A Types-as-Sets Semantics for Milner-Style Polymorphism

Mitchell Wand

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405 USA

Abstract

In this paper we present a semantics for Milner-style polymorphism in which types are sets. The basic picture is that our programs are actually terms in a typed λ-calculus, in which the type information can be safely deleted from the concrete syntax. In order to allow for common programming constructs, we allow reflexive or infinite types, and we also allow opaque types, which have private representations.

An adaptation of Hindley's Principal Typing Theorem then asserts that the type information can be reconstructed. Thus expressions are polymorphic, since they may have more than one correct typing, but values are not. Expressions that are not well-typed are syntactically ill-formed, as they are in conventional mathematics, rather than having the meaning "wrong".

The resulting semantics is simpler than that for fully polymorphic models [Leivant 83], and generalizes the standard constructions, such as retracts and ideals.

1 Introduction

In conventional mathematical discourse, the intuitive notion of "type" seems much better founded than it is in computer science. In general, one can regard types as sets of objects, and a function may only be applied to an object from its domain. If one has a function f whose domain is the integers, and one attempts to apply it to a real number, say π, then one says that the expression $f(\pi)$ is meaningless because of an error in types. Thus we hold with Reynolds [83] that type structure is a *syntactic* discipline: terms which are not well-typed are considered to be ill-formed and therefore meaningless.

This situation is muddied in computer science because our machines always do *something* with every input, including our $f(\pi)$. One is then led to the notion of types as predicates on some universal domain: an integer is an object passing the integer predicate, and a function from integers to integers is an object which, when supplied with an object passing the integer predicate, produces another object passing the integer predicate. One then proves by induction that "well-typed" expressions produce "well-typed" answers, but expressions that are not "well-typed" may produce any answer at all. As Reynolds points out, these predicates inevitably overlap, leading to problems when an object may be both an integer and a function, etc. [Reynolds 83] attempts to deal model-theoretically with these issues.

In keeping with the spirit of [Wand 82, 83], we attempt to deal proof-theoretically with these issues. This is the traditional mathematical approach to abstraction and representation independence. Our starting point is the typed λ-calculus, which is the epitome of a well-understood theory of type. In order to invest this formulation with enough power to prove any useful theorems, we must supply it with some additional structure:

1. We need to allow the use of types other than functional types. We present the definition of a typed λ-calculus with such an expanded system of types. These types permit the use of arbitrary type constructors other than "→" and so-called "reflexive" types rather than just the finite types typically considered in the typed λ-calculus.
2. We then show how such a language accounts for programs written under Milner's type discipline, including the expanded notion of type. We prove an extension of the Principal Typing Theorem to show that it is decidable whether a term is typable, and that its principal type may be deduced by a simple extension to the usual unification algorithm.
3. We need to account for some primitive type constructors, such as products and sums, that do not seem to be definable in our general paradigm. We give a treatment of these; the treatment of sums, in particular, based on Reynolds, seems to be better than the ones usually used.
4. We then present our formalization of user-defined types. We model such types by including primitive constants which provide the isomorphism between a public type (such as, say, *Complex*) and its private representation (say (*pair Real Real*)). Such type constructors may also be parameterized to account for type abstraction, such as the type of (*list-of* α) for any type α.
5. We present a definition of model for our theories. To accomplish this, we show that Meyer's combinatory model theorem [Meyer 82] extends to any set of types which is closed under "→", including reflexive types.
6. This formulation, however, does not allow induction on approximate solutions, as do the standard limit constructions. We remedy this to some extent by using a suitably modified notion of the theory of Boehm-trees. Using the resulting theory, we prove some benchmark results.

Our formulations should be regarded as preliminary: we have included just the axioms needed to prove the desired theorems, subject only to the existence of reasonable models.

This Material is based on work supported by the National Science Foundation under grant number MCS79-04183.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0-89791-125-3/84/001/0158 $00.75

2 Language and Theories

Definition: A *type discipline* Δ is a set of trees (possibly including some infinite trees) closed under $\rightarrow$, that is, if $\alpha, \beta \in \Delta$ then $(\alpha \rightarrow \beta) \in \Delta$.

Examples: (1) Let Σ be a ranked set. Then the set of all finite trees built from "$\rightarrow$" and elements of Σ (i.e., the initial algebra generated by $\Sigma \cup \{\rightarrow\}$) is a type discipline.

