Generating Abstractors from Abstraction Functions[1]

Lin Huang David Alex Lamb

December 16, 1992
External Technical Report
ISSN-0836-0227-
92-344

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Version 1.1
Document prepared December 16, 1992

Abstract

Values of an abstract data type (ADT) can be built by some functions of the type called constructors. A construction term of a value is an expression which contains only constructors and whose evaluation yields the value. The abstractor of an ADT is a function that takes a value as an input and produces the corresponding construction term as an output. Abstractors may be used in communicating ADT values in distributed programs.

For a given implementation of an ADT, the abstraction function maps values from the concrete representation (in the implementation) to some abstract representation. So far, abstraction functions have been mainly used in verifying the correctness of implementations.

This paper, in contrast to the current use of abstraction functions, explores a novel role they play in abstractor generation. It describes a notation for specifying abstraction functions and presents a simple method for transforming abstraction functions into abstractors.

[1]This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and in part by the Information Technology Research Center of Ontario (ITRC), part of the Ontario Centres of Excellence program.

Contents

List of Figures

1 Introduction

An **abstract data type** (ADT) is characterized by a collection of sorts (types) and a collection of functions. A subset of the functions is called **constructors**, in the sense that values of the type can be built by calls on them. A construction term of a value is an expression which contains only constructors and whose evaluation yields the value.

The **abstractor** of an ADT takes a value as an input and produces a construction term of the value as an output. One application of abstractors is in systems which use construction terms as exchange representations to communicate ADT values in distributed programs [HL92]. In such systems, ADT values are exchanged in their construction terms; they must be converted into their corresponding construction terms before being transmitted.

In this paper, we are interested in the problem of generating abstractors. Basically, abstractors can be generated from specifications or from implementations:

- In the method of generation from specifications, an abstractor is derived from the specification of a type. The resulting abstractor is implementation-independent, and, therefore, is applicable to any implementation of that type.
- On the other hand, in the method of generation from implementations, the abstractor is derived from the relationship between the type and the **implementation type** that implements it, and the abstractor is only applicable to that particular implementation.

Generation from specifications appears more desirable than generation from implementations, since abstractors generated from specifications are implementation-independent. However, specifications may not always provide the right functions for generating such an abstractor; sometimes, particularities of an implementation need to be considered. The two methods are complementary.

Our earlier paper [HL92] focused on generation from specifications. This paper explores generation from implementations. We shall show how to generate the abstractor for a given implementation from a user-provided abstraction function — a mapping from the implementation type to the ADT.

The organization of this paper is as follows. Section 2 introduces background information, including specification and implementation of abstract data types, and the application of abstractors in communicating ADT values in distribute programs. Section 3 presents a definition of abstraction functions and describes a notation for writing them. Section 4 gives a definition of abstractors, discusses general strategies for generating abstractors, and shows how to generate abstractors from abstraction functions. Section 5 summarizes the results of this paper. The appendices include the syntax of the notation for writing abstraction functions, example abstraction functions for some common implementations of several types, and specifications of types *Array*, *Record*, *Set*, and *Binary_tree*.

2 Background

This section describes the notion of (algebraic) specification and implementation of abstract data types, and discusses the application of abstractors in communicating ADT values in distributed programs. Those familiar with abstract data types may go directly to Subsection 2.3.

2.1 Specification of Abstract Data Types

There is a large literature on algebraic specification methods [GH78, GTW78, EM85, Wir90]. Here we briefly illustrate the basic idea using an algebraic specification for the well-known type *Stack*. A specification defines a type through the functions that are to be available to others. As shown in Figure 1, it consists of seven clauses:

Specification describing the type being specified, that is, the *T*ype *O*f *I*nterest (TOI). Here the TOI is *Stack*.

Parameters describing those items which must be instantiated in declarations. "$P : S$" (S is a set) means P must be instantiated with an element of S. In Figure 1, for example, "$Ele : Type$" indicates Ele is a parameter which must be instantiated with a type.

This clause is optional.

Declaration describing the format in which variables are declared to be of the type. Parameters, if any, must be instantiated in variable declarations. The declaration format for *Stack* variables is $Stack[Ele]$, where the parameter Ele must be instantiated with a type. As an example, one may declare s to be of a stack of integer by

$$s : Stack[Int]$$

Base_types describing the types on which the TOI is based. *Stack* is based on three types: *Boolean*, *Ele* (a parameterized type), and *Nat* (the natural number type).

Functions describing the functions of the type, including the function symbols, the domains and ranges. *Stack* has six functions: *new*, *isnew*, *push*, *pop*, *top*, and *size*.

Constructors describing a subset of the functions which can be used to construct each value of the type. The range of a constructor must be the TOI. The constructors of *Stack* are *new* and *push*.

Equations describing a set of relations that defines the semantics of the functions. For example,

$$top(push(s, e)) = e$$

means applying the function *top* to the result of $push(s, e)$ returns the value of the element just pushed onto the stack.

