PROSPER:
A Language for Specification by Prototyping

Jacek Leszczyłowski
Institute of Computer Science
Polish Academy of Sciences
P.O. Box 22
00-901 Warsaw PKiN POLAND

James M. Bieman
Department of Computer Science
Iowa State University
Ames, Iowa 50011 USA

(28 September 1988; revised 14 February 1989)

Abstract

The PROSPER functional specification language supports a "specification by prototyping" paradigm and relies on a unique and powerful type specification facility. Executable polymorphic specifications can be built from a small set of primitives. Types and functions are treated as values and can be the arguments and results of functions. Flexible parameterized type expressions are used to specify polymorphic functions and abstract data types. Abstract model specifications can be defined so that the invariants of a data structure can be part of the type, and only objects that meet the invariant are correctly typed.

Keywords: prototyping, specification, functional languages, parameterized types, dynamic typechecking, operational specification languages.

Department of Computer Science Technical Report #88–18
Iowa State University

To Appear in *Computer Languages*

Address correspondence to Dr. James M. Bieman, Department of Computer Science, Iowa State University, Ames, IA 50011

1 Introduction

Both Brooks and Turski describe the derivation of complete and consistent specifications as the "essence" of sofware engineering [1, 2]. Both authors stress the importance of the debugging of a specification, and Turski asserts that the only way to validate a specification is by testing, "as there is no other way"[2].

"To specify a system" can mean either "to describe properties of the system" or "to build a model of the system." A specification language can support the latter by enforcing a "specification by prototyping" paradigm. The specification by prototyping approach is consistent with the automation-based paradigm of Balzer et al and the operational approach of Zave [3, 4].

A program specification that executes provides a mechanism for both the design and debugging of a specification. An executable specification can demonstrate the functionality of a program even if the execution is much slower than required of the actual implementation. Inital results of Fickas et al indicate that an effective way to find errors in a specification is by proper example generation to test hypothetical cases [5].

Executable specifications can aid in the traditional development of software. An executable specification can be used in place of a subroutine stub when building a system. Thus, the partially implemented system will exhibit the full functionality of the final system. The executable specification can serve as an oracle in an automated testing environment.

We can build an executable specification of a complex system from executable specification constructs. Although the specification constructs must be executable, we are not overly concerned with execution efficiency. An efficient implementation can be developed from the executable specification either using traditional software engineering methods or through the use of program transformations [6].

In this paper, we describe the use of PROSPER (PROtotypes and SPEcifications with Relative types) as a tool for defining executable functional specifications. PROSPER is a functional language which can be used to express abstract model specifications. Functional programs are a natural mechanism for expressing functional specifications (e.g., MIRANDA and *me too* [7, 8]). PROSPER differs from the foregoing functional languages in its treatment of types as values. PROSPER also has features that are tailored to follow the abstract model approach for defining rigorous specifications.

The abstract model approach is described by [9, 10, 11, 12]. Two non-executable abstract model specification languages are VDM (meta-iv) [13] and Z [14]. Using the abstract model approach, a software system is specified as an abstract data type (ADT) using a minimum of well understood primitive types. This ADT consists of a formal domain specification and operations on the domain. The domain consists of a source set, which describes the structure of objects in the domain, and an invariant, which restricts the domain to a subset of the source set. First order predicate calculus is used to specify domain invariants and the pre and post conditions of operations. With some constraints, the constructs used in abstract model specifications are potentially executable. A major focus of building abstract model specifications is on defining types and operations on types [11].

In order to support type specification, PROSPER functions can specify types in the same manner as ordinary values. The idea of treating types as values is not new (see, [15]) and is a result of the natural evolution of programming language development. In Pascal, values, functions, and types are completely separate categories of the language. In ML [16], functions are values and are separate from types. PROSPER values include both functions and types. We treat a type as a domain of values and do not include the related operations as part of a type. Modules associate particular operations with a type when specifying an ADT.

PROSPER specifications are described in terms of fully typed objects and in particular (possibly higher order) functions. These functions can process types and functions as well as ordinary values. For example, the *List* function specified in Section 5 takes a type as an argument and produces a type as a result. Therefore, *List(integer)* and *List(boolean)* represent two different types. The generic binary search tree example in Section 5 is even more flexible. In addition to parameterizing the types of objects placed in a tree, we parameterize the boolean function used to determine the location for objects in a tree.

PROSPER has two mechanisms for introducing parameters in type expressions. These mechanisms allow the specification of polymorphic [16] and dependent types [17, 18]. One mechanism is similar to that of the programming languages ML [16] and Pebble [15]. The other mechanism is new. Polymorphic and dependent types allow the parameterization of components within type expressions.

Because of the flexibility of PROSPER type specification, complete type checking can be accomplished only at execution time. Therefore, we describe PROSPER as a *fully typed* language rather than a strongly typed language. Some type checking is possible at compile time, but type checking is limited since some type expressions can involve computations. Consider two type expressions $t[f(a)]$ and $t[g(b)]$ where f and g are functions and $t[\cdots]$ is a context such that $t[x]$ is a type expression. Although the two type expressions are equal when $f(a) = g(b)$, this fact cannot be determined at compile time. At execution time, both $f(a)$ and $g(b)$ are evaluated and then type checking can be completed. Run time type checking has been advocated for use in programming environments [19]. Dynamic typechecking is even more appropriate in a specification environment where execution speed is less important.

In traditional languages such as Pascal, we create types, such as arrays, at compile time. We can view a Pascal array as a compile-time function that takes the array bounds and the element type as an argument and produces a type, say *array[1..10] of integer*, as a compile-time result. In PROSPER, a similar type specification function operates at run-time, and operates in essentially the same manner as functions over "normal" types. As a result, types become "first class citizens." In a fully typed language, functions that operate on types must be also be typed. Thus, we must deal with a richer notion of a type. PROSPER uses a two-level type system to avoid logical paradoxes that can result from the use of "type type" as a type [20]. In this system "normal" values are typed, while types can be *supertyped.*

PROSPER can be viewed as the kernel of a specification system. Starting with primitive executable constructs we build towards higher level executable specification structures. Using PROSPER, we can "grow" executable versions of the primitives used in the abstract model approach. To define an executable version of the SPECS language used by [10], we need to specify (1) executable versions of the set, sequence, and labeled tuple (record) type building constructs, and (2) define **executable** functions that provide a pragmatic subset of the first order predicate calculus. Although some constructs used to define non-executable specifications cannot be directly executed, analogous executable specifications can be defined.

