Such a parameterised theory corresponds to a "universal" or "product" dependent type, as illustrated by the above examples, provided that the institution involved has the form $\mathcal{C}(\mathcal{I})$; thus either or both T and T' may include constraints.

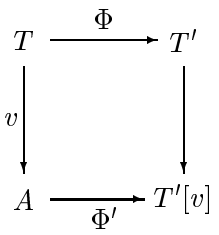


Figure 2: Application of a Parameterised Theory

2.4 Theory Instantiation

The specification language Clear provides a semantics for applying (also called "instantiating") parameterised theories. This semantics uses colimits, and was inspsired by some earlier ideas from General Systems Theory [11]:

> Given a parameterised theory $\Phi\colon T \to T'$ and a view $v\colon T \to A$ of an "actual" theory A as an "instance" of T, then the result of "applying" Φ to v is the pushout shown in Figure 2, whose pushout object is denoted $T'[v]$ and whose morphism opposite to Φ may be denoted Φ'. The denotation of $T'[v]$ is $(T'[v])^*$.

To know that we can always do this construction, we need the following results from [18]:

Theorem 14 If the category **Sign** of signatures of an institution $\mathcal{I}$ has all (finite) colimits, then so does the category **Theo**$(\mathcal{I})$ of all theories over $\mathcal{I}$. □

Corollary 15 If the category **Sign** of signatures of an institution $\mathcal{I}$ has all (finite) colimits, then so does the category **Theo**$(\mathcal{C}(\mathcal{I}))$ of all theories over $\mathcal{C}(\mathcal{I})$. □

All this generalises to the case where there are "shared" subtheories, but it is convenient to defer this topic to Section 3.1 below, where it is discussed under the heading of "inheritance".

Now let's do some simple theory instantiations, beginning with a view of the Booleans as a monoid, again using syntax from OBJ:

```
view BOOMON from MONOID to BOOL is
  sort Elt to Bool .
  op (_·_) to (_and_) .
  op e to true .
endv
```

(Actually, the sort line could be omitted, because of default view conventions that are built into OBJ [28].) Notice that there is no "preferred" view of the Booleans as a monoid; we could just as well have mapped "·" to disjunction, exclusive-or, or iff. Now we can apply `ACT` to `BOOL`, using the following syntax:

```
make BOOACT is ACT[BOOMON] .
```

which gives the theory `BOOACT` of actions of `BOOL` on sets, with conjunction as composition.

For another example,

```
make NATSTACK is STACK[NAT] .
```

gives the theory of stacks of natural numbers, exploiting OBJ's ability to infer the following default view:

```
view NATT from TRIV to NAT is
  sort Elt to Nat .
endv
```

Finally, let us instantiate `WLIN`, as follows:

```
view MY3INT from 3INT to INT is
  op a to 2 .
  op b to 3 .
  op c to 3 .
endv

make MYWLIN is WLIN[MY3LIN] .
```

which yields the following theory:

```
th MYWLIN is
  pr INT .
  op f : Int -> Int .
  var X : Int .
  eq f(X) - 3 < (2 * X) + 3 = true .
  eq (2 * X) + 3 < f(X) + 3 = true .
endth
```

2.5 Fibration and Indexed Category Semantics

Given a parameterised theory $\Phi\colon T \to T'$, notice that for each T^*-model M there is a full subcategory of T'^* whose objects are the models M' such that $\Phi^*(M') = M$. Examining this situation more closely reveals some interesting structure, first noticed in a more general form by Grothendieck [30] in the context of algebraic geometry.

Definition 16 A (**strict**) **fibration** is a functor $P\colon \mathbf{A} \to \mathbf{B}$ together with for each morphism $b\colon B \to B'$ in $\mathbf{B}$, a functor $b^{\spadesuit}\colon B'^{\spadesuit} \to B^{\spadesuit}$ such that $(b;b')^{\spadesuit} = b'^{\spadesuit}; b^{\spadesuit}$ and $1_B^{\spadesuit} = 1_{B^{\spadesuit}}$, where $B^{\spadesuit}$ denotes the full subcategory of $\mathbf{A}$ with objects A such that $P(A) = B$. We may also write $P^{\spadesuit}(B)$ for $B^{\spadesuit}$, the **fiber** over B, and $P^{\spadesuit}(b)$ for $b^{\spadesuit}$. We call $\mathbf{B}$ the **base** of the fibration. □

For example, if Φ is the theory inclusion `MONOID` $\longrightarrow$ `ACT`, then Φ^* is a strict fibration, where given a monoid homomorphism $h\colon M \to M'$, then $\Phi^{\spadesuit}(h)\colon \Phi^{\spadesuit}(M') \to \Phi^{\spadesuit}(M)$ sends an action $a\colon M' \times S \to S$ to the action $(h \times 1_S); a\colon M \times S \to S$ which sends $\langle m, s\rangle$ to $a(h(m), s)$. Here $\Phi^{\spadesuit}(M)$ is the category of all actions of M.

If $P\colon \mathbf{A} \to \mathbf{B}$ is a strict fibration, then $P^{\spadesuit}\colon \mathbf{B} \to \mathbf{CAT}^{op}$ is a strict indexed category (in the sense of [58]). Conversely, any strict indexed category gives rise to a strict fibration, by the strict Grothendieck construction described in [58].

