Design, Verification and Certified Implementation of Abstract Data Types
Rodrigo Cardoso
e-mail: rcardoso@uniandes.edu.co
Universidad de los Andes
Bogota, D.C., Colombia

Abstract

ManTa is a design, verification and certified implementation tool for equational Abstract Data Types (ADTs). In the design phase, ManTa helps the user in syntactic and semantic aspects, like definition of total functions, and allows an easy construction of ADTs which are built in terms of other already defined ADTs by means of products, direct sums, restrictions and enrichments. ManTa includes a theorem prover specially oriented to prove algebraic properties of ADTs, which can be used to validate abstract implementations. ManTa produces automatically prototypes (ANSI C implementations) for ADTs, which may be transformed manually into other which preserve correctness. Finally, the performance of the prototypes may be observed on an automatically generated execution environment, where the user can create, manipulate and evaluate objects of the implemented types. As a software product, ManTa was developed on a free software platform (GNU emacs, gcc). It is a very portable system for which it is easy to develop other interfaces according to the actual resources of an installation.

Keywords

Formal specification, abstract data types, software engineering, certified implementation, automatic code generation, automatic theorem proving, rewriting systems, rapid prototyping.
1 INTRODUCTION

Abstract Data Types (ADTs) are widely used as a modeling and programming tool which allows a separated treatment of specification and implementation of data structures in a software system cf. [Gut78], [Bau81], [Aho83], [Car91]). This separation of concerns makes easier the verification, implementation and reusability of software. There are languages and environments which manipulate concrete implementations of ADTs, so old like Alphard [Sha77], CLU [Lis77], AFFIRM [Mus80], among others.

Traditionally, ADTs are used in an informal way: though an ADT has been understood as a data structure abstract description, aspects like consistence of the underlined algebraic structure or the correctness of an implementation versus the specification are seldom formally verified. On the other way, most commercial languages allow the implementation of ADTs in some extent; however, one could say that this becomes a real advantage only if good specification and programming methodologies are followed.

A general ADT definition covers heterogeneous algebraic structures (i.e. sets of different types with operations between them) with properties which may be established, for instance, by first order logic formulae. A more constrained notion, but very comfortable to be constructively implemented on a machine, allows different kind of finitely generated models, with totally defined operations (functions), through equational first order axioms. We called that an equational ADT. ManTa is a tool for the formal treatment of equational ADTs. For here on we will omit the term "equational", except if we want to emphasize this property.

Section 2 introduces basic concepts on ADTs. Section 3 presents a panoramic view of ManTa and its facilities: formal definition of equational ADTs, theorem-proving, abstract and concrete implementation and its verification, and interface features. Section 4 includes conclusions and suggests future work.

2 BASIC CONCEPTS

The ManTa ADT definition is based on some given primitive types, e.g., Bool, Nat, etc., from which complex types are constructed. As an algebraic structure, an ADT has an underlined set, named the type of interest (toi) and some other already defined types, with operations or
functions which manipulate them. For primitive types, the toi and the operations are assumed known.

In order to define a complex ADT Y, we assume as known a set of parameter ADTs \( X = \{ x_1, \ldots, x_n \} \). Y-objects (i.e., the toi, also named Y) are the finitely generated terms from initial operations (like \( i : D_1 \rightarrow Y \), and \( D_1 \) a Cartesian product of sets in \( X \)) and constructors (operations like \( c : D_c \times Y^m \rightarrow Y \), with \( D_c \) a Cartesian product of sets in \( X \)). ADT Y may include selector operations (like \( s : D_s \times Y^m \rightarrow E \), where \( D_s \) a Cartesian product of sets in \( X \) and \( E \) is an element of \( X \cup \{ Y \} \)). The set of operations of an ADT is called its signature.

Initial and constructor operations are called generators. Their semantics reduce to denote, with the from them finitely generated terms, the elements of Y. Selectors' semantics is defined through equational first order axioms which determine a unique value for every possible element in the operation domain. In some cases this value could be undefined (denoted with \( \perp \)), but the axioms must explicit establish this fact. Equations are considered oriented, so that the left side should be interpreted as defined by the right side. Left sides always refer to terms which outermost operation symbol corresponds to a selector operation. The set of axioms whose left sides refer to the same operation symbol constitute a (primitively recursive) functional definition of the corresponding operation.

