MIT/LCS/TM-335

THE SEMANTICS OF MIRANDA'S ALGEBRAIC TYPES

Kim B. Bruce

Jon G. Riecke

August 1987

The Semantics of Miranda's Algebraic Types[*]

Kim B. Bruce and Jon G. Riecke[†]
Department of Computer Science
Williams College
Williamstown, MA 01267

July 29, 1987

Abstract

Miranda has two interesting features in its typing system: implicit polymorphism (also known as ML-style polymorphism) and algebraic types. Algebraic types create new types from old and can operate on arbitrary types. This paper argues that functions on types, or *type constructors*, best represent the meaning of algebraic types. Building upon this idea, we develop a denotational semantics for algebraic types. We first define a typed lambda calculus that specifies type constructors. A semantic model of type constructors is then built, using the ideal model as a basis. (The ideal model gives the most natural semantics for Miranda's implicit polymorphism.) The model is shown to be sound with respect to this lambda calculus. Finally, we demonstrate how to use the model to interpret algebraic types, and prove that the translation produces elements in the model.

Keywords: strong-typing, polymorphism, denotational semantics, Scott domains.

1 Introduction

Recently, much of the theoretical and practical research in programming languages has focused on extensions to *strong-typing*, where programs may be checked before run-time for a consistent use of types. One of these extensions, *polymorphism*, has been studied

[*]Partially supported by NSF Grants MCS-8402700, DCR-8603890, and DCR-8511190. The second author is supported by an NSF Graduate Fellowship. This paper will appear in the proceedings of the Third Workshop on the Mathematical Foundations of Programming Language Semantics, to be published by Springer-Verlag in the LNCS series.

[†]Current address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139

extensively. In a polymorphic language, parameters to procedures or functions may accept arguments of more than one type. (This notion of polymorphism can be traced to [Str67].) For example, one may write a "sort" routine that sorts lists of integers, characters, or strings. Compared with a strongly-typed but non-polymorphic language like Pascal, the ability to write one "sort" routine can be quite powerful. In Pascal, one would have to write three different sort routines for integers, characters, and strings – even though the algorithms are the same – since parameters can match arguments of only one type. Polymorphism adds convenience, expressibility, and security to a strongly-typed language, and for those reasons, it has been incorporated into ML [Mil78], HOPE [BMS80], Russell [DD85], and Ada [DoD83] (in the form of *generics* that accept types as parameters).

Because of its growing importance in the design of new languages, researchers in denotational semantics have examined various interpretations for polymorphism. The work has mainly dealt with versions of the lambda calculus, in order to study polymorphism on an elementary and "pure" level. Our goal was to study a "real" polymorphic language and see what complexities arose. We chose Miranda[1] [Tur85a,Tur85b] as our language. The most interesting aspect of the study was not, however, polymorphism, but *algebraic types*, another extension to types found in Miranda. In this paper, we describe how to interpret algebraic types in a model that can also interpret Miranda's polymorphism.

Miranda is descended from SASL, another language invented by Turner (see [Tur85a] and [Tur85b] for a history of Miranda). While there are minor syntactic differences between the two languages, the most important new features (at least in the view of the authors) are polymorphism and algebraic types, which create new types from old. [KS85] provides a denotational semantics for SASL, which can serve as a basis for much of the semantics of Miranda [Rie86]. ([Jon87] has also given a denotational semantics for Miranda, albeit an *untyped* semantics. Our work focuses on the meaning of types, and so differs from [Jon87] in its goal.) The semantics of Miranda will be given with respect to the *ideal* model of [MS82,MPS84,MPS86], which extends the untyped semantics of SASL to interpret Miranda's polymorphism.

The ideal model cannot, however, adequately interpret Miranda's algebraic types, and this paper shows how to extend the ideal model to interpret algebraic types. The approach here makes use of *kinds*, a concept introduced in [McC79] for models of the second-order lambda calculus. Kinds describe higher-order type constructors which, we argue, best represent the meaning of algebraic types. A careful development of the semantics of higher-order type constructors provides a way to interpret algebraic types, while using the ideal model as a basis for the semantics of Miranda. A soundness theorem shows that this extended semantics matches the type inference mechanism of the language.

This paper may be divided into three parts. We begin by introducing Miranda: Section 2 provides a brief description of Miranda's polymorphism and algebraic types, and Section

[1] "Miranda" is a trademark of Research Software Ltd.

3 explores various possibilities for interpreting algebraic types and justifies the approach taken in this paper. Then, in Sections 4 through 6, we sketch the details of the semantics of algebraic types. Finally, Section 7 shows how these semantics fit into the rest of Miranda's semantics, while Section 8 discusses remaining problems.

2 Miranda's Type System

2.1 Implicit Polymorphism

In most strongly-typed languages, a program must contain declarations of variables and their types. Miranda does not follow this convention, for types may be omitted from a Miranda program. Miranda uses an *implicit* typing system, similar to ML's [Mil78], in which the type-checker for the language deduces the types of variables from the context. As long as there are no conflicts in the types deduced for variables, a program is well-typed.

As an example of how the type-checker works, consider the Miranda function

```
    add x y = x + y
```

The parameters to `add` are `x` and `y`, and the body is `x + y`. In order to assign a type to these parameters, the type-checker examines the body of the function and finds that `x` and `y` must both have type Num for the body to be well-typed. Since we can assign consistent types to `x` and `y`, the function is well-typed. Functions are first-class objects in Miranda, and so we must also assign a type to `add`; its type is $Num \rightarrow Num \rightarrow Num$, because it takes two numbers and returns another number. (Note that "$\rightarrow$" associates to the right, so $Num \rightarrow Num \rightarrow Num$ is read as $Num \rightarrow (Num \rightarrow Num)$.) Types for functions are written in this curried notation, since applying one function of type $Num \rightarrow Num \rightarrow Num$ to a number, as in

```
    add3 = add 3
```

returns a function of type $Num \rightarrow Num$.