The following concept (from [18]) is used in giving a sufficient condition for Φ^* to be a strict fibration:

Definition 17 A theory morphism $\Phi\colon T \to T'$ is **persistent** iff Φ^* has a left adjoint $\Phi^{\$}$ such that its unit η is an isomorphism, and is **strictly persistent** iff η is an identity. □

Thus when Φ is strictly persistent, $\Phi^*(\Phi^\$(M)) = M$ for each T-model M, and general results imply that if Φ is persistent, then it is also strictly persistent, by choosing a slightly different functor $\Phi^\$$. In a more geometrical language, we might say that such a $\Phi^\$$ is a *section* of Φ^*, in the sense that $\Phi^\$; \Phi^* = 1_{T^*}$.

Proposition 18 If $\Phi\colon T \to T'$ is strictly persistent and if T'^* has pullbacks, then Φ^* can be given the structure of a strict fibration.

Proof: Given $h\colon M \to N$ in T^* and N' in $\Phi^\spadesuit(N)$, so that $\Phi^*(N') = N$, then from $h\colon M \to \Phi^*(N')$ the adjunction gives us the morphism $h^\#_{N'}\colon \Phi^\$(M) \to N'$, and it also gives us the counit $\epsilon_{N'}\colon \Phi^\$(\Phi^*(N')) \to N'$. Hence we can form the following pullback diagram in T'^*

$$\begin{array}{ccc} P & \longrightarrow & \Phi^\$(\Phi^*(N')) \\ \downarrow & & \downarrow{\scriptstyle \epsilon_{N'}} \\ \Phi^\$(M) & \xrightarrow[h^\#_{N'}]{} & N' \end{array}$$

and using strong persistence plus the fact that Φ^* preserves limits gives the pullback diagram

$$\begin{array}{ccc} \Phi^*(P) & \longrightarrow & N \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle 1_N} \\ M & \xrightarrow[h]{} & N \end{array}$$

in T^*, from which we conclude that $g = 1_M$ and hence that P is in $\Phi^\spadesuit(M)$. We now define $h^\spadesuit(N') = P$.

Next, given $m\colon N' \to N''$ in $\Phi^\spadesuit(N)$, we get a similar pullback diagram which defines $h^\spadesuit(N'') = P'$,

$$\begin{array}{ccc} P' & \longrightarrow & \Phi^\$(\Phi^*(N'')) \\ \downarrow & & \downarrow{\scriptstyle \epsilon_{N''}} \\ \Phi^\$(M) & \xrightarrow[h^\#_{N''}]{} & N'' \end{array}$$

and then noting that m induces a morphism from the diagram defining P to that defing P', we get an induced morphism $h^\spadesuit(m)\colon h^\spadesuit(N') \to h^\spadesuit(N'')$. It is now straightforward to check that $h^\spadesuit$, also denoted $\Phi^\spadesuit(h)$, is a functor $\Phi^\spadesuit(N) \longrightarrow \Phi^\spadesuit(M)$, and indeed, that $\Phi^\spadesuit$ is a functor $T^* \longrightarrow \mathbf{CAT}^{op}$. $\square$

In such cases, we may take the fibration Φ^*, or equivalently the corresponding indexed category $\Phi^\spadesuit\colon T^* \longrightarrow \mathbf{CAT}^{op}$, as the denotation of the dependent theory Φ.

For example, given the $\mathcal{C}(\mathcal{EQ})$-theory morphism $\Phi\colon$ `TRIV` $\to$ `STACK`, the indexed category $\Phi^\spadesuit\colon$ **SET** $\longrightarrow$ $\mathbf{CAT}^{op}$ sends each set X to the isomorphism class of initial `STACK[X]`-algebras.

It is natural to ask how the pushout semantics of theory application relates to the indexed category semantics, which might suggest that $v; \Phi^{\spadesuit} : T^* \longrightarrow \mathbf{CAT}^{op}$ should be the denotation of the application $T'[v]$.

Theorem 19 Let $\Phi : T \to T'$, let M be a T-model, and let $\underline{M}$ be the theory inclusion $T \to \{M\}^*$ (note that $\{M\}^*$ is the theory of M). Then

$$\Phi^{\spadesuit}(M) = (T'[\underline{M}])^*$$

and

$$v^* ; \Phi^{\spadesuit} = \Phi'^{\spadesuit}.$$

□

3 Object Oriented Concepts

This section applies the "types as theories" viewpoint to object oriented specification and programming. Although algebra has already been applied to the object oriented paradigm in a variety of different ways (e.g., see [19, 46, 53, 23, 33, 8, 56]), the present approach is based on institutions. See [47] for a general discussion of object oriented programming.

3.1 Inheritance of Modules

Many basic issues concerning inheritance are orthogonal to issues concerning object and state, because they have to do with importing one module into another, i.e., with theory morphisms; thus, they make sense in any institution. This subsection is concerned with these issues; objects, classes, methods and so on are considered in the next subsection.

The keyword `protecting` (abbreviated "`pr`") indicates that a theory is imported in a way that preserves its denotation. For example, a model M of the theory

```
th POSET is
  pr BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  cq E1 < E3 = true if E1 < E2 and E2 < E3 .
endth
```

will have two sorts[2], `Bool` and `Elt`, such that $M_{\texttt{Bool}}$ is the Booleans (up to isomorphism). In more detail, let Σ denote the signature of `POSET`, which includes the signature of `BOOL` plus the operation `_<_`. Then `POSET` is the $\mathcal{C}(\mathcal{EQ})$-theory which includes the two equations shown above, plus all the equations in `BOOL`, plus the Σ-constraint $\langle \emptyset \to \texttt{BOOL}, Sign(\texttt{BOOL}) \to \Sigma \rangle$. Thus, a Σ-algebra M satisfies this theory iff it satisfies all the equations in `POSET`, including those in `BOOL`, plus the constraint $\langle \emptyset \to \texttt{BOOL}, Sign(\texttt{BOOL}) \to \Sigma \rangle$, which says that it contains an uncorrupted copy of the Booleans.

