Typing Abstract Data Types

Judith Underwood*

February 20, 1995

Abstract

The purpose of this paper is threefold. First, we describe some basic ideas of constructive type theory, with emphasis on their value for specification. Second, we demonstrate the use of type theory as a specification language. This is done by means of a detailed example, namely, the specification of an abstract data type (ADT) for multisets. (This example is a refinement of the multiset example in [3].) Finally, we describe how a theorem proving environment built on type theory can be used to aid in implementation of the ADT.

1 Introduction

This paper describes an approach to the specification and implementation of abstract data types using type theory. A rich language of types is used to express the desired properties of an implementation of an abstract data type in such a way that correctness is enforced by the type discipline. The use of constructive type theory allows the formal development of an implementation by means of an automated proof assistant.

One goal of this paper is to introduce ideas from type theory to the algebraic specification community. To this end, we begin with an introduction to the basic ideas of constructive type theory. We then illustrate the use of these ideas by means of a detailed example, namely, the specification of

*To appear in the proceedings of the 10th Workshop on Algebraic Specifications of Abstract Data Types. This research was supported by EPSRC grant GR/H73103.

an abstract data type for multisets. In the course of the presentation of this example, we shall see how the issues of observational equivalence and parameterization of specifications can be treated in a type theoretic context. Finally, we describe how an implementation of the ADT can be developed in a theorem proving environment based on type theory.

The particular type theory we use is that of Nuprl [4]. In this paper, we will highlight aspects of Nuprl's type theory (not necessarily unique to Nuprl) which we consider useful for the purposes of specification. The implementation comes from the fact that Nuprl has a *constructive* type theory; a proof in Nuprl describes a program. Many other proof systems have been developed on type theoretic foundations [22, 14, 6, 18, 13, 15], and other proof systems [9, 16, 17] can support reasoning in this style. Nuprl's type theory is related to that of Martin-Löf [14].

This paper presents a few aspects of the work in [2], which is a development of ideas in [3]. The original motivation for this work was in the context of using constructive metamathematics to assist in developing tools for theorem provers [5]. Given a metatheorem which states that provability of certain classes of formulas is decidable, a constructive proof of this theorem leads to an implementation of a decision procedure for these formulas. If a theorem proving system is powerful enough to describe its own formulas and proofs, then such a metatheorem may be proven within the system itself. With the addition of a mechanism such as Nuprl's reflection rule [1] for mapping the internal descriptions of formulas and proofs to actual formulas and proofs, we can use the theorem proving system to soundly extend its own store of decision procedures.

This plan is carried out in [2], by the author along with Aitken and Constable. In that paper, we used abstract data types to describe the logic for which we produce a decision procedure. Here, we present one of those datatypes (for multisets) in detail and describe its use and implementation. The ADT is a refinement of one defined in [3]; a number of details have been filled in and clarifications made.

This paper differs from earlier treatments in a number of ways. (In addition to those works already mentioned, the paper of Luo [12], which uses a different type theory, has many of the same goals.) First, we chose to specify the ADT of multisets in a very flexible way. In particular, we include an induction operator, i.e. a general purpose operator for defining functions and predicates on multisets by induction. The fact that models of this specification are (isomorphic to) non-trivial quotients of the term algebra creates special problems for the specification and use of the induction operator. Furthermore, we treat in more detail the issues involved in the development of an implementation. As a part of this treatment, we present a technique for increasing the efficiency of the implementation of the induction operator by using general recursion in a principled way.

One particular difference between the type theory used in this paper and that used by Luo is in the treatment of equality. Equality in Nuprl is *extensional* – functions are equal if they give equal output on equal inputs. We believe this approach is more natural for specification than the intensional equality used in Luo's ECC.

2 Introduction to Type Theory

In this paper, we are interested in using type theory to specify abstract data types. However, it is easier to understand the power of type theory if we consider it in a broader framework; namely, as a foundation for constructive mathematics.

The general principles underlying the constructive approach to type theory are "propositions as types" and "proofs as programs". We assume the existence of some basic mathematical objects, represent these objects as terms, and describe them by giving them types. These fundamental types are called *ground* types. We may also assume we have some axioms about the properties of the elements of these types.

In order to reason about the fundamental types and their elements, we must have a language which describes how more complex formulas can be constructed from ground types, and a logic which describes the valid inferences we can make from the axioms. In a constructive type theory, the formulas are themselves types, and a logical inference is valid only when it can be expressed computationally. In this way, the logic represents a programming language, and a type corresponds to a theorem only when there is a program of that type.

For example, consider the theory of arithmetic over the natural numbers. We can express this by designating a ground type N as that of the natural numbers, and supplying axioms about the behaviour of elements of N, for example, the Peano axioms. Recall that the terms in N are defined by

$$0 \in N$$
$$x \in N \rightarrow s(x) \in N$$

where s denotes the successor function.