An implicit type-checker cannot always deduce a unique type for a variable. For example, consider the Miranda function

```
    id x = x
```

which is the identity function. Nothing in the body of this function constrains the type of `x`, so the type-checker would assign a type variable (such as α) as the type of `x`. To deduce a type for `id`, notice that it takes a value of type α and returns a value of type α. The type-checker then infers the type $\forall\alpha.\alpha \rightarrow \alpha$ for `id`, where the "$\forall$" symbol means that any type may replace α. `id` is an example of a *polymorphic* function, a function that accepts arguments of more than one type.

Miranda is considered an *implicit polymorphic* language, because polymorphic types are introduced by the type-checker, not by the programmer. The other major form of polymorphism is *explicit polymorphism*, in which types can be passed to functions. This form of polymorphism may be found in the second-order lambda calculus [Gir71,Rey74] and in Russell [DD85].

2.2 Algebraic Types

Another novel feature of Miranda's typing system is algebraic types. Algebraic types allow a programmer to define the data domains of the program [Tur85a]. For example, suppose we want to describe binary trees. One could represent binary trees in lists, as is done typically in Lisp, with the first element being the value at a node, and the next two elements being the left and right branches. This representation can be confused with other lists, though. Algebraic types allow the programmer to distinguish lists and trees through the creation of new data types, such as a type called "tree" which contains all binary trees.

Algebraic types can define completely new values. For instance, we can create a new type called "fruit":

```
fruit ::= Apple | Cherry | Peach
```

Simple algebraic types like this one resemble Pascal's enumerated types: the type "fruit" includes the values `Apple`, `Cherry`, and `Peach`. The name of the new type appears on the left of the "::=", and the names on the right are the values in this type, separated by a "`|`" which is read as "or". Any expressions in the program can use the values in `fruit` once it has been defined.

Algebraic types can also build new types from types already present in Miranda. Consider, for instance, the following algebraic type:

```
numchar ::= Partnum num  |  Partchar char
```

This algebraic type may be read as "`numchar` is comprised of the values `Partnum` followed by a number, or `Partchar` followed by a character." A program containing this algebraic type could use values like

```
Partnum 5
Partchar 'a'
```

both of which would be assigned the type *numchar* by the type-checker. Values like `Partnum 'a'` are not allowed, since the algebraic type specifies that a value of type Num must follow any occurence of the identifier `Partnum`.

The expressive power of algebraic types comes from the ability to use *recursion* inside definitions. Trees are one example of a recursive type, since trees are defined in terms of themselves, storing two subtrees at each internal node. In Miranda, one could create the algebraic type

```
treenum ::= Leaf num  |  Node num treenum treenum
```

to describe numeric binary trees. In this case, the algebraic type appears in its own definition – the identifier `Node` must be followed by a value of type Num, and two values of type *treenum*.

In Miranda, we can generalize this algebraic type. Suppose we wanted binary trees with numbers or characters at the nodes. One could define two different algebraic types, or alternatively, one could write

```
tree * ::= Leaf *  |  Node * (tree *) (tree *)
```

The * stands for "any type", and so this algebraic type describes a whole class of types, with instances being *tree* Num, *tree* $Char$, or even *tree* (*tree* Num). Values in these new types include:

```
Node 5 (Node 1 (Leaf 0) (Leaf 3)) (Leaf 11)
Node 'a' (Leaf 'b') (Leaf 'c')
```

with types *tree* Num and *tree* $Char$. A value not described by this algebraic type is

```
Node 12 (Leaf 'a') (Leaf 6)
```

because `Leaf 'a'` has type *tree* $Char$, which is inconsistent with `Node 12`. Miranda's type-checker would reject such a value.

Miranda allows more than one type variable (such as ** or ***) in algebraic types, as in

```
tagtype * ** ::= Left *  |  Right **
```

In addition, mutual recursion between algebraic types is also permitted, as in

```
tree1 * ::= Leaf1 *  |  Node1 * (tree2 *) (tree2 *)
tree2 * ::= Leaf2 *  |  Node2 * (tree1 *) (tree1 *)
```

Algebraic types are thus quite general, and this generality will complicate their semantics.

In the next section, we shall examine a few ways to interpret algebraic types. For simplicity, we assume that all algebraic types appearing on the right side of a "::=" have the appropriate number of types following them. For instance, an algebraic type like

```
tree * ::= Leaf *  |  Node * (tree) (tree)
```

should be considered illegal, because `tree` is used incorrectly on the right side. ([Rie86] describes a way to check algebraic types for these errors.) We also will not consider a further extension to algebraic types, in which "laws" are associated with an algebraic type [Tur85a]. In Section 8, we will briefly address how these *unfree* algebraic types may be incorporated into the semantics.

3 Interpretation of Algebraic Types

For any strongly-typed language like Miranda, a well-typed program always produces answers of the appropriate type. The denotational semantics of a typed language must mirror this operational behavior, or in other words, the meaning of any well-typed program must be in the appropriate type in the model. This result is commonly called a "soundness theorem", and in order to verify it, we must have an intepretation for types.

Given the extensive type system of Miranda, finding intepretations for every type is difficult. Without algebraic types, the task is easier – one can use the ideal model of [MS82,MPS84,MPS86], for it can interpret all the non-algebraic types, including polymorphic types. One would then like to find meanings for algebraic types in this model. At first glance, it does not seem possible to add algebraic types to *any* model, since they appear to create new types during the execution of a program. We must either extend the ideal model, or examine the model more closely, to make sure that types produced by algebraic types have a meaning.