Now let's consider an example with the keyword "`obj`" instead of "`th`", a module for stacks of natural numbers:

[2]Because OBJ assumes that `BOOL` is inherited into every module, the line "`pr BOOL .`" could be omitted, and future examples will do so. Also, note that the condition in a conditional equation is required to have sort `Bool` in OBJ, so that it is unnecessary to write "`= true`" at the end of the second equation of `POSET`.

```
obj STACK is
  pr NAT .
  sorts Stack NeStack .
  subsort NeStack < Stack .
  op push : Nat Stack -> Nestack .
  op empty : -> Stack .
  op pop_ : NeStack -> Stack .
  op top_ : NeStack -> Nat .
  var S : Stack .
  var N : Nat .
  eq pop push(N,S) = S .
  eq top push(N,S) = N .
endo
```

This denotes the $\mathcal{C}(\mathcal{EQ})$-theory containing all the equations stated above, plus those in NAT and the two constraints

$$\langle \texttt{NAT} \hookrightarrow \texttt{STACK},\ 1_{Sign(\texttt{STACK})} \rangle$$

and

$$\langle \emptyset \to \texttt{NAT},\ Sign(\texttt{NAT}) \to Sign(\texttt{STACK}) \rangle.$$

In general, the import declarations of a theory provide a base over which it is extended, freely if its keyword is "obj" (by adding a constraint over this base); also, all the imported constraints have their signature extended to that of the importing theory.

OBJ has two other forms of import declaration, extending and using (plus including in a more recent release), whose semantics can be handled in a similar way, using the more general forms of constraint discussed in [18]; in particular, the semantics of extending uses the form of hierarchy constraint developed in [18].

Often a theory is imported into two (or more) other theories which are later combined, for example through instantiating a parameterised theory, and then we would expect to see only one copy of this shared subtheory in result theory. In particular, BOOL is imported into every theory in OBJ but there should not be more than one copy of BOOL in (for example) STACK[NAT].

The desired effect is obtained in a natural way by letting each specification determine a *diagram* in the category of theories, called its *environment*, and defining the result theories to be the *colimits* of appropriate subdiagrams. Such environment diagrams in general have just one copy of fundamental theories such as BOOL, NAT and INT, shared among many other theories, while there may be many instances of useful dependent theories, such as LIST and SET. The extended example in Section 5 presents several environment diagrams.

Thus, just pushouts are not quite enough to give the intended semantics for modules with sharing. General colimits are also needed for evaluating general *module expressions*, which may call for more complex ways of combining modules that just the instantiation of parameterised modules; one important additional operation is the *sum* of theories, denoted "+", which is computed by coproducts; see also [3] and [18] for further discussion.

The above considerations extend to what DeBruijn [7] has called "telescopes", which are sequences

$$T_1 \xrightarrow{\Phi_1} T_2 \xrightarrow{\Phi_2} \ldots \xrightarrow{\Phi_n} T_{n+1}$$

where each Φ_i is a theory morphism. These represent types with multiple nested dependencies. There are many interesting relationships among the various denotations of the parts of a telescope, such as $(\Phi_1; \ldots; \Phi_n)^*$ and $(\Phi_1; \ldots; \Phi_n)^{\spadesuit}$. For example, Figure 3 shows how to instantiate a telescope of length 3: first instantiate T_2 with $v : T_1 \to A$, obtaining the object

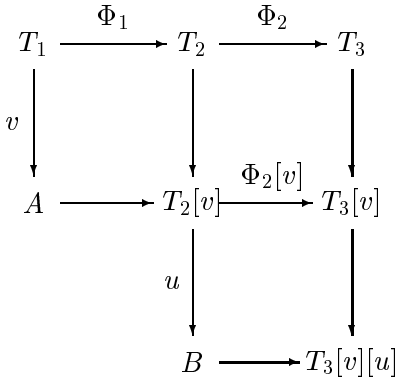


Figure 3: A Telescope Diagram

$T_2[v]$, and then instantiate T_3 with $u\colon T_2[v] \to B$. Alternatively, after instantiating with u, one could complete the top by taking a second pushout, yielding the object $T_3[v]$, and then do the bottom pushout. Well known general results about pushouts guarantee that either route leads the the same result (up to isomorphism), denoted $T_3[v][u]$ in Figure 3.

3.1.1 A Comparison

It is interesting to compare this notion of inheritance at the level of *theories* with the notion of inheritance at the level of *sorts* which is given by order sorted algebra [24]. Perhaps it is best to start by comparing the notions of type that each involves: the first is, of course, the "types as theories" notion which is the focus of this paper; the second lifts the inclusion notion of inheritance from the "types as sets" to the "types as algebras" level. Whereas inheritance for theories has to do with the reuse of code and the foundations of programming methodology, including programming in the large, inheritance by subsorts has to do with the hierarchical classification of values, whereby some value classes are contained in others. The level of theories has to do with the classification of implementations, in the sense of models defined by concrete code; here, inheritance has as its denotation a map from one model class to another, which is a forgetful functor arising from a theory inclusion. Note that many sorts of data may be involved simultaneously in a single relationship of theory inheritance.

3.2 Objects

This subsection presents an institution which provides an algebraic foundation for the object oriented paradigm, namely the **hidden sorted** (conditional) **equational institution**, in which the set S of sorts of a signature has explicitly given subsets $V, H \subseteq S$ of **visible** and **hidden sorts**, respectively. The elements with hidden sorts represent the internal states of objects, while the elements with visible sorts represent data values.