Some of the axioms for arithmetic are:

$$\begin{aligned} x + 0 &= x \\ x + s(y) &= s(x + y) \end{aligned}$$

Axioms similar to those for $+$ may be added for any primitive recursive function. In addition, we have an axiom schema for induction which allows us to prove properties of all elements of N:

$$A(0) \land \forall x.(A(x) \rightarrow A(s(x))) \rightarrow \forall x.A(x)$$

Other formulas may be built from the type N using the usual logical connectives: $\land$ (and), $\lor$ (or), $\neg$ (not), $\rightarrow$ (implies), $\forall$ (for all), and $\exists$ (there exists). However, in the inference system, these connectives are interpreted constructively. This means that a proof of a formula is an explicit construction of an element of the type which the formula represents.

As an example, consider implication. If A and B are types, the type $A \rightarrow B$ is the type of functions from A to B. A constructive proof of the formula $A \rightarrow B$ is a function mapping proofs of A to proofs of B; so, the formula $A \rightarrow B$ is provable if there is a function of type $A \rightarrow B$.

As another example, consider disjunction. The type $A \lor B$ is a disjoint union type, and an inhabitant of $A \lor B$ is either an element of A or an element of B. Furthermore, an element of $A \lor B$ is tagged, so that we can decide which of A or B it inhabits. Note that this is a dramatically different approach than that taken in classical logic, where $A \rightarrow B$ is identical to $\neg A \lor B$. No such identity holds when $A \rightarrow B$ is interpreted as a function type and $\neg A \lor B$ is interpreted as a disjoint union. (The type $\neg A$ is considered shorthand for $A \rightarrow Void$, where $Void$ is an empty type.)

Quantifiers also have computational significance. In order to prove $\exists x : A.P(x)$ constructively, we must provide a particular a of type A paired with a proof that $P(a)$ holds. It is insufficient to merely show that assuming that there is no such a leads to a contradiction; such a line of reasoning does not *construct* the a which we want. Similarly, the type $\forall x : A.\exists y : B.P(x, y)$ is provable when we can provide a function mapping elements a of A to elements b of B paired with elements of $P(a,b)$. This idea will be explained in more detail in the next section, along with the means by which a type may depend on an element of another type.

Returning to the axioms given above for arithmetic, we find that the axioms of arithmetic fall into two general categories. The logical axioms have computational content; the equational ones do not. The only non-equational axiom we have given is the induction schema. Its computational representation is a function which takes two arguments: an element of $A(0)$ (representing a proof that $A(0)$ holds) and a function from $x:N$ which returns a function from $A(x)$ to $A(s(x))$. Given these arguments, the function returns a function from N to A. We call such a function an induction operator, which can be used for defining functions by induction on the natural numbers. (We use the term induction instead of recursion in this case because recursion, in general, may be unbounded. The induction operator can only define functions by primitive recursion, though in a higher-order system this does not mean the only functions definable are primitive recursive.)

Some other type theoretic approaches to specification (e.g. [12]) distinguish between types and propositions. Types correspond to the logical axioms, and their proofs have computational content. Propositions represent the equational axioms, and their proofs need not be computationally meaningful at all. In Nuprl, no distinction is made between propositions and types; to supply evidence for noncomputational axioms, they are considered to represent a type with a single element which we will call *yes*. As an example of the treatment , the theorem $\forall x:N.\exists y:N.y = x + s(0)$ has computational content which we write as $\lambda x.\langle s(x), yes$. Given x in N, we show there exists a y such that $y = x + s(0)$ by calculating such a y, namely $s(x)$. The proof that $s(x) = x + s(0)$ has no computational content, so it is written as *yes*, denoting that it has been proven. (In the actual Nuprl system, the *yes* token is called *axiom*.)

In summary, describing mathematics in a constructive type theory allows us to formally relate proofs and programs. Given a constructive proof of a formula, we may automatically extract a program of the type corresponding to the formula, representing the computational content of the proof. In other words, if we consider the formula as a specification, we can generate a program meeting the specification from a proof of the formula.

Given this general approach, there are still many options available in the design of such a constructive type theory. Some of these relate to aspects of the logic corresponding to the types; others involve the associated computation system. For sufficiently strong type theories (in particular, any which can describe first order arithmetic), provability is undecidable by Gödel's theorem. Because of this, in implemented systems proof search is generally interactive, with the user of the system applying rules either directly or through tactics which apply a sequence of rules. Besides Nuprl, such systems include [6, 10, 18, 13, 15].

2.1 Useful Types and Type Constructors

Here we describe some desirable properties for a type system if it is to be used in the context of specification. We present a number of different types and type constructors, along with an explanation of their computational significance. All of these types are present in Nuprl.

Built in types. For a type system to be usable as a specification language for programs, we would like it to supply a reasonable number of built in ground types and type constructors, along with tools for reasoning about them. In this paper, we assume we have a ground type *bool* for Boolean values, the built in type N, and a built in type constructor *list*, which, given a type A, describes the type of lists of elements of A. We assume that among the axioms for *list* types is an induction principle, so that in the computation system, we have a corresponding induction operator.