In this paper we describe the prototype version of PROSPER that we are currently implementing. Section 2 describes the unique features of PROSPER, necessary language constructs are described in Section 3, and Section 4 presents the PROSPER type operators. In Section 5, the PROSPER specifications of a generic list and binary tree ADT are described. Section 6 compares PROSPER with related approaches.

2 Unique PROSPER Features

Every PROSPER lexical item is a *symbol*. From symbols, the PROSPER linguistic constructs are built. These basic constructs can be classified as values, parameters, and names. A *value* is the direct representation of "semantic values" in the language; we briefly introduce two important aspects concerning values:

1. Representations of "semantic values" are not necessarily unique (integers have unique representations while reals do not, e.g. *1 @ integer, 1 @ real, 1.0 @ real*).
2. Some "semantic values" do not have standard direct representations. For example, functions defined by users as well as new user defined data types do not have direct representations (user defined types are represented indirectly via the results of functions). Users, however, can declare some objects to be values and the system incorporates conventions for introducing values for types built with the SUB and SUM operators (see Section 4).

Parameters are local names of values, while *names* (of values) are global names of values. We assume a LISP like binding mechanism for global names and do not discuss the mechanism further in this paper. Our focus here is on the features that are unique to PROSPER.

2.1 Values and Parameters

Values are built from symbols and expressions using the "@" operator. Values are typed and denoted *leftpart @ rightpart* where the *leftpart* represents a "quantity" and the *rightpart* represents a "type". Example values include *1 @ integer*, *abc @ character*, *7.5 @ real*, and *boolean @ TYPE*. The *"@"* symbol is used to separate quantities from a type designator when denoting values. The "*@ rightpart*" portion of a value is ommitted only when the type of the *leftpart* can be inferred. That is, the value has been declared as a triple and there is no ambiguity. For example, *7* is a symbol and not a value. We cannot determine out of context whether *7* represents the value *7 @ integer* or *7 @ character*. Note that the problem of ambiguous values is resolved in Ada with qualified expressions, i.e., INTEGER'(7).

When specifying a **parameter** the **":"** symbol is used to separate a parameter name from the type of value that it represents. For example *'x:integer*, *'y:character*, and *'t:TYPE*. Note that parameters have a **'** symbol as a prefix. The use of a **":"** symbol when specifying parameters and a **"@"** symbol for specifying values reflects the subtle difference between the specification of values and parameters. Assume that the input parameter of a function is specified as *'x:integer*. The *'x* symbol in the body of the function represents some value (of type integer) to be determined at execution time. Since all values are typed, the *‘x* must be replaced by a **complete** value triple such as *7 @ integer*.

The same **“:”** notation is used to express type relations for function application. Consider the triple *f @ integer* $\rightarrow$ *integer* which is a function producing an *integer* result from an *integer* argument. If we use *1 @ integer* as an argument we get an integer result which is expressed as *f(1 @ integer):integer*. The result value itself, *f(1 @ integer)*, represents a triple which is “some integer symbol” *@ integer*. The symbol *“:”* used in *f(1 @ integer):integer* specifies that the result is a triple with a “*@ integer*” component. The notation *f(1 @ integer) @ integer* is not used because this combination of symbols would be interpreted as “some integer value” *@ integer*, e.g., *(7 @ integer) @ integer*, which is not desired.

Although the differences between the meaning of “:” and “@” are subtle at first inspection, the added precision is necessary when describing functions that manipulate types in the flexible manner of PROSPER.

2.2 The Basic World and Super World

The semantic domains of PROSPER are divided into a “basic world” and a “super world”. The basic world contains the normal values of computation and the super world contains objects that may be types. The two “worlds”, the basic world and super world, correspond to the object level and meta level in logic. The reason for having the two worlds is to avoid logical inconsistencies when dealing with “type type” [20]. In PROSPER, TYPE is not a type; TYPE is a super-type and belongs in the super world. Constable et al propose an infinite hierarchy of type universes, U_i, to avoid logical paradoxes in their system for automating mathematics [21]. Mitchell and Harper suggest that the typing rules of Standard ML can be understood in terms of universes U_1 and U_2 [22]. PROSPER uses a basic world and a super world as a pragmatic subset of an infinite hierarchy of universes.

In the basic world we have:

Basic Values: All values have a type component. Primitive basic values include *integer, boolean,* and *character* values. In addition to user introduced basic objects (which do not have direct representations) there is a primitive value of *$ @ single*.

Functions: Standard *boolean* (*&, or, not, etc.*) and *integer* operations (+, −, *mod*, etc.) are included as primitives. Users can also define functions.

Types: Primitive types include *integer, boolean, character*, and a special type *single* which can only have a value *$ @ single*. Structured types are built from primitive types using type expressions and type building operators like “×” (Cartesian product) and “→” (functions). Types can be defined using super-functions which return a type.

In the super world we have:

Super-Values: A super-value is an object of the form *e @ T* where *e* is an expression and *T* is a super-type (and *e* is of super-type *T*). Example super-values include *integer* @ TYPE and *(1 @ integer)* @ VALUE.

Super-Functions: Super-functions have at least one argument or a result that is a super-value. The primitive function *Cartesian?* @ TYPE$\rightarrow$*boolean* is an example. The *Cartesian?* function determines whether its argument, which is a type, is a Cartesian product of types.

Super-Types: There are two primitive super-types, VALUE and TYPE. TYPE represents the notion of type type. VALUE represents the union of all basic value types and TYPE. Other super-types can be built using "$\times$" and "$\rightarrow$" operators (as in the basic world) and by using the **sub** operator. For any super-function F @ TYPE$\rightarrow$*boolean*, one can specify a super-type
$(F)\mathbf{sub} = \{t : \text{TYPE} \mid F(t) = \textit{true}\ \}$.
which is used to express "restrictive polymorphism".

PROSPER has a polymorphic "=" operator used to compare elements of the same type or super-type. Thus, *1 @ integer = 1 @character* is an illegal expression. This operator is defined for elementary types and super-types and is automatically extended when new types are defined using the SUB and SUM superfunctions which are described in Section 4.