(2) Same, but all finite and infinite trees with nodes labelled by $\Sigma \cup \{\rightarrow\}$.

(3) Same, but all finite and rational trees (i.e. only a finite number of elements of Σ appear, and the set of occurrences of each label is a regular set).

(4) The set consisting of a single tree: the complete infinite binary tree in which every node is labelled by "$\rightarrow$."

We omit parentheses and associate arrows to the right, as usual. We say a type discipline is *effective* iff there exists a set of finite representations for the elements of Δ such that the construction of a tree from its subtrees and the decomposition of a tree into its root and its subtrees are both recursive. Examples (1), (3), and (4) are effective. Henceforth, we assume whenever necessary that the type disciplines are effective.

For every type $\alpha \in \Delta$, we assume a countably infinite supply of variable symbols $x_1^\alpha, x_2^\alpha, \ldots$. We also allow a countable set C of constants with associated types. Generally such constants come in families, e.g. $cons_\alpha : \alpha \rightarrow (list\, \alpha) \rightarrow (list\, \alpha)$ for each type α.

We now formulate the notion of a typed λ-term. This definition is standard, except that it is relative to a type discipline Δ. In fact, it follows exactly the definition in [Barendregt 81, p. 560].

Definition: The language $\Delta(C)$ of Δ-typed λ-terms consists of a set of strings with associated types in Δ. We write $M : \alpha$ to indicate that string M has type α. The terms are defined as follows:

(1) If x^α is a variable of type α, then $x^\alpha : \alpha$ is a term.

(2) If $c : \alpha \in C$, then $c : \alpha$ is a term.

(3) If $M : \alpha \rightarrow \beta$ and $N : \alpha$ are terms, then $(MN) : \beta$ is a term.

(4) If x^α is a variable of type α, and $M : \beta$ is a term, then $(\lambda x^\alpha . M) : \alpha \rightarrow \beta$ is a term.

(5) Nothing else is a term.

The notion of a theory is also standard:

Definition: A *Δ-theory* is a set of pairs of $\Delta(C)$-terms closed under α-conversion, β-conversion, reflexivity, symmetry, transitivity, congruence (from $M = M'$ and $N = N'$ deduce $MN = M'N'$), and the ξ-rule (from $M = N$ deduce $\lambda x.M = \lambda x.N$).

3 Polymorphism

A Δ-theory constitutes a strongly-typed programming language, like PASCAL. A program is just a term in the language, and we compute by reducing the term to normal form. This is not, however, a particularly convenient language to program in, because one must put in too much type information: one needs a separate *mapcar* function for lists of every type (just as in PASCAL one needs separate routines for arrays of every size).

In practice, one does not need all these separate routines for two reasons, one pragmatic and one mathematical.

The pragmatic reason is that representations of related types share facets of their representation in a computer. Thus arrays of various sizes are all implemented as linear sequences of locations, and lists of booleans, integers, etc., are all typically implemented as linked lists of cells. Thus procedures such as *mapcar* can be polymorphic because they only manipulate the portion of the representation which is shared among the various instances.

In this analysis, polymorphism is a syntactic phenomenon: expressions are polymorphic because we are too lazy to put in all the subscripts on the combinators, and it so happens that we can get away with this because of standard implementation conventions. Values are not polymorphic.

This seems to be the analysis of polymorphism implicit in the discussion, though not in the theory, of [Milner 78]. It is to be contrasted with the full polymorphism of [Reynolds 74, Leivant 83] where values can be truly polymorphic. In Milner-style polymorphism, the type structure is weaker, but in compensation, it has a simpler semantics, as we will show. Furthermore, it seems adequate for a large number of applications, including a system for semantic prototyping [Wand 83a], whose construction motivated us to examine this question.

The mathematical reason is given by the Principal Typing Theorem of [Hindley 69], which states that given an arbitrary string M, it is decidable whether that string was obtained by removing the type superscripts from some term of the typed λ-calculus. If M was obtained by removing the type information from a term of type α, we say that α is a *possible type* of M. Furthermore, one can effectively find a unique "type scheme," called the *principal typing* of M, such that every possible type of M is an instance of that scheme. The proof utilizes the unification algorithm of [Robinson 65].