Dependent product types. The dependent product type is the type theoretic representation of existential quantification. If A and B are types, then $x : A \times B$ is a type. If x is free in B, then B represents a family of types parameterized by elements of A. An element of $x : A \times B$ is a pair $\langle a, b \rangle$, where $a \in A$ and $b \in B[a/x]$. Note that this is evidence for the logical assertion $\exists x : A.B$. Note also that if x is not actually free in B, this is just a Cartesian product type. In this case, we may just write $A \times B$, omitting the variable binding. (The dependent product type is sometimes denoted $\Sigma x : A.B$; when this is the case, it is referred to as a Σ type or a dependent sum, taking the view that it is an abbreviation for the infinite disjoint union of all $B[x]$ as x ranges over A.) This type will be most useful when x ranges over types rather than elements of a type.

Dependent function types. The dependent function type is the type theoretic representation of universal quantification. If A and B are types, then $x : A \rightarrow B$ is a type. Elements of this type are functions which, given an argument $a \in A$, return an element of $B[a/x]$. An example is a function which, given a natural number n, returns the n-tuple $\langle 1, \ldots, n \rangle$. Thus, the type of the result of function application can depend on the argument to the function. Note that such a function is evidence for the logical assertion $\forall x : A.B$. If the variable x does not actually appear in B, this is a simple function type, and may be written $A \rightarrow B$. In the remainder of this paper, we may use both $\forall x : A.B$ and $x : A \rightarrow B$ notations, depending on whether we wish to emphasize the logical or the computational aspect of the type. (The dependent function type is sometimes denoted $\Pi x : A.B$, and referred to as a Π type or a dependent product. This terminology arises from the classical view of functions as sets of pairs of elements from the domain and range.)

Universe types. To express types parameterized by other types, we would like to have a "type of types" over which the parameter can range. In Nuprl, this is achieved by the use of universe types. Nuprl includes a hierarchy of universe types (to prevent foundational problems with the type of all types), so that types built from ground types only are in universe type U_1, and U_{k+1} includes types built from types in U_k plus the type U_k itself. Where it is not necessary to specify the precise universe level, we use U_i. Note that the presence of both type universes and dependent types allows a form of quantification over types (limited by the universe hierarchy), resulting in types parameterized by other types. This idea will prove very useful for introducing abstractness in a specification.

Set types. It is often convenient to describe an object as an element of a certain type satisfying some extra conditions. To describe this, we use a *set* (sometimes called *subset*) type. The type of all elements of T satisfying property P is denoted $\{t : T \mid P(t)\}$. This could be called a subtype; however, that word has somewhat different connotations in type theory.

The types and type constructors listed above provide a very flexible and powerful language for specification. The use of dependent types together with universes allows higher order types and reasoning. For example, functions may take types as arguments and return types as values. This technique can be used to represent parameterized specifications. We will return to this idea once we have seen how dependent types are used in the example in the next section.

There are a few aspects of Nuprl's type theory which distinguish it from others (e.g. ECC [11]). One such feature lies in its treatment of equality. All types come equipped with an equality predicate and rules for reasoning about equality in the type. The notation $a = a' \in T$ is used for assertions about equality to specify the type involved. The notation $a \in T$ is shorthand for $a = a \in T$, which simply asserts a is of type T. For the typing assumptions of variables, we use the notation $x : T$.

An important point about Nuprl's equality (in contrast to that used by ECC) is that it is extensional; that is, functions are equal if they produce the same output on equal inputs. Extensional equality is well suited to the purpose of implementation of specifications, since specifications are concerned with the *behaviour* of functions rather than the details of their implementation. Furthermore, given an implementation of a specification in this kind of theory, we can replace a function with another function equal to it in its type and be assured that all the properties required of it still hold.

3 Multisets: An Example

We now show how to use the machinery of type theory to describe an abstract data type. A *multiset* over a type A is a collection of elements of A in which a given element may appear multiple times, but in which there is no ordering of the elements. We will give an abstract data type for finite multisets over A. There are a number of things we require from the ADT. First, it should be parameterized by the type A, so that it can be used to implement multisets over a number of different types. Second, we need a carrier set of multisets, and tools for constructing multisets. Third, in order to provide a general mechanism for defining functions on multisets, we provide a multiset induction operator, i.e. a tool for defining functions and predicates by induction. This is a departure from the usual algebraic specification for multisets; we use this operator to avoid having to specify in advance exactly which functions on multisets we will want. This flexibility comes at the price of increased complexity of the specification, since the specification of an induction operator is the induction principle for multisets and this specification requires higher-order logic. Furthermore, since multisets are not a simple term algebra, specifying the behaviour of the induction operator is not straightforward. Finally, we need axioms which ensure that all the operators defined in the ADT have the properties we want them to have.