In order to bring out the main ideas more clearly, it is convenient to consider the special case in which the data values are given by an algebra A which is fixed once and for all. Thus, we assume as given a fixed set V of visible sorts, a fixed V-sorted signature Ψ, and a fixed Ψ-algebra A. For example, V might include sorts for booleans, natural numbers and identifiers, plus all the usual operations.

Then a **hidden sorted signature** Σ has $\Psi \subseteq \Sigma$ and $V \subseteq S$, and satisfying the following conditions, where $H = S - V$:

(S1) for each $c \in A_v$ there is some[3] $\psi \in \Psi_{[],v}$ such that ψ is interpreted as c in A;

(S2) if $\sigma \in \Sigma_{w,s}$ with $w \in V^*$ and $s \in V$, then $\sigma \in \Psi_{w,s}$; and

(S3) if $\sigma \in \Sigma_{w,s}$ then at most one element of w lies in H.

The first condition says that there are names for all data items, and the second expresses data encapsulatation, in the sense that no new operations are allowed on data. The third condition says that methods and attributes[4] act on single objects.

A **hidden sorted signature morphism**, $\Phi\colon \Sigma \to \Sigma'$, is a signature morphism $\Phi = \langle f, g\rangle\colon \Sigma \to \Sigma'$ such that

(M1) $f(v) = v$ for $v \in V$,

(M2) $g(\psi) = \psi$ for $\psi \in \Psi$,

(M3) $f(H) \subseteq H'$ (where $H' = S' - V$ and S' is the sort set of Σ'), and

(M4) if $\sigma' \in \Sigma'_{w',s'}$ and some sort in w' lies in $f(H)$, then $\sigma' = g(\sigma)$ for some $\sigma \in \Sigma$.

The first three conditions say that hidden sorted signature morphisms preserve both visibility and invisibility for both sorts and operations, while the fourth condition expresses the encapsulation of classes, in the sense that no new methods or attributes can be defined on an imported class. It is not difficult to check that hidden sorted signatures and their morphisms form a category. Notice that this definfition of hidden sorted signature morphsism allows new methods and attributes to be defined for new classes.

A **hidden sorted Σ-model** is a Σ-algebra M such that[5] $M|_\Psi = A$, and a **hidden sorted Σ-morphism** $h\colon M \to M'$ is a Σ-homomorphism such that $h|_\Psi = 1_A$, while a **hidden sorted Σ-sentence** is a (conditional) Σ-equation. Thus, we can let the Φ-translations of Σ'-models and Σ-sentences be their Φ-translations in the ordinary equational institution. However, **satisfaction** is rather different from the ordinary equational institution: We define

$$M \models^A_\Sigma (\forall X)\ t_0 = t_1$$

iff for all $v \in V$ and all $t \in T_{\Sigma \cup X \cup \{z\},v}$ having just one occurrence of z, where z is a new variable having the same sort as t_0 and t_1, we have

$$M \models_\Sigma (\forall X)\ t(z \leftarrow t_0) = t(z \leftarrow t_1).$$

Let us call this condition **behavioural satisfaction**, and let us call equations that are intended to be satisfied in this way **behavioural equations**. Also, let us call terms t of the form described above **contexts** for the equation. (The generalisation to conditional equations is straightforward.) The main result is the following:

Proposition 20 Hidden sorted equational signatures and morphisms, with ordinary (conditional) equations as sentences, and with models, translation, and satisfaction as defined above, form an institution.

Proof: Given $\Phi\colon \Sigma \to \Sigma'$, given a Σ'-algebra M', and given a Σ-sentence $e = (\forall X)\ t_0 = t_1$, we show that

$$\Phi(M') \models^A_\Sigma e \ \text{ iff } \ M' \models^A_{\Sigma'} \Phi(e),$$

i.e., that

[3] Here "[]" denotes the empty list of sorts.

[4] If $w \in S^*$ contains a hidden sort, then $\sigma \in \Sigma_{w,s}$ is called a **method** if $s \in H$ and an **attribute** if $s \in V$.

[5] Here $M|_\Psi$ denotes the restriction, or reduct, of M to Ψ, which gives a Ψ-algebra.

$\Phi(M') \models_\Sigma (\forall X)\ t(z \leftarrow t_0) = t(z \leftarrow t_1)$

iff

$M' \models_{\Sigma'} (\forall X')\ t'(z' \leftarrow \Phi(t_0)) = t'(z' \leftarrow \Phi(t_1))$,

where[6] $X' = \Phi(X)$. The "if" direction is follows easily from the Satisfaction Condition of the ordinary equational institution. For the converse, the definition of hidden sorted signature and morphism imply that each context t' for $\Phi(e)$ is of the form $t''(\Phi(t))$, where $t'' \in T_{\Psi \cup \{z'\}}$ and t is a context for e. Thus, we are reduced to showing that

$M' \models_{\Sigma'} (\forall X')\ \Phi(t)(z' \leftarrow \Phi(t_0)) = \Phi(t)(z' \leftarrow \Phi(t_1))$,

which follows directly from the Satisfaction Condition of the ordinary equational institution and the assumption. □

It is exciting (at least to the author) that the conditions (S1-3) and (M1-4), which express encapsulation in the object oriented paradigm, can be (and in fact, were) determined by the need to make the above proof of the Satisfaction Condition work. This derivation of the methodological principle of encapsulation from the metamathematical principle of the invariance of truth under change of notation (which is the intuitive content of the Satisfaction Condition) seems to confirm the naturalness of such principles.