The following example illustrates concepts and notation:

**ADT Queue[X]**

* emp: \( \rightarrow \) Queue

* ins: Queue x X \( \rightarrow \) Queue

isv: Queue \( \rightarrow \) Bool

fst: Queue \( \rightarrow \) X

rst: Queue \( \rightarrow \) Queue

siz: Queue \( \rightarrow \) Nat

eqc: Queue x Queue \( \rightarrow \) Bool

**Axioms**

isv(emp) = true

isv(ins(c,x)) = false

fst(emp) = \( \perp \)

fst(ins(c,x)) = if isv(c) then x else fst(c) fi

rst(emp) = emp

rst(ins(c,x)) = if isv(c) then emp else ins(rst(c),x) fi

siz(c) = if isv(c) then 0 else siz(rst(c)) + 1 fi

eqc(c1,emp) = isv(c1)

eqc(emp,ins(c2,x)) = false

eqc(ins(c1,x),ins(c2,y)) = (x =_X y \land eqc(c1,c2))
ADT Queue depends on a parameter type x. Its signature is the set \{emp, ins, isv, fst, rst, siz, eqc\}, from which \{emp, ins\} is the set of generators and \{isv, fst, rst, siz, eqc\} is the set of selectors. Axioms define selectors' semantics. The function eqc is an equality operator for queues. In this case, equality corresponds to syntactic identity.

A definition like that of ADT Queue is called inductive. Other ways to define ADTs, which always can be explained as shortcuts of inductive definitions are:

- **Instantiation**: \(ADT Y \equiv W[A]\), where \(W[X]\) and \(A\) are defined ADTs.
- **Products**: \(ADT Y \equiv ADT W \times ADT Z\)
  The toi is the Cartesian product of those of the components ADTs \(W\) and \(Z\).
- **Sum**: \(ADT Y \equiv ADT W + ADT Z\)
  The toi is the disjoint union of those of the components ADTs \(W\) and \(Z\).
- **Restriction**: \(ADT Y \equiv (ADT W : p)\)
  The toi is the set of elements of \(W\) that satisfy the predicate \(p\). This must be defined (with axioms) in terms of \(W\)-operations.

In any case the operations of the basic ADTs are extended in a "natural" way to the new type (cf. 3.1), though sometimes this extensions could be not quite useful.

Since the functions' semantics is given through primitively recursive definitions on the toi, many equational theorems (equations with universally quantified variables) can be proved using methods like:

- **Rewriting**: "change equals by equals". Axioms can be used as rewriting rules on the sides of an equation to be proved. If both sides are rewritten to syntactically identical terms, the equation must be true.
- **Structural induction**: induction on the complexity of the construction of the toi (e.g., proving first for initial objects and then, assuming the result for the constituents, then proving for complex objects).

Implementation of these method usually results in incomplete algorithms. However, they can be improved with sound techniques tailored to the problem class to be solved. For instance, techniques like generalization, case analysis, etc. may substantially improve the power of an equational-reasoning oriented theorem prover (cf. [Boy79]).

**ADT Y** may be implemented on another, say **ADT W**. This involves representing **Y**-objects with **W**-objects and **Y**-operations with **W**-operations which simulate them, i.e., **W**-operations that -under
the representation—satisfy the axioms of ADT \( w \). An implementation may be correct or not, and the verification of this fact amounts to the truth of a conjunction of equational theorems on the ADT \( w \). This the main reason to have an equational theorem prover in an environment to construct and test ADTs. Moreover, the resulting theorems are usually very simple and technically easy to prove only with rewriting and structural induction.

Finally, an ADT may be implemented on a computational structure (CS), i.e., the set of available objects and operations in a programming language. We may consider a CS as a "concrete type", though its semantics is not described with equational axioms, but with another well-defined mechanism, like an interpreter (operational semantics), or state assertions (à la Hoare or Dijkstra).

3 MANTA: A TOOL FOR ADT DEVELOPMENT

ManTa is a tool to define, verify, manipulate and implement equational ADTs. The following sections include more detailed descriptions of the provided services. The current version is written in ANSI C, running under Unix. Text editing can be done with a standard editor like emacs or vi.