2.3 Parameters in Type Expressions

Type expressions can have parameter components which are used to form polymorphic [16] and dependent types [17, 18]. We understand *polymorphism* as the ability to handle objects of many types. We follow the ML way of dealing with *implicit polymorphism* through free type variables occurring in type expressions. For example, the identity function, IDENTITY @ *'some:*TYPE$\rightarrow$*'some* in PROSPER, is a polymorphic function in the above sense (where *'some* is a type variable or *type parameter*). The identity function can process arguments of many types and returns an object of the same type as its argument. We understand *dependent type* as a type of function whose type of result depends on the values of the arguments. For example, the function

F('x:integer) = **if** *'x=0* **then** *true* **else** *100*

has a dependent type according to the above description. We will represent the type expression for F as F @*integer.'i* $\rightarrow$ **if** *'i=0* **then** *boolean* **else** *integer* (where *'i* is a value variable, or *element parameter*).

A parameter used in a type expression has a scope that includes the smallest type subexpression where the parameter occurs. For example, consider the type expression:

integer$\rightarrow$*'some:*TYPE$\rightarrow$*'some*

where *'some* is a parameter. The smallest subexpression where *'some* occurs is:

*'some:*TYPE$\rightarrow$*'some*

because the type expressions are by default treated as if parenthesised to the right. This corresponds to left associativity of arguments to functions when a function is invoked. Thus a function *f* @ *boolean→integer→integer* = *f* @ *boolean→(integer→integer).*

When invoked, *f true 5* @ *integer* = *(f true) 5* @ *integer.*

Two operators are used to introduce parameters. One operator, ":", is used to specify a super-type of a parameter (a *type parameter*). The other operator, ".", binds a name (an *element parameter*) to a value of a type or super-type. The ":" operater can be used to describe the types of polymorphic functions while the "." operator can be used to specify dependent types. (For a general discussion of types see [23].) The difference between the ":" and "." is described further.

We examine the ":" operator first. Consider the type expression: *'some:*TYPE → *'some.* In this expression the type parameter *'some* can be bound to any value T of super-type TYPE. The type expression is polymorphic and is used to specify a type of a function that maps a domain T @ TYPE to T @ TYPE. Thus, *'some:*TYPE → *'some* can be instantiated to: *integer* → *integer*, *boolean* → *boolean*, and *foo* @ TYPE → *foo* @ TYPE. Consider the application of a function G @ *'some:*TYPE → *'some* to an argument A @ *footype.* The type parameter *'some* is bound to the type of the argument to G which, in this case, is *footype.* The type subexpression *'some:*TYPE specifies that *footype* must be an element of supertype TYPE. That is, *footype* must be a type. The full type expression specifies that if the argument to G is of type *footype* then the result is also of type *footype.* Consider the function F @ *(('some:TYPE → some) → boolean).* F takes a function of type *'some:TYPE → 'some* as its argument and produces a result of type *boolean.* The successor function *succ* @ *integer → integer* could be used as an actual parameter to F.

To demonstrate the "." operator, consider a choice function G @ *TYPE.'t → 't.* G is a function that takes a type as an argument and produces a result of the same type as its argument. The element parameter *'t* is bound to the value supplied as an argument to G, and the argument to G must be an element of supertype TYPE. If the argument to G is *integer* @ *TYPE*, the result will be an integer value. A boolean result is produced when the value *boolean* @ *TYPE* is the argument. Thus we can use the notation *G(integer):integer* and *G(boolean):boolean.*

We say that "." is dual to ":" since: (1) *e:t* stands for the value of e with the specified type t; (2) *e.'y* stands for a type or super-type which is the value of expression e and *'y* names an argument of e. When an actual element of the type or super-type e is known for the type expression, perhaps during partial evaluation, this element replaces all occurrences of *'y* in the expression.

Consider the component *'x:t* of a larger type expression *'x:t → z.* This component *'x:t* is evaluated as specifying type *'x* where *'x* is some element of supertype t. Thus, the type expression component *'n:integer* specifies an element of type integer and is **not** a type. A subexpression *t.'y* of a larger type expression *t.'y → z* specifies type t. However, when a value of type t is (at "execution" time) supplied as an argument corresponding to the expression component *t.'y*, this value is bound to all occurrences of *'y* in z. Thus *integer.'y* specifies type *integer.* When a specified *integer* value is supplied at execution time, this value (not type) is bound to all occurrences of *'y* in z.

The notation works not only for types but for other values. Using the functions F @ *integer* → TYPE and G @ *integer.'n → F('n)* we find that the result of *G(1)* is of type *F(1)* and therefore *G(1):F(1)*. We can also see that the result of *G(2)* is of type *F(2)*, and so on. Thus, we find that *G(1):F(1), G(2):F(2)*, ... This flexibility in creating dependent types is quite useful when specifying generic functions.

3 PROSPER Constructs

To ease the readability of PROSPER specifications we use a number of notation conventions: boldface for PROSPER keywords such as **value**, capital letters for the primitive super-types and super-values such as VALUE and TYPE, and italics for other symbols such as *integer, boolean*, etc. The name of functions that return a *boolean* end with a "?" character.

The examples use the following forms of PROSPER expressions:

- type expressions: as previously described
- conditionals: **if** E_1 **then** E_2 **else** E_3
  where E_1 is of type *boolean*
- tuples: E_1, E_2
  where if E_1 is of type T_1 and E_2 is of type T_2 then
  $(E_1, E_2) : (T_1 \times T_2)$ is a 2-tuple.
  The functions *first* and *snd* select components:
  first($(5,6)$ *: (integer*$\times$*integer))*= 5 @ *integer* ,and
  snd($(5,6)$ *: (integer*$\times$*integer))*= 6 @ *integer*
- application: *function operand*
  as in *factorial 3*. Parentheses are used for grouping, so *factorial(3)* = *factorial 3*.
- local declaration: **let** *p* **be** *e1* **in** *e2*
  where *p* is a parameter construct and *e1, e2* are expressions as in:
  let *'t1* $\times$ *'t2* **be** *integer* $\times$ *boolean* **in** *'t2* $\times$ *'t1*.