It is often convenient to extend the hidden sorted equational institution to ordered sorts, Horn clauses and/or continuity. This presents no difficulty, and in fact, the following example uses the order sorted extension:

```
th STACK is
  pr NAT .
  hsorts Stack NeStack .
  hsubsort NeStack < Stack .
  op push : Nat Stack -> NeStack .
  op empty : -> Stack .
  op pop_ : NeStack -> Stack .
  op top_ : NeStack -> Nat .
  var S : Stack .
  var N : Nat .
  eq pop push(N,S) = S .
  eq top push(N,S) = N .
endth
```

Here `hsorts` and `hsubsort` are used for the hidden sorts. In our (somewhat simplified) hidden sorted equational institution, the line "`pr NAT .`" does not indicate a module inportation, but rather indicates that the algebra A of data items is some initial model of `NAT`; thus, Ψ is the signature of `NAT`. This style of specification uses a functional notation which requires an explicit state variable, contrary to the usual convention of imperative programming. However, we can regard imperative notation as syntactic sugar for the more explicit notation used above, and thus have our sweet syntactic cake and eat its healthy semantics too.

The models of `STACK` need only "appear" to satisfy its equations when observed through visible-valued terms, which necessarily have `top` as their head operation. For example, the usual implementation of a stack by a pointer and an array does not actually satisfy the first equation above, although it does satisfy it behaviourally, and hence it does satisfy `STACK`. See [14] for further discussion of this example.

[6] This notation is intended to suggest that X' has the same number of variables as X, and that their sorts are obtained by translation under Φ.

This theory specifies the stack class (in the sense of object oriented programming) by a set of axioms. These axioms can be used to reason about stack objects. If we let `STACKV` denote the above `STACK` theory with all of its sorts made visible, then its initial algebra also satisfies `STACK`, and we can reason about it in all the ways that are familiar from abstract data type theory, including induction, and be sure that any equation we prove is behaviourally satisfied by all models of `STACK`, because all these models are behaviourally equivalent.

The notion of encapsulation described above is somewhat stronger than might be desired. In particular, it might seem reasonable to allow new attributes and methods on old classes, provided that they are expressible as combinations of old attributes and methods. For example, if $\Phi : \Sigma \to \Sigma'$ is a signature inclusion morphism, where Σ has a class `PThing` with attributes `weight` and `volume`, then Σ' might introduce a new attribute `density` defined by

```
density(P) = weight(P) / volume(P),
```

assuming that volumes are non-zero and that division is available in Ψ. Also, the hidden sorted equational institution defined above does not support user defined data types or generic modules at either the data or the object level. It is not very difficult to generalise in these directions, by including specifications in signatures instead of a fixed algebra A, and by imposing a suitable condition on signature morphisms; but the details are omited here.

Although the concurrent interaction of objects is not addressed by this approach, techniques such as process algebra [49, 35] and sheaf theory [17] should be useful for this purpose, and the creation and destruction of objects can be handled using technqiues from order sorted algebra. There is also some relevant work by the ISCORE group, including [8] and [56]. Hidden sorts were introduced in [19], but behavioural satisfaction appears to be new.

4 Polymorphism is Natural

Following Milner [48], polymorphism has become a hallmark of higher order functional programming languages, such as ML [32], Miranda [59] and Orwell [1]. For this kind of polymorphism to work correctly, all the types involved must be "trivial", so that any type constructor can be applied to any other. For example, *list list* α, *stack list* α and $pair(\alpha, list\ \beta)$ are all polymorphic types, where α and β are type variables. In the language of this paper, we would say that the parameterised theories which describe these sorts must all have `TRIV` as their interface theory. It is also necessary to have a designated *principal* exported sort[7]. Thus, such a polymorphic type is a trivially parameterised theory with a designated sort, and it determines a functor $\mathbf{SET} \longrightarrow \mathbf{SET}$, because

$$\texttt{TRIV} \xrightarrow{\Phi} T \xleftarrow{\Psi} \texttt{TRIV}$$

gives rise to

$$\mathbf{SET} \xrightarrow{\Phi^\$} T^* \xrightarrow{\Psi^*} \mathbf{SET}$$

and these two functors can be composed to yield the desired denotation. More generally,

$$\texttt{TRIV}^n \xrightarrow{\Phi} T \xleftarrow{\Psi} \texttt{TRIV}$$

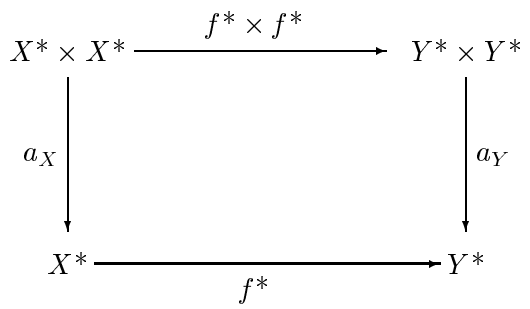
gives

$$\mathbf{SET}^n \xrightarrow{\Phi^\$} T^* \longrightarrow \mathbf{SET}.$$

An observation which has aroused interest in the functional programming community, and has been proved there in some special cases, is that polymorphic operations are natural

[7] It is natural to take the principal sort to be the one that is first mentioned in a module, as in OBJ [28].

transformations. For example, the append function on lists is natural in the sense that, given any function $f\colon X \to Y$ (in **SET**), the following diagram commutes,



where X^*, Y^* are lists of X, Y, and where a_X, a_Y are their append operations.

It is easy to prove this fact in general, making use of Lawvere's view [40] that an algebraic theory is a category with operations as its morphisms, an algebra is a product-preserving functor (to **SET**), and a homomorphism is a natural transformation. (Lawvere's inisght that homomorphisms are natural transformations is at first surprising, but is actually fairly simple and rather useful; for example, see [27].)