Hindley's proof allowed only finite types (not necessarily restricted to functional types, however). His proof, however, relies only on the decidability of the unification problem, so it goes through whenever the set of type schemes has a decidable unification problem. In particular, it goes through for the set of rational schemes. There the unification algorithm is just the conventional one, except that the so-called "occurrence check" is omitted [Robinson, personal communication].

Digression: In order to apply Hindley's proof, we need to take care of a few details. First, we need to extend the notion of a type discipline, as defined above, to a discipline of *type schemes*, which are like types except they may contain type variables. A discipline must be closed under $\rightarrow$ and under substitution of schemes for type variables. The types, as defined above, then become just the type schemes with no variables (i.e. the ground type schemes). Also, in place of the treatment of constants above, we associate with each constant symbol a principal type scheme, so rule (2) in the definition of terms becomes: "if $c : \eta \in C$, and τ is an instance of η, then $c_\tau : \tau$ is a term". Also, Hindley's proof is in terms of combinatory logic rather than the lambda-calculus, but Theorem 2 in his paper shows that this difference is inessential (see, for example, the statement of Hindley's theorem in [Barendregt 81, Proposition A.1.10]). **End of Digression**

Thus, given a term without type information, we can reconstruct the type information, and do so in the "most general" way. This allows us to program without giving types (though exactly where one should specify types anyway is a language design question). We can rely on the principal typing theorem to assure us that we will get answers of the right type, and on our shared representations to share object code.

Again, we see that expressions are polymorphic, but values are not. When we introduce a new constant, say *cons*, we actually are introducing an infinite family of typed constants $cons_\alpha : \alpha \rightarrow (list\alpha) \rightarrow (list\alpha)$ for each type α. Furthermore, $\alpha \rightarrow (list\alpha) \rightarrow (list\alpha)$ is a principal type scheme for *cons*.

What about generics? This is a troubling subject for conventional analyses. We account for generics, either local or global, by regarding them as merely syntactic sugar for their definition expressions. Thus something like "**let** $f = M$ **in** N" in ML

[Gordon *et. al.* 78] is syntactic sugar for $N[M/f]$, which has quite a different principal type from $(\lambda f.N)M$.

4 Products and Sums

In order to do any useful examples, we need some additional type structure beyond the pure typed λ-calculus. We will assume the following:

1. We have some ground types, among which are the type *Triv* with element *triv*.
2. We assume that for certain types α, there will be constants $d_{\alpha\beta} : \alpha \to \beta$ and $e_{\alpha\beta} : \beta \to \alpha$, at most one such pair per α, subject to the axioms:

$$d_{\alpha\beta}(e_{\alpha\beta}\, x) = x$$
$$e_{\alpha\beta}(d_{\alpha\beta}\, x) = x$$

These symbols are the key to the solution of domain equations, as discussed in the following section.

3. Our types are closed under the 2-place type constructors *pair* and *union*, and for all types α, β, γ, we have constants

$$pair_{\alpha\beta} : \alpha \to \beta \to pair\,\alpha\beta$$
$$lson_{\alpha\beta} : pair\,\alpha\beta \to \alpha$$
$$rson_{\alpha\beta} : pair\,\alpha\beta \to \beta$$

$$inL_{\alpha\beta} : \alpha \to union\,\alpha\beta$$
$$inR_{\alpha\beta} : \beta \to union\,\alpha\beta$$
$$case_{\alpha\beta\gamma} : union\,\alpha\beta \to (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$$

subject to the following axioms:

$$lson(pair\, x\, y) = x$$
$$rson(pair\, x\, y) = y$$
$$pair(lson\, x, rson\, x) = x$$
$$case(inL\, x) f g = f x$$
$$case(inR\, x) f g = g x$$

Our treatment of sums is motivated by the definition of a coproduct in a category, and was anticipated by Reynolds [75]. It allows expressions involving sum types to be well-typed, and avoids the introductions of canonical error elements, as required by the conventional treatment via "restrictions."

4. We need in addition to postulate that these constants interact nicely. An appropriate set of axioms seems to be the following:

$$a((case\, x f g)\, y) = (case\, x\, (a \circ f)(a \circ g))y$$
where a is *lson*, *rson*, d, or e (but not *pair*)
$$(case\,(case\, y\, i\, j)\, f\, g) = (case\, y\ \ \lambda u.case\,(iu) f g\ \ \lambda u.case\,(ju) f g)$$

5 User-defined types

In programming languages such as CLU or Simula, one can create "opaque" types, whose representation is known only in a small scope, though functions which manipulate that representation may be known in the rest of the program. This device allows a cleaner interface between types and their users.