One of these properties is functionality, in the sense that any function defined on multisets must map equal multisets to equal values. This is not trivial, because equality on multisets may not be the same as equality in the type used to implement multisets. This is the well-known issue of *behavioural equivalence* [7, 8, 19, 20]; namely, ensuring that there is no way to tell that the type used to implement multisets is not a true multiset type. We do this by defining equality on multisets explicitly as part of the ADT, and ensuring that the types of the multiset operations are such that the defined equality is respected.

We now describe in more detail the specification of the ADT. In order to parameterize the multiset data type by the type of elements A, we must have a "type of types" over which A can range. This is precisely what we have called a "universe type". For increased flexibility, we also take a parameter which is an equality decider on A, rather than assuming we wish to use the built in equality associated with A. This allows us to define multisets of multisets. For simplicity of notation, we define, for any type S,

$$EqDecider(S) \equiv \{f : S \rightarrow S \rightarrow bool \mid \lambda x, y. f(x)(y) \text{ is an equivalence on } S\}$$

(It is straightforward to describe the conditions for f to be an equivalence relation.) So, in addition to the type A, we take a parameter Aeq of type $EqDecider(A)$.

Given these parameters, we describe the abstract data type of multisets over A. This is written as a product of the types of various elements of the ADT: the carrier set, the construction and induction operators, and the axioms. We begin by using a universe type as the type of the carrier set M of multisets. Using a dependent product type, we describe the rest of the types in the product in terms of the types A and M. So, given M as the type of multisets, the type of multiset equality Eq is $EqDecider(M)$. This is intended to represent abstract multiset equality; we will ensure it does so by specifying an axiom for its behaviour later.

The construction operators we use are a nullary operator $Empty$, representing the empty multiset, and a function $Add(a,m)$, which adds the element a of type A to the multiset m. We wish to ensure that Add is functional with respect to Eq. We define $AddType$ to be

$$\begin{aligned} AddType \equiv \quad & \{ f : A \rightarrow M \rightarrow M \\ & \quad \mid \forall a, a' : A, m, m' : M. \\ & \qquad Aeq(a)(a') = true \Rightarrow Eq(m)(m') = true \\ & \qquad\quad \Rightarrow Eq(f(a)(m), f(a')(m')) = true \\ & \} \end{aligned}$$

This definition merely formalizes the requirement that functions map equal elements of A (equal under Aeq) and equal elements of M (equal under Eq) to equal elements of M. Although we have curried the arguments to Add in specifying its type, we will use the notation $Add(a,m)$ to avoid proliferation of parentheses.

We now consider the type of the multiset induction operator $MInd$. The idea is that this represents the computational content of the multiset induction principle. Given a type constructor P parameterized by multisets, the induction principle gives us a means by which to produce an element of the type $P(x)$ for any multiset x. To do so, we must provide an object of type $P(Empty)$, where $Empty$ is the empty multiset. We must also provide a function f that maps a multiset m, an element a, and an object of type $P(m)$, to an object of type $P(Add(a,m))$. The intuition is that we begin with the object of type $P(Empty)$, and use f repeatedly, once for each element of x, to build up an object of type $P(x)$.

There are a number of requirements on the type of the multiset induction operator. Specifically, these are requirements on the types of the arguments to the operator; when these are satisfied (along with axioms introduced later for the behaviour of $MInd$), functions defined by induction will be guaranteed to be functional with respect to the equality defined on multisets. To express the restrictions on types more concisely, we introduce some notation.

For any type constructor $T(a,m)$ parameterized by an element a of A and a multiset m, we define $a : A {\mapsto} m : M {\mapsto} T(a,m)$ to be the type of functions that map an element a of A and a multiset m to $T(a,m)$, and that are functional in the appropriate fashion.

$$
\begin{aligned}
a\!:\!A\mapsto m\!:\!M\mapsto T(a,m) \equiv \quad & \{f\!:\!(a\!:\!A \rightarrow m\!:\!M \rightarrow T(a,m)) \\
& \qquad \mid \forall a, a'\!:\!A, m, m'\!:\!M. \\
& \qquad\quad Aeq(a)(a') = true \Rightarrow Eq(m)(m') = true \\
& \qquad\qquad \Rightarrow (f(a)(m) = f(a')(m') \in T(a,m)) \\
& \}
\end{aligned}
$$

We specialize this notation to include

$$M\mapsto T \equiv \{f\!:\!M \rightarrow T \mid \forall m, m'\!:\!Eq(m)(m') = true \Rightarrow f(m) = f(m') \in T\}.$$

As before, this describes, for any type T, the type $M\mapsto T$ which contains all the functions from M to T that are functional with respect to the equivalence relation induced by Eq.

With this notation, we may describe the type of $MInd$. First, the type constructor P must be functional with respect to multiset equality. So, the type of P is $M\mapsto U_i$. Recall that this means $P(m)$ is a *type*; the induction operator $MInd$ is used to produce members of that type.