At least three methods can be used to interpret algebraic types within the ideal model. One method involves describing a *collection of models* for each set of algebraic types. The model for a set of algebraic types would contain all types described by the algebraic types. Furthermore, each model would be an ideal model, so that the implicit polymorphism of Miranda could be interpreted. Finding the meaning of a particular program under these semantics would involve choosing a model that contained the appropriate algebraic types.

This method, though, requires an infinite collection of models, one for each possible set of algebraic types; it would be better if we could incorporate all of the algebraic types into *one* model. An alternative method involves finding a model in which all types described by algebraic types may be found. For example, consider the algebraic type

```
tree * ::= Leaf *  |  Node * (tree *) (tree *)
```

One can think of this algebraic type as defining two new functions, `Leaf` and `Node`, where `Leaf` takes a value of some type t and returns a value of type $tree\ t$, and `Node` takes arguments of types t, $tree\ t$, and $tree\ t$, and returns a value of type $tree\ t$. One can model `Leaf` as merely returning its value, and `Node` as returning a tuple of three items. To find a meaning for `tree`, we simply combine the values returned by `Leaf` and `Node`:

$$tree\ t = t\ +\ t \times (tree\ t) \times (tree\ t)$$

(where "+" constructs a disjoint union, and "×" constructs a cross product of the types). In [MPS84,MPS86], a method is given for interpreting such recursive type equations, so for any type t, there is a meaning for $tree\ t$ in the model. This method avoids the infinite collection of models of the first method.

The second method has another flaw – no meaning is found for the algebraic type itself, only for its instantiations. A third method, and the one we chose, treats algebraic types as *type constructors*, that is, functions which take types as arguments and return types. For example, one can think of the algebraic type `tree` as a function which takes one type and returns a new type, with the value of `tree` being

$$tree = \lambda t.t\ +\ t \times (tree\ t) \times (tree\ t)$$

This method seems to capture the meaning of an algebraic type better than the first two methods – an algebraic type has a meaning, not simply as a collection of related types. While this method requires adding another layer of values to the semantic objects to the model (the "kind" hierarchy of functions on types), the method is more semantically natural. Given the structure of Miranda, a semantics of Miranda should assign a meaning to every piece of the program, so algebraic types should have a denotational meaning. The following sections outline one method for achieving this goal.

4 Overview of Semantics of Algebraic Types

The decision to model algebraic types as type constructors requires that, in addition to ordinary, first-order functions, we include the meaning of higher-order functions (the functions on types) in our model. Incorporating higher-order functions is not a new idea: they were used in the first model of the second-order lambda calculus [McC79], and have subsequently been employed in other models of the second-order lambda calculus [BMM85,ABL86]. Interestingly, the ideal model originally given in [MS82] included higher-order functions to solve certain recursive type equations (This approach was abandoned in a later paper

[MPS84]). Our semantics extends the use of higher-order functions to model a particular language construct, algebraic types.

Higher-order functions, or type constructors, may be classified like first-order functions – instead of "types," the classifiers are called *kinds* [McC79]. *Kinds* are syntactic objects that classify type constructors according to the number of types they accept as arguments, and are specified by the grammar

$$\kappa ::= T \mid \kappa \Rightarrow \kappa \mid (\kappa)$$

The T in the grammar is shorthand for "type", while "$\Rightarrow$" denotes a function space. Accordingly, any simple type expression will have kind T, and type constructors will have more complicated kinds. The binary tree type constructor, for example, has kind $T \Rightarrow T$, since it returns a new type when given a type. The standard type constructors $\times$ and $+$ also have a kind – kind $T \Rightarrow T \Rightarrow T$ – since they construct a new type from two types. (Note that kinds, like types, are specified in curried form.)

In addition to classifying type constructors, kinds will also classify the sets that contain the meaning of type constructors. These sets in the model, denoted $Kind^{\kappa}$, will contain the meaning of all type constructors with kind κ. Our model will include an infinite number of these sets, one for each possible kind.

Along with kinds, we will use another concept, *constructor expressions*, adapted from [McC79] and [BMM85]. Constructor expressions are lambda terms specified by the grammar

$$\begin{aligned} cexpr \ ::= \ & Int \mid Real \mid Num \mid Bool \mid Char \mid Trivial \mid id \mid + \mid \times \mid List \mid \\ & \forall id^{T}.cexpr \mid (cexpr) \mid cexpr \rightarrow cexpr \mid \lambda id^{\kappa}.cexpr \mid cexpr\ cexpr \end{aligned}$$

where id represents any identifier. Constructor expressions will denote the meanings of type constructors in the model, serving the same purpose as the lambda calculus in denotational semantics. Notice that constructor expressions are *typed* lambda terms: any time a variable is bound by a lambda, a kind must be given for the variable.

The syntax of constructor expressions expresses more functionals than just the type constructors, since the syntax allows functions that accept other type constructors as arguments. One example of such a constructor expression is $\lambda t^{T \Rightarrow T}.t$, a function that takes a function of kind $T \Rightarrow T$ and returns it. The kind of this constructor expression is $(T \Rightarrow T) \Rightarrow (T \Rightarrow T)$. We will take advantage of higher-kinded bound variables in the definition of the *List* type constructor.

Having these ideas in mind, we now give the semantics of algebraic types. First, in Section 5, we will consider the semantics of constructor expressions. Using these semantics, we will show how to convert algebraic types into constructor expressions in Section 6.

5 Constructor Expressions

Both constructor expressions and algebraic types represent type constructors. Constructor expressions, however, have a more uniform syntax and provide insight into type constructors. Before considering the semantics of algebraic types, we carefully consider the semantics of constructor expressions. We devise a system that type-checks, or rather *kind-checks*, each constructor expression. (Algebraic types will be converted into well-kinded constructor expressions in the next section.) We then define a model and give the semantics of constructor expressions. In this section we also sketch a key result, namely that any well-kinded constructor expression has a meaning in the appropriate kind set.