A full automatic prototyping of not generic ADTs (ADTs independent of variable parameters) is provided. Corresponding c programs are generated, and the user may manually refine these rapid prototypes, changing the representation or the actual implementation of some functions in order to improve the performance.

3.1 ADT Definition

Inductive definition is the basic way to construct new ADTs in ManTa. In this case, a new ADT must be denoted with an identifier and its signature (classifying generators and selectors) and axioms (to specify selectors) must be given. On defining a selector's signature, the user may specify induction arguments, useful to guide the function definition\(^1\) and theorem proving related to the function.

\(^1\) ManTa supplies left sides for axioms which cover the corresponding inductive cases. For instance, if inADT Queue \( X \) the signature of \( \text{fst} \) is defined as "\( \text{fst}: \text{Queue}(I) \rightarrow X \)" where "\( I \)" indicates an induction argument, ManTa asks the user to complete axioms with left sides "\( \text{fst}(\text{emp}) = \ldots \)" and "\( \text{fst}(\text{ins}(q,x)) = \ldots \)", i.e., to establish the function value in order to achieve an inductive total definition.
For each selector, ManTa verifies that the given axioms cover every possible concrete instantiation of the function's arguments. So, every function is total, though this means that "undefined" (⊥) is an acceptable value, i.e., result sets are considered extended to include this value. The current version cannot by itself warrant that for each domain argument the defined value is well-defined and unique.

Primitive types to begin user defined constructions are ADT Nat and ADT Bool. These are unmodifiable system definitions.

ManTa allows definitional shortcuts, which can be viewed as abbreviations of inductive defined ADTs. As mentioned in Section 1, definition of instantiations, products, sums and restrictions are supported. In each case, the new type inherits from its constituents the functions extended to the new domain and can be enlarged with function definitions which could add interrelations between the constituents.

3.2 Theorem proving

The correctness of a representation amounts to prove the truth of a conjunction of equations over the implementing ADT. ManTa's theorem prover is specially oriented to verify abstract implementations of ADTs. Restricted to this work, the prover has a very good performance, since one can expect that the user defines types and representations in a simple and comprehensible way, what results in simple and comprehensible proofs. These mechanical proofs are error-free and avoid lots of clerical work.

A more powerful prover could be used to verify algebraic equational properties of ADTs, which could be useful to ensure that some definition is sound with respect to the modeled reality. For instance, one may easily establish equational axioms to define addition and multiplication of natural numbers. ManTa's theorem prover is unable to prove associativity of these functions with only rewriting and induction, but it can do it using techniques of case analysis and generalization.

In order to prove an equation of the form

$$M[w] = N[w]$$

the theorem prover follows a proof protocol, which resembles, in some extent, that of the Boyer-Moore theorem prover [Boy79]:
- **Rewriting**: Reduce both sides of (1) to normal forms, in the sense of rewriting rules systems\(^2\). The resulting equation is of the form:

\[
M^* [w] = N^* [w]
\]  

If both sides are identical, equality holds. It is known from universal algebra theory that these theorems are just the valid formulæ (which hold in every model of the ADT).

If reduced sides are not identical, try to prove equation (2):

- **Structural induction**: from the user given information on inductive definition of the selectors, prove the equation instantiated, substituting the corresponding arguments for initial objects and then, assuming the truth of the equation for structurally simpler cases, prove the equation instantiated substituting arguments for constructors. Algebraically speaking, the so provable theorems are those which are truth in an initial model of the ADT.

If structural induction fails:

- **Semantic proof**: ManTa requires for each ADT \( Y \) an user defined equality function \( \text{eq}_Y: Y \times Y \rightarrow \text{Bool} \). Of course, it is assumed that this is a true equality function (an algebraic congruence, i.e., an equivalence relation which supports some logical substitution axioms). Then, if normalizing \( \text{eq}_Y(M^* [w], N^* [w]) \) results in a Boolean value, (1) holds if and only if this value is true. This method is complete for ground equations (which do not include variables), but could fail in more general cases.

If semantic proof fails:

- **Case analysis**: if (1) includes an expression of the form \( \text{eq}_Z(t_1, t_2) \), where \( t_1 \) and \( t_2 \) are irreducible terms without variables of the toi, and if \( \mu \) is a most general unifier for \( t_1, t_2 \):
  i. Prove \( \mu(M^* [w]) = \mu(N^* [w]) \) assuming that \( \text{eq}_Z(t_1, t_2) = \text{true} \)
  ii. Prove \( \mu(M^* [w]) = \mu(N^* [w]) \) assuming that \( \text{eq}_Z(t_1, t_2) = \text{false} \).