To formalize this, let *Pub* be an opaque type with representation *Priv*. We add two constants to the typed λ-calculus: $d : Pub \to Priv$ (decode) and $e : Priv \to Pub$ (encode), subject to the axiom that these constants are two-sided inverses. We can then define the "public" operations on *Pub* in terms of d and e. This makes *Pub* a type that is isomorphic to *Priv* but distinct from it: one cannot manipulate an element of *Pub* except by using the decode and encode operations explicitly. Thus, the resulting calculus may be modelled by interpreting *Pub* as any set isomorphic to *Priv*, but not necessarily the same as *Priv*.

We are now in a position to do some examples. We introduce a bit of syntax:

```
deftype type rep type
    definitions
    end
```

This defines a new type in terms of its representation, and gives the definitions of the functions for manipulating it in terms of d and e. As a programming language construct, **deftype** ought to restrict the scope of d and e, but we will not do so, since that is an issue of language design, not of semantics.

```
deftype Bool rep union Triv Triv
  true = e(inL triv)
  false = e(inR triv)
  if_α : Bool → α → α → α = λ xyz.case (d x)(λu.y)(λu.z)
  end
```

This defines *Bool* as an opaque type, represented by the disjoint sum of two copies of *Triv*, with constants *true* and *false*, and a family of functions $if_\alpha : Bool \to \alpha \to \alpha \to \alpha$, one for each α. Thus, *if* (without the subscript) becomes a global generic symbol for the code above, so that one may have multiple occurrences of *if*, with different types, in a single scope. Note also that in keeping with our philosophy, we have suppressed the subscripts wherever possible.

The same machinery allows us to specify reflexive types. Here we introduce stream as a type constructor by simultaneously defining $(stream\,\alpha)$ for all α:

```
deftype (stream α) rep pair α (stream α)
  first_α : (stream α) → α
     = λs.lson(d s)
  rest_α : (stream α) → (stream α)
     = λs.rson(d s)
  cons-stream_α : α → (stream α) → (stream α)
     = λas.e(pair a s)
  end
```

As in the case of *Bool*, we have introduced a family of operators *first*, *rest*, and *cons-stream*, one for each type α. This is quite similar to the definition of an **absrectype** in ML [Gordon 78]. We can build streams without using a stream constant by using

fixed points, e.g.:

$$stream\text{-}of\text{-}false = fix(\lambda s.cons\text{-}stream\ false\ s)$$

which builds an infinite stream of *false*'s. (Note that since we have allowed rational types, *fix* is definable).

Now let us do something slightly more complicated: a data type of lists.

deftype $(list\ \alpha)$ **rep** *union Triv* $(pair\ \alpha\ (list\ \alpha))$
  $nil_\alpha : (list\ \alpha)$
    $= e(inL\ triv)$
  $cons_\alpha : \alpha \to (list\ \alpha) \to (list\ \alpha)$
    $= \lambda al.c(inR(pair\ al))$
  $list\text{-}case_{\alpha\beta} : (list\ \alpha) \to \beta \to (\alpha \to (list\ \alpha) \to \beta) \to \beta$
    $= \lambda laf.(case(d\ l)(\lambda u.a)(\lambda u.f(lson\ u)(rson\ u)))$
  end

Here we have homogeneous lists, with *nil* and *cons* as the usual list construction functions; the *inL* and *inR* serve to inject values into the appropriate summand of the representation. The list decomposition, however, is not the usual one: *list-case* takes a list and two more arguments. The first extra argument is returned if the list is empty; otherwise the second argument (f above) is applied to the first element and the remainder of the list. Like *if*, *list-case* has an additional degree of genericity. We have used this style of programming extensively. It turns out to be quite pleasant (and rather reminiscent of HOPE [Burstall, MacQueen, & Sannella 80]).