5.1 Kind-Checking

Due to the presence of constants in the syntax for constructor expressions, not all constructor expressions have a meaningful interpretation. The syntax can generate constructor expressions like

$$\lambda t^{T \Rightarrow T}.t + t$$

that misuse these constants. (This expression does not respect the meaning of $+$, in that $+$ should be applied to two types, not two objects of kind $T \Rightarrow T$.) In order to detect such illegal expressions, we devise a deduction system that infers the kinds of expressions. Any expression for which a type can be inferred will be called *well-kinded.* If the system cannot infer a kind for an expression, the expression will be illegal.

The kind-checking system resembles many deduction systems. We first need a notion of *syntactic kind assignments.* During the inference of kinds, the system must be able to remember the kinds of free variables in subexpressions; syntactic kind assignments "remember" the kinds of these variables. Formally,

Definition 1 *A* syntactic kind assignment *is a partial function that assigns kind expressions to identifiers.*

To change an arbitrary syntactic kind assignment A, we use the notation

$$A[v_1 : \kappa_1, \ldots, v_n : \kappa_n]$$

which denotes the same syntactic kind assignment as A except at the variables $v_1, \ldots, v_n$ which are assigned kinds $\kappa_1, \ldots, \kappa_n$.

Like other formal systems, the kind-checking system consists of axioms and rules. We begin with the axioms of the system that determine the kinds of "simple" constructor expressions. We will use the standard logic notation

$$A \vdash \mu : \kappa$$

to mean that "assuming the free variables of μ have kinds determined by A, μ has kind κ."

1. $A \vdash \tau : T$, where $\tau \in \{Int, Real, Num, Bool, Char, Trivial, NIL\}$

   This axiom states that the base types have kind T.

2. $A \vdash v : (A\ v)$

   Any free variable has a kind determined by the syntactic kind assignment.

3. $A \vdash List : T \Rightarrow T$

   "List" is a special type constructor that builds lists of a given type. Its kind is $T \Rightarrow T$.

4. $A \vdash +, \times : T \Rightarrow T \Rightarrow T$

   The standard type constructors $+$ (disjoint union) and $\times$ have kind $T \Rightarrow T \Rightarrow T$ as noted before. As is customary, $+$ and $\times$ will be written infix.

In addition to these rules, we will need four rules of inference. Typically, a typed lambda calculus only requires two rules of inference, one for lambda abstractions and one for applications. Two additional rules are necessary for constructor expressions, in order to guarantee that the set of well-kinded constructor expressions can be interpreted by the semantics.

1. $\dfrac{A[t : \kappa_1] \vdash \mu : \kappa_2}{A \vdash \lambda t^{\kappa_1}.\mu : \kappa_1 \Rightarrow \kappa_2}$, ($t$ not free in the left argument of a "$\rightarrow$" or in any subexpression $(\forall s^T.\alpha)$ of μ.)

   This rule assigns a kind for lambda abstractions, and the restriction is necessary to accommodate the ideal model. Our model only considers continuous functions on types; unfortunately, "$\rightarrow$" is not a continuous function in the ideal model (for reasons which are given below) so we must not treat "$\rightarrow$" as a function that can be applied like "$+$" and "$\times$". (In the final section, we will consider models in which "$\rightarrow$" is continuous.) Also, notice that "$\forall$" is not considered a function on types. We do not know if this constructor is continuous in the ideal model.

2. $\dfrac{A \vdash \mu_1 : \kappa_1 \Rightarrow \kappa_2, A \vdash \mu_2 : \kappa_1}{A \vdash \mu_1\mu_2 : \kappa_2}$

   An application of a constructor expression with kind $\kappa_1 \Rightarrow \kappa_2$ to another with kind κ_1 is a constructor expression with kind κ_2.

3. $\dfrac{A[s:T] \vdash \mu : T}{A \vdash \forall s^T.\mu : T}$

   This rule says that if a constructor expression has kind T if a variable t is assumed to have kind T, then using "$\forall$" produces a constructor expression that is a type. This rule is necessary since the constant "$\forall$" is not treated as a function on types.

4. $\dfrac{A \vdash \mu_1 : T, A \vdash \mu_2 : T}{A \vdash \mu_1 \rightarrow \mu_2 : T}$

   As with "$\forall$", this rule is needed since "$\rightarrow$" is treated as a special constant.

By way of example of how these rules can be used, consider the derivation of the kind for $\lambda t^T.t \times t$:

Deduction	Reason
$\emptyset[t:T] \vdash t : T$	Axiom 2
$\emptyset[t:T] \vdash \times : T \Rightarrow T \Rightarrow T$	Axiom 4
$\emptyset[t:T] \vdash t \times t : T$	Rule 2
$\emptyset \vdash \lambda t^T.t \times t : T \Rightarrow T$	Rule 1

These axioms and rules completely define the kind-checking of constructor expressions.

5.2 Semantic Sets and Semantics of Constructor Expressions

Now we present a semantic model to interpret constructor expressions. In order to build the model, we will begin with the semantic model of Miranda without algebraic types. The ideal model makes it convenient to interpret other Miranda statements, as was stated earlier.