The next argument to $MInd$ is the value of the function being defined on the empty multiset; therefore, the type of this argument is $P(Empty)$. After this, we have a function f which, given an element a of type A and a multiset m of type M, produces a function from $P(m)$ to $P(Add(a,m))$ which describes how to compute an element in $P(Add(a,m))$ from a,m, and an element of $P(m)$. Since we require f to be functional with respect to the defined equalities on types A and M, the type of f is partly described by $a : A\mapsto m : M\mapsto(P(m)\rightarrow P(Add(a,m)))$. However, we have an additional restriction on f, namely that the function it produces (i.e. the element of $P(m)\rightarrow P(Add(a,m))$) be insensitive to the order in which m is constructed. We can describe this in the type of f by using the set type to restrict f to those elements f of $a : A\mapsto m : M\mapsto(P(m)\rightarrow P(Add(a,m)))$ which have this property. We examine the need for this restriction further in the next section. Formally, we define

$$
\begin{aligned}
& FInd(A, Aeq, M, Eq, Add, P) \equiv \\
& \quad \{f\!:\!(a\!:\!A\mapsto m\!:\!M\mapsto(P(m) \rightarrow P(Add(a,m)))) \mid \\
& \qquad \forall a, b\!:\!A, m\!:\!M, p\!:\!P(m). \\
& \qquad\quad f(a)(Add(b,m))(f(b)(m)(p)) = f(b)(Add(a,m))(f(a)(m)(p)) \\
& \qquad\qquad \in P(Add(a, Add(b,m)))\}.
\end{aligned}
$$

We shall abbreviate $FInd(A, Aeq, M, Eq, Add, P)$ as $FInd(\ldots)$.

The result of applying the $MInd$ operator to arguments P, $e\!:\!P(Empty)$, and $f\!:\!FInd(\ldots)$ is a function from $m\!:\!M$ to $P(m)$. Thus, the final type of $MInd$ is

$$P\!:\!(M\mapsto U_i) \to e\!:\!P(Empty) \to f\!:\!FInd(\ldots) \to m\!:\!M \to P(m).$$

The axioms which describe the desired behaviour of the operators on multisets are also expressed as types, although we are only concerned that these types be inhabited. There are four axioms. The first is a restriction on the Add operator, requiring it to produce equal multisets regardless of the order of the addition of elements. It is sufficient to require that for any pair of elements a and a' and multiset m, the result of adding a and then a' to m is equal to the result of adding a' and then a. This is expressed as the axiom:

$$\begin{array}{l} Axiom1 \equiv \\ \quad \forall a, a'\!:\!A, m\!:\!M.Eq(Add(a, Add(a', m)))(Add(a', Add(a, m))) = true. \end{array}$$

Rather than introducing this as an axiom, we could have included this as part of the type of Add.

The other axioms are guarantees on the behaviour of $MInd$.

$$\begin{array}{l} Axiom2 \equiv \\ \quad \forall P\!:\!M\mapsto U_i, e\!:\!P(Empty), f\!:\!FInd(\ldots), m, m'\!:\!M. \\ \quad\quad Eq(m)(m') = true \Rightarrow MInd(P)(e)(f)(m) = MInd(P)(e)(f)(m') \\ \quad\quad\quad \in P(m) \\ Axiom3 \equiv \\ \quad \forall P\!:\!M\mapsto U_i, e\!:\!P(Empty), f\!:\!FInd(\ldots). \\ \quad\quad MInd(P)(e)(f)(Empty) = e \in P(Empty) \\ Axiom4 \equiv \\ \quad \forall P\!:\!M\mapsto U_i, e\!:\!P(Empty), f\!:\!FInd(\ldots), a\!:\!A, m\!:\!M. \\ \quad\quad MInd(P)(e)(f)(Add(a, m)) = f(a)(m)(MInd(P)(e)(f)(m)) \\ \quad\quad\quad \in P(Add(a, m)) \end{array}$$

The first of these axioms ensures that functions on multisets defined by induction are actually functions, i.e. they respect multiset equality. This seems obvious, but it does not follow from the type of $MInd$ alone, since the type in this case does not carry any restrictions on the behaviour of $MInd$. The remaining two axioms are similar, in that they guarantee that the functions produced by $MInd$ actually do what we expect them to do, given the arguments to $MInd$; the value of the function on $Empty$ is the value given to $MInd$, and the value of the inductive case is computed by f.

The result is the definition:

$$
\begin{aligned}
& \mathit{MULTISET} \equiv \\
& \qquad A : U_i \\
& \quad \to Aeq : EqDecider(A) \\
& \quad \to M : U_i \\
& \qquad \times\ Eq : EqDecider(M) \\
& \qquad \times\ Empty : M \\
& \qquad \times\ Add : AddType \\
& \qquad \times\ MInd : \\
& \qquad\quad P : (M \mapsto U_i) \to e : P(Empty) \to f : FInd(\ldots) \to m : M \to P(m) \\
& \qquad \times\ Axiom1 \\
& \qquad \times\ Axiom2 \\
& \qquad \times\ Axiom3 \\
& \qquad \times\ Axiom4
\end{aligned}
$$

An interesting aspect of this description of multisets is that the types of the operators and the axioms which describe their behaviour are expressed in the same language. However, we have provided this specification with some structure by separating the "local" requirements for the behaviour of operators (expressed in their types) from the "interactive" behaviour (expressed in the axioms). Clearly there is not a strict distinction between these two ideas; the first axiom could have been expressed instead as a restriction on the type of the *Add* operator. However, such separation can be conceptually helpful, as in the separation between the type of the $MInd$ operator and the list of its properties.