Declarations can be of the following form:

- simple declarations: **value** *S* @ *T* **is** *E*
  which associates the symbol *S* with the value of expression *E* which is of type *T*.
- descriptive declarations:
  value *S* @ $T_1 \rightarrow T_2$ **such-that** *S 'a* **is** *E*
  which associates the symbol *S* with the value of expression *E* and all occurrences of *'a* in *E* are replaced by the argument, *'a:*T_1, and *E:*T_2.
- module declarations:
  module *M* **export** S_1 @ E_1; S_2 @ E_2; ...
  from D_1; D_2; ... **end-from**

Modules facilitate the creation of ADT's. The exported symbols appear as primitives to the user while the representation is hidden in the **from** clause. The representation is a sequence of declarations (D_1; D_2; ...).

4 Special Type Operators

One powerful feature of PROSPER is the ability to create subtypes. The flexibility of the subtyping capability is a direct result of the use of types as arguments and results of functions. For example, we can define a new type *evens* which consists of all even integers. Superfunction SUB creates the subtype. SUB is supplied a function that can test a particular value of a specified type to see if it meets a restriction. Any values of the specified type that meet the restriction are members of the subtype. A full description of SUB itself is SUB @ *('some:*TYPE→*boolean)*→TYPE. The following specifies *evens*:

> **value** *evens* @ TYPE **is** SUB(*IsEven?*)

where *IsEven?* is specified:

> **value** *IsEven? @ integer→boolean*
>   **such-that** *IsEven? ('A)*
>   **is** *mod('A, 2 @ integer) = 0 @ integer*

Type *evens* is defined in terms of function *IsEven?*. For expression *s @ evens* to be valid, the result of the application of *IsEven?* to *s @ integer* must be *true*. For example, *4 @ evens* and *6 @ evens* are correct expressions since *IsEven? (4 @ integer) = true* and *IsEven? (6 @ integer) = true*. The expression *5 @ evens* generates an error. Note that the @ operator allows us to create values of a new type in a standard "automatic" way. As the example shows, a subtype is created by specifying a function that determines whether a value meets a specified criterion for restricting a larger type.

Type *evens* can demonstrate the PROSPER approach to the issues of type equivalence and inheritance (We use the term "type relativity" for the PROSPER mechanism dealing with inheritance issues). When the representation for a type is accessible, structural equivalence is used and operations are inherited from the source type. For example, *4 @ evens* = *4 @ SUB(IsEven?)* and defined integer operations such as $+, \times, \div$, etc. can be performed on *evens*. The algorithm used to perform *evens* addition is (1) convert the operands from *evens* to *integers*, (2) perform *integer* addition, and (3) convert the result to evens, if possible. For example: *2 @ evens* divided by *2 @ evens* is *1 @ integer*, in contrast to *4 @ evens* divided by *2 @ evens* is *2 @ evens*.

Inheritance and structural equivalence are restricted through the use of modules. Consider the module:

> **module** *Evens-Module*
> **export** *even-primitive* @ TYPE
> **from value** *even-primitive* **is** *SUB(IsEven?)*
>   **comment**
>   *4 @ even-primitive* = 4 @ SUB(IsEven?) here
>   **end-comment**
> **end-from**
> **comment**
>   *4 @ even-primitive* ≠ 4 @ SUB(IsEven?) here
> **end-comment**

Specifications within the **from** clause are hidden from external specifications. Since only the type name *even-primitive* is exported, users cannot access the representation, *SUB(IsEven?)*. Outside the **from** clause

1. integer operations are not defined over the relative type *even-primitive*, and

2. name equivalence between types is used for any value of type *even-primitive*.

The SUM super-function creates a type that is a possibly infinite union of types. SUM is supplied a function that can test a type to see if it meets a criterion and builds a "disjoint union" of types. Elements of the new type are built from all elements of all types that meet the criterion. SUM @ (TYPE→*boolean*)→TYPE is really a partial super-function. Its argument must be a function which returns *true* for countably many types. To demonstrate the use of SUM, we can define type *int-bool* which is the union of types *integer* and *boolean*:

```
value int-bool @ TYPE is SUM(Int-or-bool? )
value Int-or-bool? @ TYPE→boolean
   such-that Int-or-bool? (‘x) is
      ‘x = integer or ‘x = boolean
```

Type *int-bool* is defined in terms of the function *Int-or-bool?*. For a value *v* to be correctly typed as *int-bool*, *true* must result from the application of *Int-or-bool?* to the type of *v*. Thus *(1 @ integer) @ int-bool* and *(false @ boolean) @ int-bool* are correctly typed values.

The leftside of a value of type *int-bool*, or any type defined via SUM, must have a type designator. Unlike SUB, SUM cannot always infer the type. For example, *(7 @ integer) @ SUM(Int-or-char?)* is not the same value as *(7 @ character) @ SUM(Int-or-char?)*, and thus *7 @ SUM(int-or-char?)* is ambiguous.

Two functions associated with SUM create objects of a summed type and retrieve values in their "original" types:

1. Inj — injects new values into the summed type. Inj is supplied the TYPE→*boolean* function that SUM uses to define a specific summed type and the base value to be injected. Inj produces a value of the summed type:
   (Inj *Int-or-bool?*) *(1 @ integer) = (1 @integer)* @ SUM(*Int-or-bool?*)

2. Proj — projects values the other way:
   Proj*((1 @ integer)* @ SUM*(Int-or-bool?)) = 1 @ integer*

The types of these functions are:
  Inj @ (TYPE→ *boolean).‘F* → *‘some:(‘F)***sub** → SUM(*‘F*)
  Proj @ SUM*(‘F).‘e* → BASE-TYPE*(‘e)*
where BASE-TYPE is a superfunction which, for an element *e* of a given sum-type, returns the type of the element before injection. For example,
  BASE-TYPE*((1 @ integer)* @ SUM*(‘F)) = integer* @ TYPE
SUM differs from discriminant union types (Pascal variant records) in that SUM can create an infinite union of types. Such an infinite type is used to define a generic list and a generic binary search tree ADT specification in the following section.

5 Example Specifications

We demonstrate two useful type building functions constructed from the PROSPER primitives. The first example is a specification of a generic list ADT. Although the *Generic-List* type operator is not a primitive in PROSPER, we can easily define a generic list ADT that is similar to a LISP list, but is fully-typed. A list type is generated by the *List* function when supplied a type as an argument. Once a list type is generated, all of the normal list operations can be performed on objects of that type. The second example is a generic binary search tree specification. This example is a highly flexible, robust, and secure type building function: the specification incorporates an invariant as part of the specification; preconditions of operations are enforced through type checking; and highly customized types of binary search trees can be defined. For example, a binary search tree is usually defined only to contain elements of types where a comparison operator such as $>$ is defined. However, the PROSPER specification allows the user to supply the comparison function as a parameter.