Now given a trivially parameterised type

$$\texttt{TRIV} \xrightarrow{\Phi} T$$

we get a functor

$$\mathbf{SET} \xrightarrow{\Phi^\$} [\mathrm{T} \xrightarrow{\pi} \mathbf{SET}]$$

where T is T as a Lawvere theory, and $[_ \xrightarrow{\pi} _]$ indicates the category of product-preserving functors with natural transformations as morphisms. Viewing this as a category with *graph* morphisms as objects and with natural transformations as their morphisms, let us pick out an edge, say $\sigma\colon s_1 \ldots s_n \longrightarrow s$ representing $\sigma \in \Sigma_{s_1 \ldots s_n, s}$. Then by restriction, we get the functor

$$\mathbf{SET} \xrightarrow{\Phi^\$} [(\cdot \xrightarrow{\sigma} \cdot) \longrightarrow \mathbf{SET}],$$

where "$\cdot \xrightarrow{\sigma} \cdot$" denotes the graph with two nodes and one edge, labelled "σ". But a functor $\mathbf{SET} \longrightarrow [(\cdot \to \cdot) \longrightarrow \mathbf{SET}]$ is *exactly* the same thing as a natural transformation between two functors $\mathbf{SET} \to \mathbf{SET}$, which is what we set out to find.

In the usual functional programming context, the "types" (i.e., sets X, Y, etc.) are defined recursively by a given set of type constructors, including "$\longrightarrow$" and *list* (i.e., *); but unfortunately "$\times$" is probably encoded into "$\longrightarrow$" by currying.

Similarly, polymorphism can be provided for languages like OBJ by regarding it as a notation for functions from implicitly imported instantiations of parameterised theories. Under this view, polymorphism is not limited to theories whose parameter interface is trivial, and moreover it embodies the second order capabilities (such as `map` functions) of parameterised modules in what would otherwise be a first order language, using the notation of conventional functional programming but with an underlying logic that is entirely first order, and hence is simpler to reason with. See [15] for further discussion. The same conventions can be used for object oriented programming, because of the developments in Section 3.2.

5 Programming in the Large

The polymorphism for trivial types discussed in the previous section is rather limited, and is not useful for programming in the large (i.e., for modules), where interface theories may

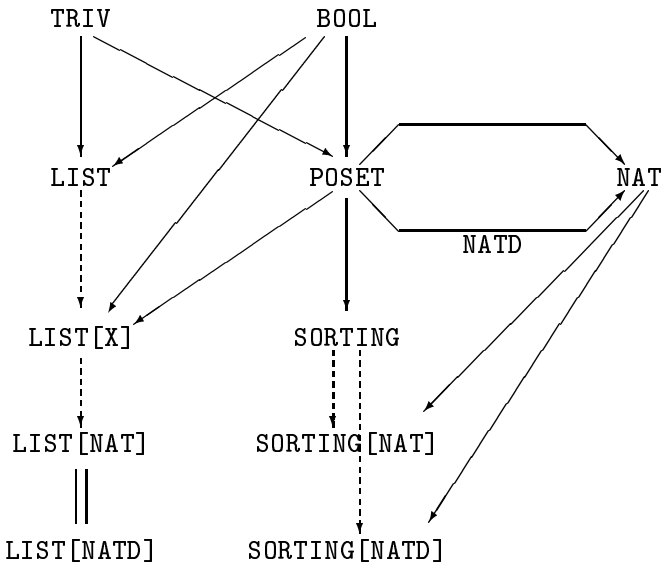


Figure 4: SORTING[NATD]

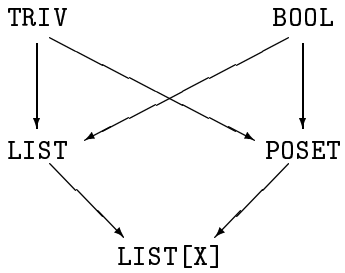


Figure 5: LIST[X]

involve many sorts and operations (as in Clear, Ada, OBJ, and Standard ML) and may also have laws (as in Clear and OBJ). We want a calculus for putting together: theories (specifications), code (compiled or source), documentation (explanations), animations, management and accounting information, test cases, correctness proofs, and so on. Let us call such a capability **hyperprogramming** [13, 16]. It provides an approach for integrating many diverse aspects of programming environments, and has been given a concrete syntax in the LIL system [13], originally intended for combining Ada modules.

Two key concepts in hyperprogramming are module expression and colimit. The first generalises the UNIX "`make`" command, which actually constructs a (sub)system when evaluated, while the second provides its semantics, as in Section 2.4.

First a series of declarations build up an "environment", which is a diagram in the category of theories of some institution; its denotation is its colimit. For example, the following code for `SORTING[NAT]` and `SORTING[NATD]` has the environment shown in Figure 4, in which the dotted lines indicate the relationship of "instantiation" for parameterised theories:

```
obj LIST[X :: TRIV] is
```