Before we examine the use of the ADT of multisets, a couple of points should be noted. First, the treatment of parameterization should be highlighted. Dependent types have been used to describe parameterization of the ADT by the choice of the carrier type for multisets. In fact, it appears that dependent types can be used to describe the two kinds of parameterization distinguished in [21], namely, the specification of a parameterized object and the parameterization of the specification itself. As written, the multiset ADT specification is of the first kind; any implementation must describe the other components of the specification *uniformly* with respect to the choice of a carrier type. This is because we have used a dependent function type. However, if we had instead used a dependent product type to describe the dependency of the implementation on the choice of a carrier type, we would have avoided the need for a uniform implementation.

Second, and more generally, the definition of the ADT seems very complex compared with an algebraic approach. This is due to two factors. First is the decision to include an induction operator; the specification of such a general operator is more complex than that of a simple function in any system. Second, because we have explicitly specified our intended equality on multisets, we must ensure in the specification that other parts of the ADT respect that equality. In an algebraic approach, this issue is separated from the specification and must be dealt with by reasoning about the behaviour of an implementation or a model. Since the type theoretic treatment effectively includes such reasoning, it is more complex.

3.1 Examples of the use of the ADT.

Perhaps the most unusual aspect of this ADT is the $MInd$ operator. Because of this, and because of the complexity of its type, it may be illuminating to examine some examples of its use.

First, we define a function $Size$ which computes the size of a multiset. The type of this function is $M \rightarrow N$. The first argument (P) to $MInd$ is a function describing how the type of the result of $MInd$ depends on the argument m in M it is given. Since in this case the type of the result does not depend on m, the first argument to $MInd$ is simply $\lambda x : M.N$. Since this function does not depend on the element m it is given, it is clearly in $M \mapsto U_i$.

The value of $Size$ on the empty multiset is zero, so this is the second argument to $MInd$.

The third argument to $MInd$ is a function f from arguments a of type A, m of type M, and p of type N to N, which describes how to compute $Size(Add(a, m))$ from a, m, and p, the value of $Size(m)$. This function is simply $\lambda a.\lambda m.\lambda p.p + 1$. It is clear that this function respects the equalities on A and M. Furthermore, it is easy to see that

$$f(a)(Add(b, m))(f(b)(m)(p)) = f(b)(Add(a, m))(f(a)(m)(p))$$

since both sides evaluate to $p + 2$.

Applying $MInd$ to these arguments, we can now define

$$Size = MInd(\lambda x.N, 0, \lambda a.\lambda m.\lambda p.p + 1).$$

Note that this is of type $m\!:\!M \rightarrow P(m)$, or, given $P = \lambda x.N$, simply $M \rightarrow N$, as desired.

Suppose we wanted to define a function from multisets to lists. According to our understanding of the restrictions on the types involved, this should be very difficult, since the functions defined by induction must map equal multisets to arguments equal in the result type, in this case, lists. A straightforward mapping of $Add(a_1, \ldots (Add(a_n, Empty)))$ to $(a_1, \ldots, a_n)$ should not be definable by multiset induction.

To see that it is not definable, let us attempt to define it and see where the attempt fails. The type of the resulting function is simply $M \rightarrow A\ list$, so the first argument to $MInd$ will be $\lambda x.A\ list$. The value of the map on the empty multiset should be *nil,* the empty list, so this is the second argument.

The third argument to $MInd$ is a function describing how to build a list for $Add(a, m)$ from a, m, and the list resulting from m. Our candidate for this function is $\lambda a.\lambda m.\lambda p.cons(a, p)$. This function respects equality on both A and M (the latter because the result does not depend on m directly), but it fails the extra condition that

$$f(a)(Add(b, m))(f(b)(m)(p)) = f(b)(Add(a, m))(f(a)(m)(p)).$$

4 Implementation

In this section, we demonstrate the use of Nuprl as a tool for implementing an ADT and proving that the implementation is correct. We show how the multiset example of the previous section can be implemented using Nuprl's built in list type, how the resulting implementation may be used, and how the implementation can be made more efficient without sacrificing correctness.

Recall that an implementation of the multiset ADT is a function which, given a type A and Aeq (the equality on A) returns a tuple consisting of a type M representing the carrier set for multisets of elements of A, an equivalence relation Eq on that type, a distinguished element $Empty$ of M representing the empty multiset, an operator Add for adding elements to a multiset, and a multiset induction operator $MInd$, along with evidence that the axioms listed in the previous section hold for the types and operators supplied.