To illustrate this programming style, we can write the function *reduce* (see, *e.g.*, [Henderson 80, p. 41]) as follows:

$$reduce = \lambda f\,a\,l.fix(\lambda\theta.list\text{-}case\ l\ a\ (\lambda xy.f\,x(\theta y)))$$

As a second example, consider the case of stacks.

deftype $(stack\ \alpha)$ **rep** $(list\ \alpha)$
  $push_\alpha : \alpha \to (stack\ \alpha) \to (stack\ \alpha)$
    $= \lambda as.e(cons\ a\ s)$
  $pop_{\alpha\beta} : (stack\ \alpha) \to ((stack\ \alpha) \to \beta) \to \beta \to \beta$
    $= \lambda sf\,errvalue.\ list\text{-}case(d\ s)errvalue(\lambda al.f(el))$
  $top_{\alpha\beta} : (stack\ \alpha) \to (\alpha \to \beta) \to \beta \to \beta$
    $= \lambda sf\,errvalue.\ list\text{-}case(d\ s)errvalue(\lambda al.f\,a)$
  end

Again, *pop* and *top* have additional genericity: the type scheme of *top*, for example, is $(stack\ \alpha) \to (\alpha \to \beta) \to \beta \to \beta$. Hence, to take the top of a stack of α, one supplies the function $top_{\alpha\beta}$ with the stack, an error value of type β to be returned in case the stack is empty, and a function (of type $\alpha \to \beta$) to receive the top element in case the stack is non-empty. Such functions are ubiquitous in this programming style. They are analogous to Reynolds' acceptors [Reynolds 81], and provide a smooth, type-checkable treatment of error conditions.

6 Models

We now present the definition of a model for our theories. The definition is adapted from [Meyer 82].

Definition: A model $\mathcal{E}$ of $\Delta(C)$ consists of the following data:

a. for every $\alpha \in \Delta$, a set D_α.

b. for every $c : \alpha \in C$, an element $c^{\mathcal{E}} \in D_\alpha$.

c. for every $\alpha, \beta, \in \Delta$, a function $\cdot_{\alpha\beta} : D_{\alpha\to\beta} \times D_\alpha \to D_\beta$

d. for every $\alpha, \beta, \gamma \in \Delta$, elements

$$k_{\alpha\beta} \in D_{\alpha\to\beta\to\alpha}$$
$$s_{\alpha\beta\gamma} \in D_{(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}$$
$$e_{\alpha\beta} \in D_{(\alpha\to\beta)\to(\alpha\to\beta)}$$

such that

1. for all $x \in D_\alpha$, $y \in D_\beta$, $k_{\alpha\beta} \cdot x \cdot y = x$.
2. for all $x \in D_{\alpha\to\beta\to\gamma}$, $y \in D_{\alpha\to\beta}$, and $z \in D_\alpha$, $s_{\alpha\beta\gamma} \cdot x \cdot y \cdot z = x \cdot z \cdot (y \cdot z)$.
3. for all $x \in D_{\alpha\to\beta}$, $y \in D_\alpha$, $e_{\alpha\beta} \cdot x \cdot y = x \cdot y$.
4. for all $x, y \in D_{\alpha\to\beta}$, $((\forall z \in D_\alpha)(x \cdot z = y \cdot z) \supset (e_{\alpha\beta} \cdot x = e_{\alpha\beta} \cdot y))$

End of definition

Given a model $\mathcal{E}$, we can extend it to a valuation, which we will also call $\mathcal{E}$, on $\Delta(C)$-terms. To do this, let an environment ρ be any type-preserving function from variable symbols to $\bigcup D_\alpha$. Then the valuation is defined in the usual way, letting

$$\mathcal{E}[\![\lambda x^\alpha.M]\!]\rho = e_{\alpha\beta} \cdot d_{M\rho x}$$

when M is of type β and $d_{M\rho x}$ has the property that for all $d \in D_\alpha$, $d_{M\rho x} \cdot d = \mathcal{E}[\![M]\!]\rho[d/x]$. It is easy to show that the standard bracket-abstraction algorithm preserves types, from which it follows that the $d_{M\rho x}$ exist.

Given a model $\mathcal{E}$, the *theory* of $\mathcal{E}$, $Th(\mathcal{E})$ is defined as $\{M = N \mid \mathcal{E}[\![M]\!] = \mathcal{E}[\![N]\!]\}$.