The model for the base language is constructed via a sequence of mutually-recursive domain equations. *Domains* are complete partial orders (cpo's), i.e. partial orders in which every directed set has a least upper bound, or *sup*, and that are additionally:

1. *consistently-complete* (every bounded set has a sup)
2. *ω-algebraic* (every element is a sup of basic elements)

(see [ABL86,Sco82]). The domain of values for the base language is constructed using the standard domain constructors $+$, $\times$, and $\rightarrow$ (as opposed to type constructors), using the series of equations:

$$
\begin{array}{lcl}
Value & \cong & Bool + Num + Char + NIL + Trivial + Newtype + Function \\
Bool & \cong & flat\ cpo\ of\ truth\ values \\
Int & \cong & flat\ cpo\ of\ integers \\
Real & \cong & flat\ cpo\ of\ representable\ reals \\
Num & \cong & Int + Real \\
Char & \cong & flat\ cpo\ of\ characters \\
NIL & \cong & flat\ cpo\ of\ the\ set\ \{nil\} \\
Trivial & \cong & flat\ cpo\ of\ the\ set\ \{()\} \\
Newtype & \cong & Value + (Value + Value) + (Value \times Value) \\
Function & \cong & Value \rightarrow Value
\end{array}
$$

(A *flat cpo* is a cpo in which $x \leq y$ if and only if $x = \perp$ or $x = y$.) These equations may be solved using the general methods given in [ABL86] and [Sco82]. The domain $Value$ is essentially the same domain used in the ideal model of [MS82,MPS84], which allows us to use their results.

Using this domain as a basis, we build a series of domains, the kind sets, that will contain the meaning of all constructor expressions. We first recall that in the ideal model, types are *ideals* which are nonempty, consistently-closed, downward closed subsets of a cpo [MS82]. We then let

$$Kind^T = \{A \subseteq Value \mid A\ is\ an\ ideal\}$$

This set includes the meanings of all type expressions. Furthermore, under the partial order of inclusion, $Kind^T$ is a domain [MS82]. Using this fact we build the other kind sets using the equation

$$Kind^{\kappa_1 \Rightarrow \kappa_2} = Kind^{\kappa_1} \rightarrow Kind^{\kappa_2}$$

which is the domain of all continuous functions (under the Scott topology) from the domain $Kind^{\kappa_1}$ to the domain $Kind^{\kappa_2}$. For example, the kind set $Kind^{T \Rightarrow T}$ is specified by the domain equation

$$Kind^{T \Rightarrow T} = Kind^T \rightarrow Kind^T$$

These domains, it will be shown, contain the meanings of all well-kinded constructor expressions.

Using these kind sets, we may now specify the semantics of constructor expressions. We define two semantic sets upon which the semantics will be defined:

$$
\begin{array}{rcl}
Cvalue & = & \bigcup_{\kappa \in KindExpr} Kind^{\kappa} \\
Cenv & = & Ide \rightarrow Cvalue
\end{array}
$$

$Cvalue$ contains the meaning of all constructor expressions, and $Cenv$ is the set of all *constructor environments.* A constructor environment allows the semantics to remember the values of free variables when interpreting constructor expressions. We will use the symbol η to represent an arbitrary constructor environment. In order to update a constructor environment, we use the notation

$$\eta[\mu_1/v_1, \ldots, \mu_n/v_n]$$

which means that the new environment has the same values as the old environment η except at the variables $v_1, \ldots, v_n$ which are assigned values $\mu_1, \ldots, \mu_n$.

Now we may define $Cval$, the semantic function that interprets constructor expressions. The interpretation of a constructor expression is determined from the following semantic clauses that specify $Cval$:

$$Cval : ConsExpr \to Cenv \to Cvalue$$

1. $Cval[\![Int]\!]\eta = \mathbf{Int}$, where **Int** denotes the ideal corresponding to the integers in the domain $Value$.

   Thus the base type Int has a meaning as an ideal. There are similar clauses for the other base types ($Real, Bool, Char, NIL, Trivial$) as well.

2. $Cval[\![Num]\!]\eta = Cval[\![Int + Real]\!]\eta$

   The type Num is the disjoint union of **Int** and **Real** in the model.

3. $Cval[\![v]\!]\eta = \eta\ v$, where v is an identifier.

   The meaning of a free variable is determined by looking up the variable in the constructor environment.

4. $Cval[\![+]\!]\eta = \oplus$, where $\oplus$ represents the disjoint union of ideals and is written in infix notation.

5. $Cval[\![\times]\!]\eta = \otimes$, where $\otimes$ represents the direct product of ideals and is written in infix notation.

6. $Cval[\![\mu_1 \to \mu_2]\!]\eta = \{f \in Function \mid f(Cval[\![\mu_1]\!]\eta) \subseteq Cval[\![\mu_2]\!]\eta\}$

   This definition of "$\to$" comes from [MS82], and is a fundamental definition in the ideal model. It states that any function in the domain $Value$ that takes objects in type μ_1 to objects in the type μ_2 is in the type $\mu_1 \to \mu_2$. This definition is quite intuitive, but there are problems with it that we discuss below.

7. $Cval[\![\forall t^T.\mu]\!]\eta = \bigcap_{a \in Kind^T} Cval[\![\mu]\!](\eta[a/t])$

   This definition is another fundamental definition in the ideal model. To see how it works, consider the polymorphic identity function. It lies in

$$\begin{aligned} Char &\rightarrow Char \\ Int &\rightarrow Int \end{aligned}$$

   and so on, according to the definition of "$\rightarrow$" given above. The identity function lies in all of these ideals, and so it lies in the intersection of the ideals. Polymorphic types are therefore modelled as intersections of ideals.

8. $Cval[\![List]\!]\eta = \mathbf{List} = \mathbf{fix}(\lambda l^{T \Rightarrow T}.\lambda t^T.\mathbf{NIL} \oplus (t \otimes l\, t))$, where **fix** is the least fixpoint operator in $Kind^{((T\Rightarrow T)\Rightarrow(T\Rightarrow T))\Rightarrow(T\Rightarrow T)}$. The operator **fix** is defined by the equation

$$\mathbf{fix}(f) = \bigsqcup_{i<\omega} f^i(\bot)$$

   where $f^i = f \circ f \ldots f$ with f appearing i times.

   This clause defines the *List* type constructor that builds homogeneous lists (i.e. those with elements drawn from a single type). This definition was given in [MS82].

9. $Cval[\![\lambda v^\kappa.\mu]\!]\eta = \lambda u \in Kind^\kappa.Cval[\![\mu]\!](\eta[u/v])$

   The meaning of a constructor expression representing a function is a function accepting arguments from the appropriate kind set.