5.1 A Generic List Specification

Figure 1 displays the generic list specification. The **export** section of the specification contains all of the information necessary to access the ADT *Generic-List*. The actual representation is expressed in the **from** section, and the user need not be concerned with the representation. In this example, the *List* type constructor is not defined in the usual recursive manner. A *List* type is defined as the union of the infinite set of types marked by *Is-List-Rep?*. Essentially the SUM superfunction unions the types "list of length 0," "list of length 1," "list of length 2," etc. Note that the type marking functions can be expressed by context-free grammar rules. Thus, *(Is-List-Rep? 'some)* can be represented by the rule:

$\mathrm{T} \Longrightarrow \mathit{single} \mid \mathit{'some} \times \mathrm{T}$

We union the family of types generated by this rule for our representation of lists. This approach allows us to eliminate recursion and all "heavy machinery" concerning fixed-points. It seems to be simpler than Scott's theory approach to recursive types [24]. For more details and the semantics of all the type constructors (defined in set theory) see [25].

To use the ADT *Generic-List* we invoke the *List* function with a TYPE argument. For example, *List(integer)* is the type "list of integers," *List(real)* is the type "list of reals," and *List(foo)* is the type "list of foo." We can use the ADT specification to introduce some list values:

value *Newlist @ List(integer)* **is** *Nil @ List(integer)*
value *A @ List(integer)* **is** *Cons(5 @ integer, Newlist)*

Only integer values can be added to *Newlist* and *A*. And all exported *Generic-List* operations can be performed on *A*. However the *Head* and *Tail* operations are insufficiently specified for empty lists such as *Newlist* and the system response is undefined. We can correct the situation by specifying a precondition that prevents the application of *Head* and *Tail* to an empty list. First, we introduce the following convention used throughout the rest of the paper. If *f* and *g* are the symbols of functions then *f_g* is the symbol of their composition. For example *Not_Null? 'L* is *Not(Null? 'L)*. We use it in the new version of the *Generic-List*

```
module Generic-List
export
   List @ TYPE→TYPE;
   Nil @ List(‘some:TYPE);
   Head @ List(‘some:TYPE)→‘some;
   Tail @ List(‘some:TYPE)→List(‘some);
   Cons @ (‘some:TYPE×List(‘some))→ List(‘some);
   Null? @ List(‘some:TYPE)→boolean
from
   value List such that List(‘t) is SUM(IsListRep? ‘t);
   value IsListRep? @ TYPE→TYPE→boolean
     such that IsListRep? ‘a ‘b is
       if ‘b = single then true
       else if Cartesian? ‘b then let ‘z × ‘y  be ‘b
         in (‘z = ‘a) & (IsListRep? ‘a ‘y)
         else false;
   value Nil is ($ @ single) @ SUM(IsListRep? (‘some:TYPE));
   value Head such that Head(‘x) is first(Proj(‘x));
   value Tail such that Tail (‘x:List(‘t))
     is (Inj (IsListRep?(‘t)))(snd(Proj ‘x));
   value Cons such that Cons(‘a, ‘b:List(‘t))
     is (Inj (IsListRep?(‘t))) (‘a,(Proj ‘b));
   value Null? such that Null? ‘L is ‘L = Nil
   comment We use “=” here to compare elements
     of type single. We do not export “=” to
     to avoid comparing functions, when we specify
     lists of functions. end-comment
end-from
```

Figure 1: A Generic *List* Specification

module where the only difference with respect to the old one is in the types of the *Head* and *Tail* operations. the SUB superfunction to specify that *Head* and *Tail* can only operate on non empty lists:

> *Head@SUB(Not_Null?@(List('x:TYPE)→boolean)) → 'x*
> *Tail@SUB(Not_Null?@(List('x:TYPE)→boolean)) → List('x)*

An attempt to define a value such as *Head(Newlist)* results in an execution time "type error" since the new **export** specification of *Head* only operates on a subtype of lists that excludes empty lists. A user of *Generic-List* can easily test the pre condition by using the *Null?* function which operates on all lists. In this case, a list object has a different type than the parameter specification of the new *Tail* function. To perform the *Tail('L)* operation the system performs type conversions as described in Section 4.

5.2 A Generic Binary Search Tree Specification

Figures 2 – 4 specify a generic binary search tree (GBST). Figure 2 is the exported access to a GBST; a user of this specification only has access to the exported functions. We can specify one integer binary search tree type using

> **value** *IntBST @ TYPE* **is** *GBST(integer,>)*

We can build objects of type *IntBST*:

> **value** *NewIntTree @ IntBST* **is** *Clear @ IntBST*
> **value** *SmallTree @ IntBST* **is** *Insert(Insert(NewIntTree,5@integer),7@integer)*

We can use any integer relation to specify an integer binary search tree – we are not restricted to the ">" relation. The *GBST* specification is flexible enough to allow us to construct a tree of lists:

```
value ListTree @ TYPE→TYPE
    such-that ListTree 't is GBST(List 't, ListComparer)
```

A polymorphic *ListComparer* function is used to specify the comparison between lists. For example, a *ListTree* can be defined in terms of a polymorphic comparison function such as "length":

```
value Length @ List('some:TYPE) → integer
    such-that Length 'L is
        if Null? 'L then 0 @ integer
        else 1 @ integer + Length(Tail 'L)
```

Now we can specify *ListComparer* in terms of length:

```
value ListComparer @ List('some:TYPE)×List('some) → boolean
   such-that ListComparer('L1,'L2) is Length('L1) > Length('L2)
```

```
module GBST
export
GBST @ TYPE.'t ×('t× 't→boolean) → TYPE;
Clear @ GBST('t:TYPE, 'f:('t× 't→boolean));
IsEmpty? @ GBST('t:TYPE, 'f:('t× 't→boolean)) → boolean;
Data @ SUB(Not_IsEmpty? @ (GBST('t:TYPE, 'f:('t× 't→boolean))→boolean)) → 't;
InGBST? @ GBST('t:TYPE, 'f:('t× 't→boolean)) × 't → boolean;
Insert @ GBST('t:TYPE, 'f:('t× 't→boolean)) × 't
   → GBST('t:TYPE, 'f:('t× 't→boolean));
Leftside @ SUB(Not_IsEmpty? @ (GBST('t:TYPE, 'f:('t× 't→boolean))→boolean))
   → GBST('t, 'f);
Rightside @ SUB(Not_IsEmpty? @ (GBST('t:TYPE, 'f:('t× 't→boolean))→boolean))
   → GBST('t, 'f));
from
  ⋮
end-from
```

Figure 2: Generic Binary Search Tree Exported Access

The use of a non-polymorphic *ListComparer* function to define a *ListTree* would restrict the types that can be applied to *ListTree* — a type error would result if an invalid type is applied to such a *ListTree*.