It is easy to show that $Th(\mathcal{E})$ is always a Δ-theory. Furthermore, we can state

Theorem 1. *(Completeness Theorem). If T is a Δ-theory, then there is a model $\mathcal{E}$ such that $Th(\mathcal{E}) = T$.*

Proof: We construct the term model: choose $D_\alpha = \{[M]_T \mid M : \alpha\}$, that is, the set of T-equivalence classes of terms of type α. Let $[M] \cdot [N] = [(M\ N)]$, and let the s, k, e be the denotations (independent of environment) of appropriately typed versions of $\lambda xyz.xz(yz)$, $\lambda xy.x$, and $\lambda xy.xy$. Then $Th(\mathcal{E}) = T$; the proof, as sketched in [Meyer 82] goes through in the typed version as well. ∎

With these models, we complete the semantics for reflexive types in Milner's polymorphic system, in which expressions but not values are polymorphic. Terms in the system are terms in the typed λ-calculus, without the type subscripts, which can then be inserted by the principal typing algorithm. The fully typed terms can then be interpreted in the model.

It is easy to construct models of the sort we have described. Besides the term models constructed in the proof, the standard universal model constructions, with retracts, provide many such models [Scott 80]. The Semantic Soundness Theorem in [Milner 78] essentially shows that the ideals are a model of the finite functional types; this result has recently been extended to rational and other types by McQueen, Plotkin, and Sethi [83].

7 Representation Independence

Reynolds, Donahue, and others have discussed the notion of representation independence. The idea is that the result of a computation involving opaque types, such as stacks, should not depend on how they are actually represented, that is, upon the model chosen. Our picture provides a simple treatment of this idea. One cannot talk about the first component of a complex

number; one can merely talk about the first component of the Cartesian representation of a complex number, because typing provides the syntactic discipline. Whatever one can prove about the complex numbers will be true in any model, so it must be independent of the actual representation of the complexes. This provides a notion of representation independence that seems to be adequate for implementation; the results of [Donahue 79, Fokkinga 81, Haynes 82, *etc.*] remain of interest, however, as studies in model theory.

8 Restoring induction

Our treatment of opaque types deals with them as solutions to domain equations. Though our treatment is proof-theoretic, it of course relies on the work of Scott [72, 76], Lehmann and Smyth [81], and others who established the existence of such solutions by model-theoretic means, primarily using the notion of limits. (Lehmann and Smyth even pointed out that operations on data types could be defined in terms of the isomorphisms; this is implicit as well in the standard implementations, *e.g.* [Gordon 78], [Liskov *et. al.* 77].

How are solutions constructed by limits different from arbitrary solutions of domain equations? By analogy with the case of fixed points, one finds that limit solutions admit proofs by induction.

The natural proof-theoretic analog of an induction rule is the use of Boehm trees [Barendregt 81]. The Boehm tree of a λ-term is constructed, roughly, by taking the leftmost reduction until one gets a term of the form $\lambda x_1 \ldots x_n . y M_1 M_2 \ldots M_p$, and then proceeding similarly with the M_j; in general this gives an infinite tree. In our case, one needs to add the axioms of section 5 as reduction rules as well. One then adds all the equations of the form $M = N$ when $BT(M) = BT(N)$. This technique was used for compiler optimization in [Wand 83].

We conclude by showing some examples of theorems that can be proved by this principle. Consider the following definitions:

$$succs : int \rightarrow (stream\ int) \\ = fix(\lambda f n. cons\text{-}stream\ n (f (1+\ n)))$$

$$map : (\alpha \rightarrow \beta) \rightarrow ((stream\ \alpha) \rightarrow (stream\ \beta)) \\ = fix(\lambda \theta f s. \\ cons\text{-}stream\ (f (first\ s))(\theta f (rest\ s)))$$

$$ints = fix(\lambda s. cons\text{-}stream\ 0\ (map\ 1+\ \ s))$$

Here *succs* is a function which, given an integer *n*, produces the stream consisting of *n* followed by its successors in order. Thus (*succs*0) produces the stream 0, 1, 2, ... The function *map* is like *mapcar* for streams. Last, *ints* is the stream which begins with 0, and whose *rest* is obtained from *ints* by adding 1 to each element. Thus *ints* is also the stream 0, 1, 2, ...

Theorem 2. $ints = (succs\,0)$.

Proof: Both have Boehm tree $e(pair\,0\,(e(pair\,1 \ldots)))$. (See Figure 1). To conserve space, we have omitted the rather mechanical deduction. This example requires the use of the rules for *d* and *e* as reduction rules. ∎

Theorem 3. $map(f \circ g) = (map\ f) \circ (map\ g)$.