10. $Cval[\![\mu_1\mu_2]\!]\eta = Cval[\![\mu_1]\!]\eta\ (Cval[\![\mu_2]\!]\eta)$

    A constructor expression representing an application is intepreted by applying the meaning of the function to the meaning of the argument.

Note that "$\rightarrow$" is not a function in these semantics; this is because it is not a continuous function, and so does not lie in any kind set. To see that "$\rightarrow$" is not continuous, consider the two ideals

$$\begin{aligned} Value &\rightarrow Int \\ Int &\rightarrow Int \end{aligned}$$

Any function in the first ideal is also in the second, so

$$Value \rightarrow Int \subseteq Int \rightarrow Int$$

However, $Int \subseteq Value$, so "$\rightarrow$" is *antimonotonic* in its first argument. Since any antimonotonic function is not continuous, "$\rightarrow$" is not continuous.

5.3 Soundness of Constructor Semantics

Having defined the semantics of constructor expressions and kind-checking rules for them, we must now show that they match via a soundness theorem. The theorem basically states that the meaning of any well-kinded constructor expression is in the appropriate kind.

The theorem relies on a connection between syntactic kind assignments and constructor environments. Intuitively, the value of a free variable, as defined by an environment, should be in the kind set determined by the syntactic kind assignment. We call this concept *compatibility*:

Definition 2 *Let A be a syntactic kind assignment, and η be a constructor environment. Then η is* compatible *with A, written $\eta \models A$, if whenever $v \in Ide$ and v is in the domain of A, $\eta\, v \in Kind^{(A\, v)}$.*

If $\eta \models A$, we know that the semantics of constructor expressions of the form v, where $v \in Ide$, are sound since their meaning is given by $\eta\, v$ which lies in the appropriate kind set $(A\, v)$.

We now state the theorem:

Theorem 1 *If $A \vdash \mu : \kappa$ and $\eta \models A$, then $Cval[\![\mu]\!]\eta \in Kind^{\kappa}$.*

Proof: (Sketch) We use induction on the length of the deduction of $A \vdash \mu : \kappa$. Beginning with the base case, when the length is 1, we notice that only axioms can give a valid proof of length 1. We can check all of the axioms to ensure that they satisfy the conclusion.

For the induction case, where $n \geq 2$, we assume that all proofs of length less than n satisfy the conclusion. We can then show, by case analysis, that if any rule is used as the last step of a deduction, then the conclusion holds. This proves the theorem.

6 Semantics of Algebraic Types

Having built up a theory of type constructors, we apply this theory to study algebraic types, the goal of this paper. Using the function $Cval$, we convert constructor expressions representing algebraic types to elements in the model. We show that the semantic functions produce elements in the model, ensuring that the semantics is well-defined.

6.1 Conversion to Constructor Expressions

We need a general technique to convert algebraic types to constructor expressions. The main difficulty lies in interpreting mutually-recursive algebraic types. Our method is based on a method used in [KS85] (which uses the method to solve sets of mutually-recursive, first-order functions in SASL). The idea is to construct a chain of environments, with the meaning of each algebraic type determined from its definition and the last environment. Taking the sup of the chain results in finding the sup of the approximations to each algebraic type.

To do this construction, we need two semantic functions *Usertype* and *Settype*. *Settype* builds an environment in the chain that contains an approximation to the meaning of a set of algebraic types. *Usertype* actually builds the chain of environments and takes the sup of the chain, using *Settype* to build each approximation in the chain. The definitions for these functions are:

$$Usertype : TypeDefs \rightarrow Cenv \rightarrow Cenv$$
$$Usertype[\![AlgTypes]\!]\eta = fix(\lambda\eta\prime.Settype[\![AlgTypes]\!]\eta\prime\,\eta)$$

$$Settype : TypeDefs \rightarrow Cenv \rightarrow Cenv \rightarrow Cenv$$

1. $Settype[\![AlgType; AlgTypes]\!]\eta\prime\,\eta = Settype[\![AlgTypes]\!]\eta\prime\,(Settype[\![AlgType]\!]\eta\prime\,\eta)$
2. $Settype[\![\tau\, t_1 \ldots t_n ::= id_1\, arg_{1,1} \ldots arg_{1,j_1} \mid \ldots \mid id_k\, arg_{k,1} \ldots arg_{k,j_k}]\!]\eta\prime\,\eta = \eta[e/\tau]$

$$\begin{array}{rcl} where\ e & = & Cval[\![\lambda t_1^T.\ldots\lambda t_n^T.subtype_1 + \cdots + subtype_k]\!]\eta\prime \\ for\ subtype_i & = & Trivial\ if\ j_i = 0 \\ & & arg_{i,1} \times \cdots \times arg_{i,j_i}\ otherwise \end{array}$$

Suppose, for example, that we wanted to interpret the following two mutually-recursive algebraic types using this method:

```
f * ::= Nil   |  Push * (g *)
g * ::= Cons * (f *)
```

To build a chain of environments, we start with

$$\eta_0 = \eta[Bot_{T\Rightarrow T}/f, Bot_{T\Rightarrow T}/g]$$

and define, for all $i \geq 0$,

$$\eta_{i+1} = Settype[\![AlgTypes]\!]\eta_i\,\eta$$

where Bot_κ is the bottom element in the domain $Kind^\kappa$. (Note that $Bot_{T\Rightarrow T}$ applied to anything of kind T returns Bot_T.) Constructing the chain of environments yields

$$\begin{aligned} \eta_1 &= \eta[(\lambda t_1^T.Trivial + (t_1 \times Bot_T))/f, (\lambda t_1^T.t_1 \times Bot_T)/g] \\ \eta_2 &= \eta[(\lambda t_1^T.Trivial + t_1 \times t_1 \times Bot_T)/f, (\lambda t_1^T.t_1 \times (Trivial + t_1 \times Bot_T))/g] \end{aligned}$$

and so on. Taking the sup of these environments is equivalent to calculating

$$fix(\lambda \eta\prime.Settype[\![AlgTypes]\!]\eta\prime\, \eta)$$

which is the definition of $Usertype$.