If both cases succeed, (1) holds. This method is useful to demonstrate conditional assertions that involve external elements to the toi, sometimes generic types without a concrete or known structural construction. The annotated cases correspond to suppose that this equalities hold or not.

If case analysis fails:

\(^2\) Here is needed that rewriting finishes and that normal forms are unique. Of course, these features depend on the well definition of the involved functions.
• **Generalization:** If a term \( t \) appears more than once in (1) and \( v \) is a fresh variable, try to prove

\[ M^*[w]_v = N^*[w]_v \]

where \( \exp^a_b \) is the result of a substitution of the term \( a \) for the term \( b \) along the expression \( \exp \). If this proof succeeds, (1) holds (indeed, a more general result holds).

Failing of this last procedure must be understood as "ManTa could not prove the result".

### 3.3 Abstract implementation

Formal specification with ADTs allows the implementation of an ADT \( Y \), called the source type, in terms of another ADT \( W \), called the object type. A subset of the object type is mapped onto the source type with a representation (partial) function \( \Phi: W \rightarrow Y \). On the other side, \( Y \)-selectors must be implemented with (compositions of) \( W \) functions, in such a way that the semantics, regarded through the representation function \( \Phi \), is preserved.

For instance, a ManTa implementation of \( \text{ADT}^4 \text{ Stack}[X] \) in \( \text{ADT} \text{ Queue}[X] \):

\begin{verbatim}
ImpADT Stack[X] <- Queue[X]: \Phi
  empty = \Phi(emp)
  push(\Phi(q),x) = \Phi(ins(q,x))
  isempty(\Phi(q)) = isv(q)
  top(\Phi(q)) = lst(q)
  pop(\Phi(q)) = \Phi(del_lst(q))

Auxiliary Operations
  lst: Queue \rightarrow X
  del_lst: Queue \rightarrow Queue

Auxiliary Axioms
  lst(emp) = \perp
  lst(ins(q,x)) = x
  del_lst(emp) = emp
  del_lst(ins(q,x)) = q

TDA\text{pmI}
\end{verbatim}

This notation allows to implicitly define the meaning of the representation function using an uniform way to describe how source functions are represented in terms of object functions. The definition could include new auxiliary object functions which must be defined and enlarge the expressive power of the object type without affecting its semantics.

---

3 \( \Phi \) is an onto partial function if it is explained how \( Y \) finitely generated terms are represented by \( W \) terms.

4 ADT \text{ Stack}[X] definition is not given here, but it is supposed that the reader can supply the details.
An implementation should be verified, in the sense that the axioms of the source type must be satisfied by the corresponding object terms. ManTa can be used to carry out the whole verification of an implementation. This task is accomplished by means of the theorem prover.

For instance, one could wonder if the axiom
\[
pop(push(s,x)) = s
\]
is satisfied. Since \( \Phi \) is onto, there exists a \( q \in \text{Queue} \), such that the equation is true if
\[
pop(push(\Phi(q),x)) = \Phi(q)
\]
which can be rewritten in terms of the implementation functions as:
\[
\Phi(del_lst(ins(q,x)) = \Phi(q).
\]
Therefore, the axiom would be satisfied if
\[
del_lst(ins(q,x) = q
\]
and this can be proved true by normalization.

Finally, ManTa allows implementation composition: this means data refinement in the best sense, since semantics is always maintained and object types are supposed more and more concrete. It is important to remark that if there is an interpreter for the object type, it is easy to build an interpreter for the source type, using the representation as a translator between the types. In other words, implementing an ADT into another is the same as building an implementation of the source type on an abstract machine that can interpret source terms as object terms. This fact may be used to experimentally confirm how sound an ADT specification could be.

3.4 Concrete implementations

At the limit stage of the implementation process, it is expected to arrive to the computational structure \( cs \) of a desired programming language. This last step, though conceptually the same as the others, differs in the sense that the object type \( cs \) is not formally defined, or at least its semantics is not defined in the same way than that of the real ADTs. On the contrary, \( cs \)'s semantics is usually operationally established or, for imperative programming languages, one may have an axiomatic semantics à la Hoare.