Given the types A and Aeq, we choose the Nuprl type $A\ list$ as the carrier set. We must supply an equivalence relation Eq, distinct from list equality, which identifies lists which differ only in the order of their elements. Otherwise, this instantiation is straightforward. The empty multiset is implemented as the empty list; multiset element addition is implemented using $cons$; multiset induction is implemented by list induction.

List induction in Nuprl is performed by a built-in induction form $list\_ind$. A function f from type $A\ list$ to T is defined by

$$\lambda l.list\_ind(l, b, \lambda h \lambda t \lambda v.e)$$

where l is the argument to the function, $b\!:\!T$ is the value of the function on the empty list, and e is an expression describing how to construct the value of $f(cons(h,t))$ given h, t, and $v = f(t)$ (i.e. v is the value of the function on the tail of the list). The $list\_ind$ form is evaluated by "unrolling" the induction, so that

$$\begin{array}{rcl} list\_ind(nil, b, \lambda h \lambda t \lambda v.e) & \longrightarrow & b \\ list\_ind(cons(a,l), b, \lambda h \lambda t \lambda v.e) & \longrightarrow & e[a/h, l/t, list\_ind(l, b, \lambda h \lambda t \lambda v.e)/v] \end{array}$$

Given these choices for the implementation of the components of the ADT, evidence for the various axioms is most easily constructed by proving them true in Nuprl for the list type and its operations. This is fairly straightforward, since we have proof rules describing the type and its operators. The evidence for these axioms has no computational content, so we do not express it in detail.

The implementation is then:

$$
\begin{array}{l}
Multiset \equiv \lambda A.\lambda Aeq.\langle \\
\quad A\ list \\
\quad \lambda x.\lambda y.is\_permutation(x, y, Aeq) \\
\quad nil, \\
\quad \lambda a.\lambda m.cons(a, m), \\
\quad \lambda P.\lambda hb.\lambda hi.\lambda m.list\_ind(m, hb, \lambda u\lambda v\lambda w.hi(u)(v)(w)), \\
\quad \lambda a.\lambda b.\lambda m.\mathit{(evidence\ for\ axiom\ 1)} \\
\quad \lambda P.\lambda hb.\lambda hi.\lambda m.\lambda n.\lambda x.\mathit{(evidence\ for\ axiom\ 2)} \\
\quad \lambda P.\lambda hb.\lambda hi.\mathit{(evidence\ for\ axiom\ 3)} \\
\quad \lambda P.\lambda hb.\lambda hi.\lambda a.\lambda m.axiom\mathit{(evidence\ for\ axiom\ 4)} \\
\rangle
\end{array}
$$

Here, $is\_permutation(x, a, Aeq)$ returns the boolean true value if x and y are permutations of each other, and returns false otherwise. The function Aeq is used to decide equality between elements of x and y. The implementation of $is\_permutation$ is a straightforward application of Nuprl's $list\_ind$ operator.

Another implementation possible in Nuprl is through Nuprl's quotient types. We can define the type of multisets by taking the quotient of $A\ list$ by the equivalence relation $is\_permutation(x, y, Aeq)$. The notation for this is $(x, y) : A\ list//is\_permutation(x, y, Aeq)$. The leading (x, y) serves as a binding operator for the variables representing the arguments to $is\_permutation$. Elements of $(x, y) : A\ list//is\_permutation(x, y, Aeq)$ are elements of $A\ list$, and two elements t and t' are equal in the type if $is\_permutation(t, t', Aeq)$ holds. So, Eq is the standard equality on this type. With this implementation, the conditions ensuring that the various operators on multisets are functional become trivial. Multiset induction can still be implemented by list induction.

4.1 Efficient Induction

In the implementation given above, we use the built in $list\_ind$ induction form to implement multiset induction. While this is correct, it can be inefficient; evaluation of a function f defined by $list\_ind(l, b, \lambda h\lambda t\lambda v.e)$ requires that, on each unwinding of the definition, we evaluate the head h and tail t of the list, when we may not use this information at all. For example, if we merely wish to count the number of items in the list, e is merely $v+1$, and we use neither h nor t. Furthermore, if we wish to compute the value of of f by dividing the list into two parts and evaluating the function on each part, we cannot express this naturally in terms of h, t, and v. The induction operator follows strictly the form of the inductive definition of lists, and when a function is more naturally defined by another induction, *list_ind* can be cumbersome and inefficient.

Because of this, we may wish to replace the induction form with a more flexible and more efficient recursive operator. To do this, we prove a more general induction principle for well founded orderings defined by a measure on lists.