6.2 Well-Definedness of Semantics

We now show that our semantic functions for interpreting algebraic types produce valid constructor environments:

Theorem 2 *Let AlgTypes be a set of algebraic types defined*

$$\begin{aligned} AlgTypes = \quad & \tau_1\, t_{1,1} \ldots t_{1,j_1} ::= id_{1,1} arg_{1,o_1} \ldots arg_{1,p_1} \,|\, \ldots \,|\, id_{1,m_1} arg_{1,q_1} \ldots arg_{1,r_1} \\ & \qquad\qquad \vdots \\ & \tau_k\, t_{k,1} \ldots t_{k,j_k} ::= id_{k,1} arg_{k,o_k} \ldots arg_{k,p_k} \,|\, \ldots \,|\, id_{k,m_k} arg_{1,q_k} \ldots arg_{1,r_k} \end{aligned}$$

If η is a constructor environment, then

$$(Usertype[\![AlgTypes]\!]\eta)\ \tau_i \in Kind^{\kappa_i}$$

for $\kappa_i = T \Rightarrow \cdots \Rightarrow T$ with $j_i + 1$ T's.

Proof: (Sketch) To show that

$$(Usertype[\![AlgTypes]\!]\eta)\ \tau_i \in Kind^{\kappa_i}$$

we verify that for each η_j in the chain, that

$$\eta_j\ \tau_i \in Kind^{\kappa_i}$$

Taking the sup of this chain of environments is the same as taking the sup of each of the elements in the environments, so

$$(Usertype[\![AlgTypes]\!]\eta)\ \tau_i = \bigsqcup_{j<\omega} (\eta_j\ \tau_i)$$

Since $Kind^{\kappa_i}$ is a domain,

$$\bigsqcup_{j<\omega} (\eta_j\, \tau_i) \in Kind^{\kappa_i}$$

hence

$$(Usertype[\![AlgTypes]\!]\eta)\ \tau_i \in Kind^{\kappa_i}$$

which proves the theorem.

7 Connection to Miranda's Semantics

Algebraic types represent a significant part of the full semantics of Miranda, and the rest of the semantics must be defined so that the algebraic types fit into the model. In this section, we give an overview of the Miranda's semantics, tying it in to our work on algebraic types.

Specifying the semantics of Miranda, a typed language, consists of defining a set of type-checking rules, giving the semantics, and showing that the semantics is sound with respect to the type-checking system. Our type-checking system for Miranda was patterned after the systems found in [Mil78] and [MS82], since these type-checkers work with implicit polymorphic languages. The semantics for Miranda is based on the semantics given for SASL in [KS85].

Like the kind-checking system given before, the type-checking system for Miranda is composed of axioms and inference rules. An example of such an inference rule is

$$\frac{B \vdash g : \tau_2 \rightarrow \tau_3, B \vdash f : \tau_1 \rightarrow \tau_2}{B \vdash g.f : \tau_1 \rightarrow \tau_3}$$

where B is a *syntactic type assignment* that assigns types to variables, and "." is the operator in Miranda that composes two functions. This inference rule infers a type for a composition of two functions. With implicit polymorphic functions, the rules can get quite complicated; this rule, however, gives a flavor of the style.

To specify the semantics of Miranda, we use the semantics of SASL as a basis. A typical semantic clause looks like

$$Eval[\![g.f]\!]\rho = \lambda x.Eval[\![g]\!]\rho\ ((Eval[\![f]\!]\rho)x)$$

This clause interprets function composition, and the "ρ" is an environment that assigns values to identifiers.

The most important semantic clause, for the purposes of this paper, is the one that interprets algebraic types. Algebraic types define a new type *and* functions on that new type. These new functions must be added to the environment of a program, which the following semantic function $Tdecl$ does:

$Tdecl : TypeDefs \rightarrow Env \rightarrow Env$

1. $Tdecl[\![\tau\, t_1 \ldots t_n ::= id_1\, arg_{1,1} \ldots arg_{1,j_1} \mid \ldots \mid id_k\, arg_{k,1} \ldots arg_{k,j_k}]\!]\rho =$ $\rho[f_1/id_1, \ldots f_k/id_k]$

   where $f_i = \lambda e_1 . \ldots e_{j_i} . (e_1, \ldots e_{j_i})$

2. $Tdecl[\![AlgType; AlgTypes]\!]\rho = Tdecl[\![AlgTypes]\!]\,(Tdecl[\![AlgType]\!]\rho)$

The functions thus defined by algebraic types return tuples when given the appropriate number of arguments. This is the way we thought of these functions in Section 3.

The model for this semantics uses the underlying ideal model "Value". Given the semantics and the type-checking rules, one can prove the key theorem

Theorem 3 *If M is a Miranda expression with type τ, then*

$Eval[\![M]\!]\rho \in Cval[\![\tau]\!]\eta$

(For a more precise rendering of this theorem and its proof, see [Rie86].) Note that in order to prove this theorem, we need an intepretation for type expressions and, consequently, the algebraic types that appear in type expressions. It was the necessity to prove this theorem that generated the interest in modelling algebraic types.

8 Remaining Problems and Relevance to Previous Work

Our semantics has raised a number of issues in attempting to find a meaning for algebraic types. In this section, we clarify some of these issues, and propose solutions to them.