ManTa is able to build ADTs' implementations, called prototypes, on the computational structure of ANSI C. Prototypes are modular, reusable and portable. Their generation is automatic, but the user may change the definitions of some selected functions (for instance, the generators, which implicitly define the representation), to improve implementation's performance where the user could consider important to do it.
ManTa employs GNU `gcc` and `emacs` for code generation and modification. The use of these development tools facilitates diagnosis and bug fixing (in case of manual transformation). The automatically generated code is complemented with a partial documentation (comment templates with basic information of author and date of generation) to be completed by the user.

As for the composition of abstract implementations, ManTa can compose an abstract specification with a concrete one, to achieve a rapid prototype of a source ADT via an intermediate ADT already concrete implemented. For instance, the in 3.3 described abstract implementation of `ADT Stack[X]` into `ADT Queue[X]` may be composed (previous instantiation of `X` with `Nat`) with a concrete implementation of `ADT Queue[Nat]`, giving as result a concrete implementation of `ADT Stack[Nat]`.

3.5 ADT's Workbench

Though formal verification of concrete implementations cannot be accomplished in the same way as in the abstract case, ManTa provides an experimental tool, called the ADT's `Workbench`. This is a menu-driven program which allows the user to

- select ADT's implemented operations to be tested
- create and modify ADT's implemented objects
- save running environments which contain ADT's implemented objects
- visualize the effect of ADT's operations.

The Workbench is automatic generated, too. Some output features may be manual modified by the user, like the desired way in which ADT's implemented objects must be displayed.

3.6 User interfaces

The user can chose between two interfaces to interact with ManTa:

- a "plane" one, where actions are chosen from a list, corresponding effects are immediately (as text) displayed and then one may chose a next action to be performed, and
- an `emacs`- menu-driven interface, with some advantages in text editing and displaying menus and results in different windows.

Both interfaces were developed to increase software portability, using standard free software (GNU `gcc` and `emacs`) under Unix. Their existence is a clear evidence of the loose coupling
between interface and kernel modules of ManTa. Therefore it is rather easy to develop other interfaces, depending on installation features or user convenience.

4 CONCLUSIONS

ManTa is a software tool for designing, verifying and certified implementing of ADTs. The following features should be remarked:

- Assistance in ADT's definitions, suggesting the form that necessary axioms for function definitions should have and checking if every function is total defined (included user controlled error cases).
- User is forced to be aware of error cases which he/she must control.
- Through definitional variants, ADTs are defined in a short, understandable and always rigorous way.
- The theorem prover helps in soundness checking of the pretended abstract model.
- Abstract implementations into ADTs for which there exist an interpreter (abstract or concrete) allow experimental observation of source ADT's behavior, using programs like the Workbench of 3.5.
- Implementation phase of equational ADTs may be automatically accomplished and refined, preserving correctness.
- Automatic generated prototypes are portable and compatible with by any other means produced ANSI C code. They include partial documentation to be completed by the user.
- System installation allows easy interface tuning (text editor, axioms syntax, language for message display and menu options).

Some aspects to be considered in the future:

- Primitive types Bool and Nat should be integrated to rewriting and theorem-proving (e.g., with built-in rewriting). The current version forces, for instance, that proofs which involve arithmetic must use Peano's representation of numbers (e.g., Suc(Suc(Suc(Zero)))) instead of 3).
- Workbench idea could be easily extended to a more abstract level.
- Implementation of ADT's genericity should be managed at a higher level. At the present generic types are implemented on C with variables and functions of type void. A more stringent type representation for genericity would help in type error detection in manual modification.
Manual prototype modification demands from the user a rather detailed knowledge of the way ManTa generates its own prototypes. Though this seems to be hard to avoid, it would be nice to have software tools for this work.

Acknowledgments
I want to acknowledge César Muñoz for his work in a preliminary Spanish version of this paper.

BIBLIOGRAPHY


[Apo95] Aponte, H., ManTa 3.0, Un sistema para la implantación certificada y automática de tipos abstractos de datos, Tesis de Magister, Departamento de Sistemas y Computación, Universidad de los Andes, Bogotá, 1995.