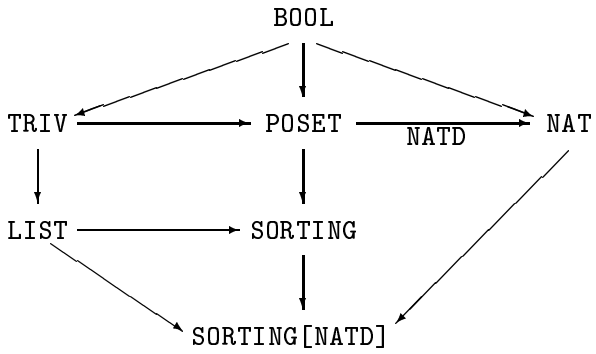


Figure 6: SORTING[NATD] Again

```
  sorts List NeList .
  subsorts Elt < NeList < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil prec 9]
  op head_ : NeList -> Elt .
  op tail_ : NeList -> List .
  var X : Elt .  var L : List .
  eq head(X L) = X .
  eq tail(X L) = L .
  op empty_ : List -> Bool .
  eq empty L = L =/= nil .
endo

obj SORTING[X :: POSET] is
  pr LIST[X] .
  op sorting_ : List -> List .
  op unsorted_ : List -> Bool .
  vars L L' L'' : List .  vars E E' : Elt .
  cq sorting L = L if unsorted L =/= true .
  cq sorting L E L' E' L'' = sorting L E' L' E L'' if E' < E .
  cq unsorted L E L' E' L'' = true if E' < E .
endo

make SORTING-NAT is SORTING[NAT].
reduce sorting 1 4 2 3 .

view NATD from POSET to NAT is
  vars L1 L2 : Elt .
  op L1 < L2 to L1 divides L2 and L1 =/= L2 .
endv

make SORTING-NATD is SORTING[NATD].
reduce sorting 1 4 2 3 .
```

The results of these two reductions are the lists `1 2 3 4` and `1 2 4 3`, respectively.

Figure 5 shows the structure of `LIST[X]` inside of `SORTING[X :: POSET]`. Figure 6 shows the diagram for the result of the second make, done all in one step.

6 Summary

We have explained a notion of "types as theories" and shown that it

- supports dependent types,
- supports a very general form of inheritance, based on theory morphisms,
- extends abstract data types to abstract objects, using an institution of hidden sorted equational logic,
- explains the naturality of polymorphic types,
- provides a foundation for programming in the large, and
- suggests a new way to unify programming environments.

In addition, we have suggested a new denotation for dependent theories based on strict Grothendieck fibrations and indexed categories.

It could be argued that the proper handling of abstraction is the most important problem in modern computer science, because it is the key to mastering the enormous complexity of modern software and software/hardware systems. It is hoped that the "types as theories" viewpoint can contribute to this effort by clarifying the concepts involved, because it captures several different forms of abstraction. The ideas in this paper have direct applications to the design of programming languages and environments; in particular, they have been used in designing OBJ, FOOPS and Eqlog, and have influenced the design of other languages.

References

[1] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[2] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[3] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland.

[4] Rod Burstall and Joseph Goguen. Algebras, theories and freeness: An introduction for computer scientists. In Martin Wirsing and Gunther Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 329–350. Reidel, 1982. Proceedings, 1981 Marktoberdorf NATO Summer School, NATO Advanced Study Institute Series, Volume C91.

[5] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In Giles Kahn, David MacQueen, and Gordon Plotkin, editors, *Proceedings, International Symposium on the Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 1984.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[7] Nicolas de Bruijn. Telescope mapping in typed lambda calculus. Technical Report P161-M17, Department of Mathematics and Computing Science, Eindhoven University of Technology, April 1989. Earlier versions from 1986 and 1987.

[8] Hans-Dieter Ehrich, Amilcar Sernadas, and Christina Sernadas. Objects, object types, and object identification. In Hartmut Ehrig et al., editors, *Categorical Methods in Computer Science with Aspects from Topology*, pages 142–156. Springer, 1989. Lecture Notes in Computer Science, Volume 393.

[9] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[10] Jean-Yves Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, Université Paris VII, 1972.

[11] Joseph Goguen. Mathematical representation of hierarchically organized systems. In E. Attinger, editor, *Global Systems Dynamics*, pages 112–128. S. Karger, 1971.

[12] Joseph Goguen. Some remarks on data structures. Abstract of 1973 Lectures at Eidgenoschiche Technische Hochschule, Zürich, 1973.

[13] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society, 1987, pages 251-263, and in *Domain Analysis and Software Systems Modelling*, Ruben Prieto-Diaz and Guillermo Arango, editors, IEEE Computer Society, 1991, pages 125-137.

[14] Joseph Goguen. An algebraic approach to refinement. In Dines Bjorner, C.A.R. Hoare, and Hans Langmaack, editors, *Proceedings, VDM'90: VDM and Z – Formal Methods in Software Development*, pages 12–28. Springer, 1990. Lecture Notes in Computer Science, Volume 428.

[15] Joseph Goguen. Higher-order functions considered unnecessary for higher-order programming. In David Turner, editor, *Research Topics in Functional Programming*, pages 309–352. Addison-Wesley, 1990. University of Texas at Austin Year of Programming Series; preliminary version in SRI Technical Report SRI-CSL-88-1, January 1988.

[16] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology.* Joint System Development Corporation, Tokyo, Japan, January 1990.

[17] Joseph Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, to appear 1991. Given as lecture at Engeler Festschrift, Zürich, 7 March 1989, and at U.K.-Japan Symposium on Concurrency, Oxford, September 1989; draft as Report CSLI-91-155, Center for the Study of Language and Information, Stanford University, June 1991.

[18] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992. Draft, as Report ECS-LFCS-90-106, Computer Science Department, University of Edinburgh, January 1990; an ancestor is "Introducing Institutions," in *Proceedings, Logics of Programming Workshop*, Edward Clarke and Dexter Kozen, editors, Springer Lecture Notes in Computer Science, Volume 164, pages 221-256, 1984.

[19] Joseph Goguen and José Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E.M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer, 1982. Lecture Notes in Computer Science, Volume 140.

[20] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179-210, September 1984.

[21] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22. Springer, 1987. Lecture Notes in Computer Science, Volume 250.