Proof: Both Boehm trees look like

$$\lambda s.e (pair (f (g (lson (d\ s))))(e (pair (f (g (lson (d (rson (d s)))))) \ldots)))$$

(See Figure 2) ∎

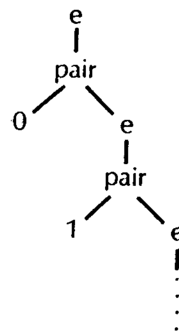

Figure 1. Boehm tree for $ints = (succs\,0)$.

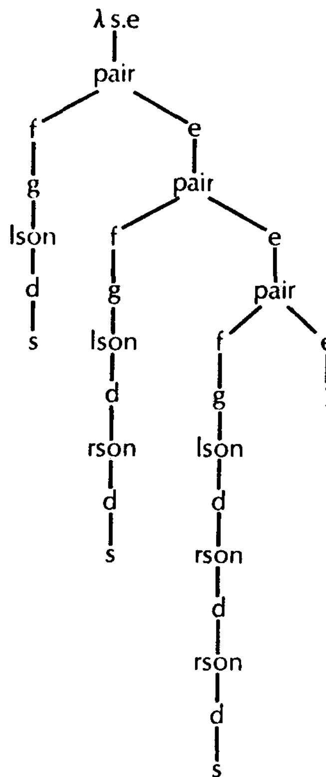

Figure 2. Boehm tree for $map(f \circ g) = (map\ f) \circ (map\ g)$.

A similar result can be shown for *mapcar*; in that case the use of the rewriting rules for *case* is necessary.

9 Open Problems

A variety of extensions and open problems suggest themselves:

- Our construction of term models extended that of Meyer to reflexive types. Leivant's construction of term models for Reynolds-style polymorphism [Leivant 83] also does not involve induction on type, and hence the same extensions should work to get an analog of our Theorem 1 for the logic of full polymorphism.
- If the set of types is closed under product as well as $\rightarrow$, then our models form Cartesian closed categories in the obvious way. Can one characterize the CCC's formed in this way?
- It may be posited that the theory of Boehm trees is the "real" theory that we are interested in, rather than simply the theory of β-conversion. Can we characterize the models of Boehm-tree theories in the same way that we now know how to characterize the models of lambda-theories?
- What are the Church-Rosser properties of these systems? Our system includes the surjective pairing axiom, *pair*(*lson x*, *rson x*) $= x$, although this rule was not used in any of the examples in Section 8. While this axiom causes the Church-Rosser theorem to fail for the untyped calculus, it is known that the pure typed λ-calculus with surjective pairing is Church-Rosser [Pottinger 81]. The situation for the typed λ-calculus with infinite types is unknown. Nothing is known about the Church-Rosser properties of typed λ-calculi with *case* operators, though in our examples we could interpret sums as products in the usual way. Another problematic axiom is the surjective sum axiom, *case x inL inR* $= x$.

10 Conclusions

We have presented a simple picture for Milner-style polymorphism. The picture accounts for type inference (polymorphism), type abstraction (opaque types) including reflexive types, and representation independence in a single framework.

We use proof theory to get abstraction, both as type abstraction and abstraction from representations; this is the traditional mathematical approach to abstraction. Our treatment of reflexive types is just a proof-theoretical counterpart to the model-theoretic approach of Scott, Plotkin, Lehmann and Smyth, *et. al.*; our work would be semantically vacuous without the hard-won knowledge that our theories had tractable models.

Acknowledgements

A preliminary version of this paper was presented at the Workshop on Types and Polymorphism in Programming Languages, at Carnegie-Mellon University. We thank Ravi Sethi, David MacQueen, and Daniel Leivant for the opportunity to present the paper on very short notice. They, Dana Scott, and Albert Meyer provided useful discussion.

References

[Barendregt 81]
Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.

[Burstall, MacQueen, & Sannella 80]
Burstall, R.M., MacQueen, D.B., and Sannella, D.T. "HOPE: An Experimental Applicative Language," *Conf. Rec. 1980 LISP Conference*, 136–143.

[Donahue 79]
Donahue, J. "On the Semantics of 'Data Type'," *SIAM J. Comput. 8* (1979), 546 560.