One issue that has been raised by Albert Meyer is the issue of structural equivalence versus name equivalence of types. Suppose, for example, we have the following two algebraic types in a program:

```
f ::= Nil1  |  Cons1 int f
g ::= Nil2  |  Cons2 int g
```

The type-checker will not allow us to write a value like

```
Cons2  1  Nil1
```

since `Nil1` is not of type `g`, so the expression would be rejected. Our semantics, however, does not distinguish between the two types – they are *structurally equivalent*. Operationally, Miranda distinguishes the values in the two types by tags – that is, Miranda uses *name equivalence* of types. One could, perhaps, tag every type in the model with an appropriate tag, but there may be complications in solving domain equations with tags.

An alternative approach is to consider the difference between structural and name equivalence to be simply a syntactic issue. That is, the type inference mechanism screens out all terms which are illegal under a name equivalence approach to typing. After that it does not matter if the semantics of the two types are different, since type-checking has eliminated the terms which would have been problematic. We leave open the inclusion of name equivalence and the issues it raises in semantics.

Another open problem is the incorporation of *unfree algebraic types* [Tur85a] into the semantics. Unfree algebraic types are algebraic types with laws for reducing elements to a normal form. For example, one could build a type consisting only of ordered lists:

```
olist  ::=  Onil   |  Ocons num olist
Ocons a (Ocons b x) => Ocons b (Ocons a x), a>b
```

This example is taken from [Tur85b]. The second line is the law for reducing an object of type *olist* to a normal form, and states that if `a>b`, reverse the order of `b` and `a` in the expression. [Tho87] gives one possible interpretation for an unfree algebraic type, by defining a set of functions that correspond to the reduction rules. For this example, we need only one function for the one rewrite rule:

```
Ocons' a Onil = Ocons a Onil
Ocons' a (Ocons b x) = Ocons b (Ocons' a x), a>b
                     = Ocons a (Ocons' b x)
```

One can then transform any program written with `Ocons` into one written with `Ocons'` by replacing any outer `Ocons` with an `Ocons'` and defining `Ocons'` in the program. One would then want to show that the values produced by `Ocons'` is a type in the model. For this example, the values produced by `Ocons'` are indeed a type in the model. (Proof hint: Without the law, the meaning of `olist` is a flat cpo, and so any subset of it including the bottom element is an ideal.) What is not known, however, is if this function construction can be generalized to all normalizing rewrite rules, and if the functions always produce values that form a type.

There is another alternative for adding unfree algebraic types. Since all terms of unfree types are required to have a normal form, the semantics could take advantage of this fact. We would find the interpretation of the corresponding free type and interpret the terms as the interpretations of their normal forms in that type. Like name equivalence, however, we leave the incorporation of unfree types as an open problem.

Our main problems, though, arise from problems in the ideal model, especially by the fact that "$\rightarrow$" is not a continuous function on types. If we allow algebraic types like

```
fun * ::= * -> num
```

in the language, we cannot interpret them. Using this example, the corresponding constructor expression would be

$$\lambda t^T.t \rightarrow Num$$

which is not continuous since it is antimonotonic in its first argument. There are polymorphic models, though, that can interpret constructor expressions like these: the closure model [McC79] and the finitary projection model [ABL86]. An interesting exercise would involve writing the semantics of Miranda using one of these two models as a basis.

We have, however, shown how to interpret many of Miranda's type constructors within the ideal model. The most important result in this paper is the practical use made of higher-order functions; these functions describe algebraic types very naturally, and are included in the semantic model. We hope that the use of higher-order functions on types in semantics will now be justified.

9 Acknowledgements

We would like to thank Albert Meyer for his suggestions on structural equivalence and name equivalence, and Matt Kaufmann for helpful comments on a preliminary version of this work.

References

[ABL86] Roberto Amadio, Kim B. Bruce, and Giuseppe Longo. The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In *First Annual Symposium on Logic in Computer Science*, Cambridge, MA, 1986.

[BMM85] Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The semantics of second-order lambda calculus. 1985. To appear, *Information and Computation.*

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: an experimental applicative language. In *Proceedings of the First International LISP Conference*, Stanford, CA, 1980.

[DD85] Alan Demers and James Donahue. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7:426–445, July 1985.

[DoD83] U.S. Department of Defense. *Reference Manual for the Ada Programming Language*, Springer-Verlag, New York, 1983.

[Gir71] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et al théorie des types. In J.E. Fenstad, editor, *Second Scandanavian Logic Symposium*, pages 63–92, North Holland, Amsterdam, 1971.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[KS85] Matthew Kaufmann and Douglas Surber. *Syntax, Semantics, and a Formal Logic for SASL.* Technical Report ARC 85-03, Burroughs Austin Research Center, January 1985.

[McC79] Nancy McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* PhD thesis, Syracuse University, 1979.

[Mil78] Robin Milner. A theory of type polymorphism. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MPS84] D. B. MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the Eleventh ACM Symposium on the Principles of Programming Languages*, Salt Lake City, UT, pages 165–174, 1984.

[MPS86] D. B. MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71, 1986.

[MS82] D. B. MacQueen and Ravi Sethi. A semantic model of types for applicative languages. In *1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, PA, pages 243–252, 1982.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation, Lecture Notes in Computer Science 19*, pages 408–425, Springer-Verlag, New York, 1974.

[Rie86] Jon G. Riecke. A denotational approach to the semantics of polymorphic languages. B.A. Honors Thesis, Department of Computer Science, Williams College. 1986.

[Sco82] Dana S. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages, and Programming, Lecture Notes in Computer Science 140*, pages 577–613, Springer-Verlag, New York, 1982.

[Str67] Christopher Strachey. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen. August 1967.

[Tho87] Simon Thompson. Lawful types in Miranda. Unpublished manuscript. 1987.

[Tur85a] David A. Turner. Functional programs as executable specifications. In Hoare and Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[Tur85b] David A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, Lecture Notes in Computer Science 201*, Springer-Verlag, New York, 1985.