Of course we can define a monomorphic (like IntBST) type using GBST; for example:

value *IntListTree @ TYPE* **is** *GBST(List(integer),FooComparer)*

where

value *FooComparer @ List(integer)×List(integer) →boolean*
  such-that *FooComparer('L1,'L2)* **is** *Head('L1) > Head('L2)*

The representation of GBST in Figure 3 consists of a basic structure and an invariant. The basic structure, specified as type constructor *GBTREE*, corresponds to the source set used in the SPECS specification language of [10]. *GBTREE* is specified as the union of the types "empty tree", which can only have the value *$@single*, and "non-empty tree", which contains a value of a generic type and two subtrees. The invariant, *GBST-inv?*, uses the super function SUB to restrict all binary search trees to GBTREE's that satisfy the generic relation.

Figure 4 contains the the specification of *GBST* operations illustrate the definition of functions that return "normal" values.

5.3 Readability of PROSPER Specifications

Unfortunately the example PROSPER specifications are not easy to read. Readability problems are partly because we are explicit with all details here; future systems should infer most

```
module GBST
export ...
from
  value GBST such-that GBST(‘t, ‘f) is SUB (GBST-inv? (‘t, ‘f));
  value GBST-inv? @ (TYPE.‘t×(‘t×‘t→boolean)) → GBTREE(‘t) → boolean
    such-that GBST-inv? (‘t,‘f) ‘bst is
      if BASE-TYPE(‘bst) = single then true
      else let (‘data, ‘left, ‘right) be Proj(‘bst)
      in (biggest? ‘f (‘data, ‘left)) & (smallest? ‘f (‘data , ‘right));
  value biggest? @ ((‘t:TYPE×‘t)→boolean) → (‘t × GBTREE(‘t)) → boolean
    such-that biggest? ‘f (‘elem,‘subtree) is
        comment ‘f is the comparison function end-comment
      if BASE-TYPE(‘subtree) = single then true
      else let (‘data,‘left,‘right) be Proj(‘subtree)
        in (‘f (‘elem, ‘data)) & (biggest? ‘f (‘elem, ‘left))
          & (biggest? ‘f (‘elem, ‘right)) ;
  value smallest? @ ((‘t:TYPE×‘t)→boolean) → (‘t × GBTREE(‘t)) → boolean
    such-that smallest? ‘f (‘elem,‘subtree) is
      if BASE-TYPE(‘subtree) = single then true
      else let (‘data,‘left,‘right) be Proj(‘subtree)
        in (‘f(‘data,‘elem)) & (smallest? ‘f (‘elem, ‘left))
          & (smallest? ‘f (‘elem, ‘right)) ;
  value GBTREE @  TYPE → TYPE such-that
    GBTREE ‘t is SUM(Is-GB-structure? ‘t );
  value Is-GB-structure? @ TYPE → TYPE → boolean
    such-that Is-GB-structure? ‘a ‘b is
      if ‘b = single then true
      else if Cartesian? ‘b then let ‘r × ‘z be ‘b
        in if Cartesian? ‘z then let ‘s × ‘t be ‘z
        in ‘r = ‘a & (Is-GB-structure? ‘a ‘s) & (Is-GB-structure? ‘a ‘t)
        else false else false;
  ⋮
end-from
```

Figure 3: Generic Binary Search Tree Representation

```
module GBST
export
  ⋮
from
  ⋮
  value Clear is Inj (GBST-Inv?(‘t,‘f)) ($ @ single);
  value IsEmpty? such-that IsEmpty? (‘x) is ‘x = Clear;
  value Data such-that Data ‘x is first(Proj(‘x));
  value InGBST? such-that
    InGBST? (‘tree:GBST(‘t:TYPE,‘f), ‘elem) is
     if ‘tree = Clear then false
     else if ‘elem = Data(‘tree) then true
       else if ‘f(‘elem, Data(‘tree))
         then InGBST? (Rightside (‘tree), ‘elem)
         else InGBST? (Leftside (‘tree), ‘elem);
  value Insert such-that
    Insert(‘tree:GBST(‘t:TYPE,‘f),‘elem) is
     if ‘tree = Clear then
     (Inj (GBST-Inv?(‘t,‘f)))(‘elem,$ @ single,$ @ single)
     else if ‘f(‘elem,Data(‘tree))
       then
(Inj (GBST-Inv?(‘t,‘f))) ((Data(‘tree)), Leftside(‘tree), Proj(Insert(Rightside(‘tree),‘elem)))
       else
(Inj GBST-Inv?(‘t,‘f))) ((Data(‘tree), Proj(Insert(Leftside(‘tree),‘elem)), (Rightside(‘tree)));
  value Leftside such-that
    Leftside(‘tree:GBST(‘t,‘f)) is
     (Inj (GBST-Inv?(‘t,‘f)))(first(snd(Proj ‘tree));
  value Rightside such-that
    Rightside(‘tree:GBST(‘t,‘f)) is
     (Inj (GBST-Inv?(‘t,‘f)))(snd(snd(Proj ‘tree));
  ⋮
end-from
```

Figure 4: Generic Binary Search Tree Operations

of the types. The use of parameterized types will always add flexibility and generality at a cost of greater complexity to the human reader anyway. We remind the reader that the examples are of ADT's that are likely to be the core of a specification system, and that ADT's built from the example specification should be easier to read. However, formal specifications do tend to be difficult to read. Preciseness comes at a cost. Balzer has also commented on the difficulty of reading GIST specifications [26], and he describes one solution to the problem — a program that translates a GIST specification into natural language. We do not feel that any executable formal specification will be easy to read, because a formal specification reflects the essential complexity of a system [1]. We suggest that a translator for PROSPER similar to the one designed for GIST specifications will be a useful tool to aid the reading of PROSPER specifications.