[Fokkinga 81]
Fokkinga, M.M. "On the Notion of Strong Typing," in *Algorithmic Languages* (deBakker and van Vliet, eds.), North-Holland, 1981, pp. 305–320.

[Gordon, *et. al* 78]
Gordon, M., Milner, R., Morris, L., Newey, M., and Wadsworth, C. "A Metalanguage for Interactive Proof in LCF," *Proc. 5th Annual ACM Symp. on Principles of Programming Languages* (1978) 119–130.

[Haynes 82]
Haynes, C. T. "A Theory of Data Type Representation Independence," University of Iowa Computer Science Department Technical Report Numer 82-04, December, 1982.

[Henderson 80]
Henderson, P., *Functional Programming: Application and Implementation*, Prentice-Hall International, Englewood Cliffs, NJ, 1980.

[Hindley 69]
Hindley, R. "The Principal Type-Scheme of an Object in Combinatory Logic," *Trans. Am. Math. Soc. 146* (1969) 29–60.

[Lehmann & Smyth 81]
Lehmann, D.J. and Smyth, M.B. "Algebraic Specification of Data Types: A Synthetic Approach," *Math. Sys. Th. 14* (1981), 97–139.

[Leivant 83]
Leivant, D. "Structural Semantics for Polymorphic Data Types (preliminary report)," *Conf. Rec. 10th ACM Symposium on Principles of Programming Languages* (1983), 155–166.

[Liskov *et. al* 77]
Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. "Abstraction Mechanisms in CLU," *Comm. ACM 20* (1977), 564–576.

[MacQueen, Plotkin, & Sethi 83]
MacQueen, D.B., Plotkin, G., and Sethi, R. "An Ideal Model for Recursive Polymorphic Types," presentation at Workshop on Data Types, Carnegie-Mellon University, June 9–10, 1983.

[Meyer 82]
Meyer, A.R. "What Is a Model of the Lambda Calculus," *Information and Control 52* (1982), 87-122.

[Milner 78]
Milner, R. "A Theory of Type Polymorphism in Programming," *J. Comp. & Sys. Sci. 17* (1978), 348–375.

[Pottinger 81]
Pottinger, G. "The Church-Rosser Theorem for the Typed λ-Calculus with Surjective Pairing," *Notre Dame Journal of Formal Logic 22* (1981) 264–268.

[Reynolds 74]
Reynolds, J.C. "Towards a Theory of Type Structures," in *Programming Symposium (Colloque sur la Programmation, Paris)* Springer Lecture Notes in Computer Science, Vol. 19, Berlin, 1974, pp. 408–425.

[Reynolds 75]
Reynolds, J.C. "User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction" *Conf. on New Directions on Algorithmic Languages,* IFIP WG 2.1, Munich, August, 1975.

[Reynolds 81]
Reynolds, J.C. "The Essence of Algol," in *Algorithmic Languages,* (J. W. deBakker and J.C. van Vliet, eds.) North-Holland, Amsterdam, 1981, pp. 345–372.

[Reynolds 83]
Reynolds, J.C. "Types, Abstractions, and Parametric Polymorphism," *Proc. IFIP 83.*

[Robinson 65]
Robinson, J.A. "A Machine-Oriented Logic Based on the Resolution Principle," *J. Assoc. Comput. Mach. 12* (1965), 23–41.

[Scott 72]
Scott, D. "Continuous Lattices" in *Toposes, Algebraic Geometry, and Logic* (F.W. Lawvere, ed.), Lecture Notes in Mathematics, vol. 274, Springer-Verlag, New York, pp. 97–136.

[Scott 76]
Scott, D. "Data Types as Lattices" *SIAM J. Comput. 5 (1976),* 522–587.

[Scott 80]
Scott, D. "Relating theories of the λ-calculus," in *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism* (Hindley and Seldin, eds.) Academic Press, New York and London, 1980, pp. 403–450.

[Wand 82]
Wand, M. "Semantics-Directed Machine Architecture" *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.* (1982), 234–241.

[Wand 83]
Wand, M. "Loops in Combinator-Based Compilers," *Conf. Rec. 10th ACM Symposium on Principles of Programming Languages* (1983), 190–196.

[Wand 83a]
Wand, M. "A Semantic Prototyping System," June, 1983.