[22] Joseph Goguen and José Meseguer. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. In *Proceedings, Second Symposium on Logic in Computer Science*, pages 18–29. IEEE Computer Society, 1987. Also Report CSLI-87-92, Center for the Study of Language and Information, Stanford University, March 1987, and revised version to appear in *Information and Computation*.

[23] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.

[24] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989. Originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; many draft versions exist.

[25] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80-149.

[26] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Abstract data types as initial algebras and the correctness of data representations. In Alan Klinger, editor, *Computer Graphics, Pattern Recognition and Data Structure*, pages 89–93. IEEE, 1975.

[27] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. An introduction to categories, algebraic theories and algebras. Technical report, IBM Watson Research Center, Yorktown Heights NY, 1975. Report RC 5369.

[28] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988. Revised version to appear with additional authors José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud, in *Applications of Algebraic Specification using OBJ*, edited by Joseph Goguen, Derek Coleman and Robin Gallimore, Cambridge, 1992.

[29] Robert Goldblatt. *Topoi, the Categorial Analysis of Logic*. North-Holland, 1979.

[30] Alexandre Grothendieck. Catégories fibrées et descente. In *Revêtements étales et groupe fondamental, Séminaire de Géométrie Algébraique du Bois-Marie 1960/61, Exposé VI*. Institut des Hautes Études Scientifiques, 1963. Reprinted in Lecture Notes in Mathematics, Volume 224, Springer, 1971, pages 145–94.

[31] John Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto, 1975. Computer Science Department, Report CSRG–59.

[32] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[33] Fiona Hayes and Derek Coleman. Objects and inheritance: An algebraic view. Technical report, Hewlett-Packard Labs, Bristol, November 1989.

[34] C.A.R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[35] C.A.R. Hoare. Communicating sequential processes. *Communications of the Association for Computing Machinery*, 21:666–677, August 1978.

[36] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[37] Peter Landin. A correspondence between ALGOL60 and Church's lambda notation. *Communications of the Association for Computing Machinery*, 8(2):89–101, 1965.

[38] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[39] Saunders Mac Lane and Garrett Birkhoff. *Algebra*. Macmillan, 1967.

[40] F. William Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences, U.S.A.*, 50:869–872, 1963. Summary of Ph.D. Thesis, Columbia University.

[41] F. William Lawvere. An elementary theory of the category of sets. *Proceedings, National Academy of Sciences, U.S.A.*, 52:1506–1511, 1964.

[42] Barbara Liskov and Steve Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE-1(1):7–19, 1975.

[43] David MacQueen, Ravi Sethi, and Gordon Plotkin. An ideal model for recursive polymorphic types. In *Proceedings, Symposium on Principles of Programming Languages*, pages 165–174. Association for Computing Machinery, 1984.

[44] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holand, 1982.

[45] John McCarthy, Michael Levin, et al. *LISP 1.5 Programmer's Manual*. MIT, 1966.

[46] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[47] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[48] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[49] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980. Lecture Notes in Computer Science, Volume 92.

[50] Peter Mosses. Unified algebras and institutions. Technical Report DAIMI PB-274, Computer Science Department, Aarhus University, 1989.

[51] Horst Reichel. Initially restricting algebraic theories. In Piotr Dembinski, editor, *Mathematical Foundations of Computer Science*, pages 504–514. Springer, 1980. Lecture Notes in Computer Science, Volume 88.

[52] John Reynolds. Towards a theory of type structure. In *Colloquium sur la Programmation*, pages 408–423. Springer, 1974. Lecture Notes in Computer Science, Volume 19.

[53] Donald Sannella and Andrzej Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Science*, 34:150–178, 1987. Earlier version in *Proceedings, Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science, Volume 185, Springer, 1985.

[54] Dana Scott. Lattice theory, data types and semantics. In Randall Rustin, editor, *Formal Semantics of Algorithmic Languages*, pages 65–106. Prentice Hall, 1972.

[55] Dana Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–586, 1976.

[56] Amilcar Sernadas, José Fiadeiro, Christina Sernadas, and Hans-Dieter Ehrich. The basic building block of information systems. In Falkenberg and P. Lindgreen, editors, *Proceedings, IFIP 8.1 Working Conference on Information Systems Concepts*, pages 225–246. North Holland, 1989.

[57] Michael Smyth and Gordon Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computation*, 11:761–783, 1982. Also Report D.A.I. 60, University of Edinburgh, Department of Artificial Intelligence, December 1978.

[58] Andrzej Tarlecki, Rod Burstall, and Joseph Goguen. Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories. *Theoretical Computer Science*, 91:239–264, 1991. Also, Monograph PRG-77, August 1989, Programming Research Group, Oxford University.

[59] David Turner. Miranda: A non-strict functional language with polymorphic types. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architectures*, pages 1–16. Springer, 1985. Lecture Notes in Computer Science, Volume 201.

[60] Mitchell Wand. Final algebra semantics and data type extension. *Journal of Computer and System Sciences*, 19:27–44, 1979.

[61] Steven Zilles. Abstract specification of data types. Technical Report 119, Computation Structures Group, Massachusetts Institute of Technology, 1974.

Contents

1 Introduction **1**
1.1 Types as Sets 2
1.2 Types as Algebras 3

2 Types as Theories **4**
2.1 Institutions 5
2.2 Theories 5
2.3 Dependent Theories 7
2.4 Theory Instantiation 11
2.5 Fibration and Indexed Category Semantics 12

3 Object Oriented Concepts **14**
3.1 Inheritance of Modules 14
3.1.1 A Comparison 16
3.2 Objects 16

4 Polymorphism is Natural **19**

5 Programming in the Large **20**

6 Summary **23**