6 Related Work

A PROSPER specification models the functionality of a system; it does not model the environment outside the system. Other researchers have developed specification techniques that include the outside environment. Feather calls such specifications "closed-system" specifications [27]. Closed-system specification techniques include the Jackson System Development Method, JSD [28, 29], GIST [27], and PAISLey [30]. JSD is an informal technique and is not executable, however it has been used successfully in industrial settings. GIST specifications are formal and potentially executable, rely on access to an associative database, and are based on acceptable histories of system behavior. GIST was developed from an artificial intelligence and database system perspective and can be used to model concurrent systems. The PAISLey executable specification language was developed to formally specify real-time systems. A PAISLey specification consists of a set of asynchronous processes, and each process is specified in a functional manner. "Exchange functions" (which are not mathematical functions) are used to specify interprocess interactions. PAISLey can be used to specify both real-time constraints and concurrency.

PROSPER is designed to model only the functional aspects of system behavior. Thus, we do not address non-functional aspects such as concurrency and real-time constraints. We seek to explore the potential of dynamic typing in functional specifications using abstract model specification techniques. Although we do not currently address non-functional aspects of specification, abstract model techniques have been applied to concurrency issues [31]. PROSPER is distinguished by its flexible and powerful type specification facilities and its conformance to the abstract model paradigm.

PROSPER is designed to provide an executable means of expressing specifications that can be defined using model based specification languages. Thus the PROSPER design is influenced by model based non-executable languages such as the SPECS language defined by Baker et al. [10], VDM (meta-iv) [13, 9], and Z [14]. These specification languages assume a small set of scalar types such as integers, reals, and booleans, and a small set of type constructors. The structure of data objects and the types of procedures and functions are specified in a manner very similar to that of imperative programming languages such as Pascal and Ada. Additional type restrictions on data objects are expressed using invariants. Both the invariants and the pre and post conditions of operations are specified using first order predicate calculus. Because, in general, first order predicate calculus is not executable, these specifications are not executable and the verification of specifications and implementations must be performed by humans.

A synthesis system that generates Pascal or PL/1 implementations from abstract model specifications was developed by Belkhouche and Urban [32]. The system statically enforces invariants on generic parameters, and must determine all types and restrictions at compile time since Pascal is one of the target languages.

PROSPER differs from the foregoing abstract model specification languages. PROSPER is functional and side effect free. The functional nature of PROSPER eases the algebraic analysis of PROSPER specifications [33]. Besides, a functional language is a natural vehicle for expressing functional specifications. A profound difference between PROSPER and the foregoing abstract model specification languages is that PROSPER has one form for expressing both type structure and invariants – functions. Thus, using a subtype operator, an invariant is merged with the type structure specification. A type restricted with an invariant is distinct from the "same" type without the invariant. Because PROSPER is executable, full first order predicate calculus can not be used and the results of operations must be specified in a constructive fashion.

MIRANDA [7] and *me too* [8] are also executable functional specification languages. Both languages allow the specification of infinite data objects, while PROSPSER does not. Infinite objects are not included in PROSPER because we have "eager" (not "lazy") evaluation of PROSPER expressions. This avoids the problem of undecidable functions on infinite objects in MIRANDA ([7], see pp. 385-386). The most significant difference between PROSPER and either *me too* or MIRANDA is the extensive type specification facilities of PROSPER.

The *me too* language has been implemented in a Lisp environment [34] and has been used to prototype a number of software systems including a decision analysis system [35] and an expert system shell [36]. PROSPER takes a similar approach as *me too* and, in addition, is able to enforce invariants and pre conditions. Such constraints are expressed only as comments in *me too*.

The enforcement of invariants and pre conditions and the desire for generic and polymorphic specifications, resulted in PROSPER's unique treatment of types. Nordström and Petersson suggest using an extended type system to completely specify a program [37]. Using their system both compile-time and run-time type checking are undecidable. Typechecking can only be performed using formal proofs provided by the programmer. The PROSPER type system is not as strong as that of Nordstöm and Petersson, but it is automatable and typechecking is performed at run-time as part of specification testing.

One language with a related treatment of types is Pebble [15]. Pebble and PROSPER differ in their treatment of the concept of "value." PROSPER values are built from symbols using "@", while ":" indicates the type of a parameter or a name. In Pebble, the symbols themselves are values. Both Pebble and PROSPER treat types as values. However Pebble types are values only during the typechecking phase, which is a distinct phase from execution. Such a distinction is not made in PROSPER; the computation of PROSPER type values is as flexible as the computation of normal values. As a result, PROSPER typechecking is limited at compile time — much type checking must be performed at execution time.

In Pebble, bindings and modules are also values. In PROSPER, the concepts of declarations and values are not glued conceptually. Both PROSPER and Pebble types are built from a few primitive types without operations, and more complex types are constructed from the primitives. Neither language deals with assignment, exceptions, or concurrency.

One difference between the two languages is the treatment of polymorphic types. PROSPER polymorphism is implicit (as in ML), while in Pebble the polymorphism is explicit. We illustrate the difference with an example. Consider a generic *swap* function. In Pebble a generic *swap* routine is of type

$(t_1\textit{:type} \times t_2\textit{:type}) \rightarrow> (t_1 \times t_2 \rightarrow t_2 \times t_1)$.

Note the different "$\rightarrow>$" operator for function space construction and also note that t_1*:type* in Pebble is equivalent to TYPE.'t_1 in PROSPER. This *swap* routine can be instantiated as *swap [int,bool]:int*$\times$*bool*$\rightarrow$*bool*$\times$*int.* In PROSPER, a similar *Swap* function is defined:

Swap @ 'a:TYPE$\times$*'b:TYPE*$\rightarrow$*'b*$\times$*'a.*

The PROSPER *Swap* function does not need to be supplied with explicit type arguments. The type of the results depends on the type of the arguments. Thus, *Swap(1 @ integer,false @ boolean):boolean*$\times$*integer.* In PROSPER, there is one notation "." for uniform treatment of dependent types (depending on both types or ordinary values). Thus, the need for special operators such as "$\rightarrow>$" is avoided. We can define (as in Pebble) a super-type TYPE.*'t* $\times$ *'t* which represents pairs of the form: (type, element of that type). In PROSPER we can also define a function *G @ integer.y*$\rightarrow$*F(y)*, where perhaps *F @ integer*$\rightarrow$*TYPE* and *F(1)=boolean*, in which case *G(1):boolean.* A function similar to *G* is not discussed in [15]. Also Pebble does not have the equivalents of the superfunctions SUB and SUM.