The induction lemma we prove is

$$
\begin{array}{l}
\forall A\!:\!U_i. \\
\quad \forall P\!:\!A\ list.\!\rightarrow\! U_i. \\
\quad\quad \forall f\!:\!A\ list\!\rightarrow\! N(\forall l\!:\!A\ list. \\
\quad\quad\quad (\forall l'\!:\!\{l'\!:\!A\ list|f(l') < f(l)\}.P(l'))\Rightarrow P(l)) \\
\quad\quad\quad\quad \Rightarrow(\forall l\!:\!A\ list.P(l))
\end{array}
$$

The use of the set type $\{l'\!:\!A\ list|f(l') < f(l)\}$ guarantees that the computational content of the proof that the inductive measure of l' is less than the measure of l is not used in the extraction. The set type effectively prevents this by hiding the proof that $f(l') < f(l)$. This means that the numerical induction measures need not be explicitly calculated at run time as long as they were shown to have been reduced when this lemma was used. In other words, the proof that the measure is reduced is required once, while the program is being developed; the induction measure need not be checked at each stage of the evaluation.

The induction principle above may be proved in Nuprl by supplying an explicit witness for its truth. In this case, the witness is

$$\lambda A.\lambda P.\lambda f.\lambda g.(Y \lambda h.\lambda x.(gxh))$$

where $Y = \lambda f.(\lambda x.f(x(x)))(\lambda x.f(x(x)))$ is the usual fixpoint combinator. This term is shown to be an inhabitant of the appropriate type using Nuprl's computation rules, which allow the Y combinator to be "unrolled" in order to use the inductive hypothesis.

The result of this is a more flexible and efficient induction operator for lists, which can also be used for multiset induction. The only significant change in the use of this operator is that one must explicitly supply the inductive measure as a function to natural numbers, and show that it is reduced at each stage. The tradeoff here is then between complexity of the proof using this induction and the efficiency of the algorithm extracted from the proof.

5 Conclusions

We have seen how constructive type theory can be used to specify an abstract data type and guide its implementation. The advantages of this approach lie in the expressiveness of type theory as a specification language and in its close relationship to implementation. This expressiveness includes a uniform language for types and statements about behaviour, as well as a representation of parameterization and higher order logic. Flexibility extends to the specification of an induction operator, which allows users of the ADT to write their own functions. Finally, an automated proof assistant for constructive type theory can be used to guide the implementation of the ADT and verify its correctness.

Thanks to Don Sannella for helpful comments, and Bill Aitken and Bob Constable for previous collaboration.

References

[1] W. Aitken. *Metaprogramming in Nuprl Using Reflection.* PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, 1994.

[2] William Aitken, Robert Constable, and Judith Underwood. Metalogical frameworks II: Using reflected decision procedures. unpublished manuscript.

[3] D. Basin and R. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, chapter 1, pages 1–29. Cambridge University Press, Great Britain, 1993.

[4] R. L. Constable et al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[5] R. L. Constable and D. J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R.B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 45–76. Elsevier Science Publishers (North-Holland), 1990.

[6] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[7] V. Giarratana, F. Gimona, and U. Montanari. Observability concepts in abstract data type specification. In *Proc. 1976 Symposium on Mathematical Foundations of Computer Science*, pages 567–578. Springer LNCS 45, 1976.

[8] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In *Proc. 9th Intl. Colloquium on Automata, Languages, and Programming*, pages 265–281. Springer LNCS 140, 1982.

[9] M. Gordon. HOL: A machine oriented formalization of higher order logic. Technical Report 68, Cambridge University, 1985.

[10] Christian Horn. The nurprl proof development system. Technical report, University of Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[11] Z. Luo. ECC: An extended calculus of constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.

[12] Z. Luo. Program specification and data refinement in type theory. In *Proceedings of the Fourth International Joint Conference on the Theory and Practice of Software Development (TAPSOFT)*, 1991.

[13] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[14] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.

[15] Bengt Nordström. The ALF proof editor. In *TYPES'93: Proceedings of the Fourth Annual Workshop of the EU Esprit Basic Research Action "Types for Proofs and Programs"*, number 806 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[16] S. Owre, J.M. Rushby, and N. Shankar. PVS: An integrated approach to specification and verification. Preprint, January, 1992.

[17] L.C. Paulson. A formulation of the simple theory of types (for Isabelle). In P. Martin-Löf and G. Mints, editors, *Proc. of the Int. Conference on Computer Logic, Lecture Notes in Computer Science, Vol. 417*, pages 246–274. Springer-Verlag, New York, December, 1988.

[18] R. Pollack. *The Theory of Lego.* PhD thesis, University of Edinburgh, 1994. Forthcoming as a technical report.

[19] D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[20] D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: model theoretic foundations. In *Proc. 19th Intl. Colloquium on Automata, Languages, and Programming*, pages 656–671. Springer LNCS 623, 1992.

[21] Donald Sannella, Stefan Sokolowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: parameterisation revisited. *Acta Informatica*, 29:689–739, 1992.

[22] D.T. van Daalen. *The Language Theory of AUTOMATH.* PhD thesis, Technical University of Eindhoven, Eindhoven, Netherlands, 1980.