7 Conclusions

Specification is a critical phase of software development. The goal of the specification phase is to create a formal description of the system to be designed. The specification itself can mean "to describe properties" or "to build a prototype." In the latter case, it means to define data domains and functions over them. Such specifications can be called "executable specifications" or operational specifications.

PROSPER supports the "specification by prototyping" paradigm and provides powerful facilities for defining executable functional specifications. The language is based on a small set of primitive constructs which allows the construction of flexible generic structures. PROSPER treats types and functions as "first class citizens" and has new mechanisms for defining types that depend upon actual parameters at execution time. The merging of invariants with the structural specifications of data types insures that type invariants are preserved. PROSPER constructs are designed to be fully executable. As a result, a complex software specification that is grown from the primitives is executable.

The PROSPER approach has potential for reducing the cost of software development. Relatively simple (mathematically), the model based system makes the design process easier than in a system based on the axiomatic approach or in a system incorporating imperative features. Our system provides the ability to test design by execution on different levels of the construction process. Thus, PROSPER can be used for rapid prototyping. One can test and/or experiment with a prototype to check the functionality of the system.

Current research activities include the implementation of a prototype PROSPER interpreter, the specification of additional generic types useful in the general application of model based specifications, and using PROSPER to specify non trivial software systems.

References

[1] F. Brooks. No silver bullet – essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[2] W. M. Turski. And no philosophers' stone, either. *Information Processing 86*, 10:1077–1080, 1986.

[3] R. Balzer, Jr. T. E. Cheatham, and C. Green. Software technology in the 1990's: using a new paradigm. *Computer*, 16(11):39–45, November 1983.

[4] P. Zave. The operational versus the conventional approach to software development. *Comunications of the ACM*, 27(2):104–118, February 1984.

[5] S. Fickas, S. Collins, and S. Oliver. *Problem Acquisition in Software Analysis: A Preliminary Study.* Technical Report CIS-TR-87-04, Dept. of Computer and Information Science, University of Oregon, Eugene, Oregon 97403, 1988.

[6] W. L. Scherlis and D. S. Scott. First steps toward inferential programming. *Information Processing 83*, 9:199–212, 1983.

[7] D. A. Turner. Functional programs as executable specifications. *Philosophical Transactions of the Royal Society of London*, A 312(1522):363–388, October 1984.

[8] P. Henderson. Functional programming, formal specification, and rapid prototyping. *IEEE Transactions on Software Engineering*, SE-12(2):241–250, February 1986.

[9] C.B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall International, London, 1980.

[10] A. Baker, J. Bieman, and P. Clites. Implications for formal specifications – results of specifying a software engineering tool. *Proc. of the IEEE Computer Society's Eleventh Annual International Computer Software & Applications Conference (COMPSAC87)*, 131–140, October 1987. Tokyo, Japan.

[11] J. Bieman, A. Baker, P. Clites, D. Gustafson, and A. Melton. A standard representation of imperative language programs for data collection and software measures specification. *The Journal of Systems and Software*, 8(1):13–37, January 1988.

[12] B. Cohen, W. Harwood, and M. Jackson. *The Specification of Complex Systems.* Addison-Wesley, Great Britain, 1986.

[13] D. Bjørner and C. Jones. *Formal Specification and Software Development.* Prentice-Hall, London, 1982.

[14] I. Hayes (editor). *Specification Case Studies.* Prentice-Hall International, London, 1987.

[15] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. *Lecture Notes in Computer Science*, 173:1–50, 1984.

[16] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. *Lecture Notes in Computer Science*, 78, Springer-Verlag, New York, 1979.

[17] A. Demers and J. Donahue. Datatypes, parameters, and typechecking. *Proc. 7th ACM Symposium on Principles of Programming Languages*, 12–23, 1980.

[18] J. Reynolds. Towards a theory of type structure. *Lecture Notes in Computer Science*, 19:408–425, 1974.

[19] J. Goodwin. Why programming environments need dynamic data types. *IEEE Trans. Software Engineering*, SE-7(5):451–457, September 1981.

[20] A. R. Meyer and M. B. Reinhold. Type is not a type. *Proc. 13th ACM Symposium on Principles of Programming Languages*, 287–288, January 1986.

[21] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[22] J. C. Mitchell and R. Harper. The essence of ml. *Proc. 15th ACM Symposium on Principles of Programming Languages*, 28–46, January 1988.

[23] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[24] D. S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–586, September 1976.

[25] J. Leszczyłowski. *Treating types as values.* Technical Report, Institute of Computer Science, Polish Academy of Science, Warsaw, 1988. in preparation.

[26] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.

[27] M. S. Feather. Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems*, 9(2):198–234, April 1987.

[28] M. A. Jackson. *System Development.* Prentis-Hall, Englewood Cliffs, NJ, 1982.

[29] J. R. Cameron. An overview of jsd. *IEEE Transactions on Software Engineering*, SE-12(2):222—240, February 1986.

[30] P. Zave and W. Schell. Salient features of an executable specification language and its environment. *IEEE Transactions on Software Engineering*, SE-12(2):312–325, February 1986.

[31] C.B. Jones. Specification and design of (parallel) programs. *Information Processing 83*, 9:321–332, 1983.

[32] B. Belkhouche and J. Urban. Direct implementation of abstract data types from abstract specifications. *IEEE Transactions on Software Engineering*, SE-12(5):649–661, May 1986.

[33] J.W. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[34] P. Henderson, C. Minkowitz, and J. S. Rowles. me too *Reference Manual.* STC Technology Ltd., Staffordshire, 1987.

[35] C. Minkowitz. *A Formal Design of a Decision Analysis System.* Technical Report TR. 27, Dept. of Computing Science, University of Stirling, Stirling FK94LA Scotland, 1986.

[36] V. Jones, S. Jones, and C. Minkowitz. *A Formal Specification of an Expert System Shell.* Technical Report TR. 20, Dept. of Computing Science, University of Stirling, Stirling FK94LA Scotland, 1985.

[37] B. Nordstrom and K. Petersson. Types and specifications. *Information Processing 83*, 9:915–920, 1983.