2.2 Implementation of Abstract Data Types

An ADT is implemented using another type called **implementation type**. Implementing an ADT is to represent its values in terms of values of the implementation type and define its functions in terms of the implementation type's functions.

Figure 2 gives an implementation of *Stack* using implementation type *Record* with fields $buf : Array$ and $ptr : Nat$, where *Record* and *Array*[1] have the same meaning as their counterparts in imperative languages like Pascal, but their operations are functional. For an array A, $fetch(A, i)$ returns the element with index i, and $store(A, i, e)$ returns a new array which is equal to A except the element with index i is e. Similarly, for a record R, $fetch.f(R)$ returns field f, and $store.f(R, e)$ returns a new record which is equal to R except its field f has value e.

In the implementation, a stack is represented by an array which stores its items and and a natural number which points to the position of the top item in the array; the abstraction function *absF_Stack* relates concrete stacks to abstract stacks (see Section 3 for the details of abstraction functions); each function of *Stack* is defined in terms of functions of *Record*, *Array* and *Nat*.

[1] Appendix C gives a formal definition of *Record* and *Array*.

Specification

Stack

Parameters

Ele:	*Type*

Declaration

Stack[Ele]

Base_types

Boolean, Ele, Nat

Functions

new:	$\rightarrow$ *Stack*
isnew:	*Stack* $\rightarrow$ *Boolean*
push:	*Stack* $\times$ *Ele* $\rightarrow$ *Stack*
pop:	*Stack* $\rightarrow$ *Stack*
top:	*Stack* $\rightarrow$ *Ele*
size:	*Stack* $\rightarrow$ *Nat*

Constructors

new, push

Equations

top(push(s, e))	= *e*
pop(push(s, e))	= *s*
isnew(new)	= *true*
isnew(push(s, e))	= *false*
size(new)	= *0*
size(push(s, e))	= *size(s)+1*

Figure 1: Specification of *Stack*

Implementation

Stack

Representation

Stack = Record[buf:Array[Ele], ptr:Nat]

Abstraction function

absF_Stack(r) = abs1(fetch.buf(r), fetch.ptr(r))
where
abs1(a) = **if** *n=0* **then** *new* **else** *push(abs1(a,n-1), fetch(a,n))* **end**
end

Definitions

new = *store.ptr(store.buf(newRec, newArray), 0)*
push(s, e) = *store.buf(s1, store(fetch.buf(s1), fetch.ptr(s1), e))*
where *s1=store.ptr(s, fetch.ptr(s)+1)* **end**
top(s) = *fetch(fetch.buf(s), fetch.ptr(s))*
pop(s) = *store.ptr(s, fetch.ptr(s)-1)*
isnew(s) = **if** *fetch.ptr(s)=0* **then** *true* **else** *false* **end**
size(s) = *fetch.ptr(s)*

Figure 2: Implementation of *Stack*

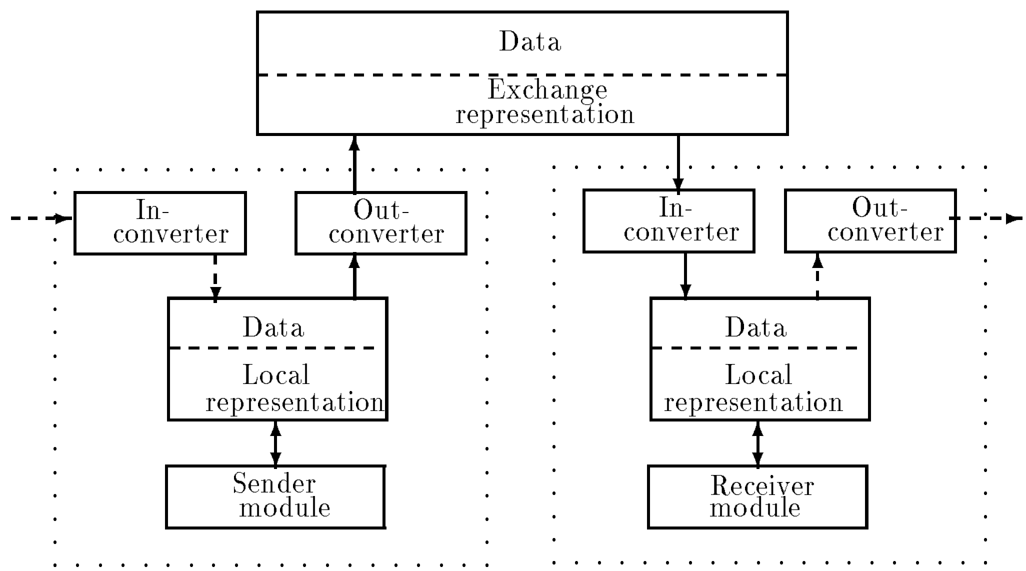


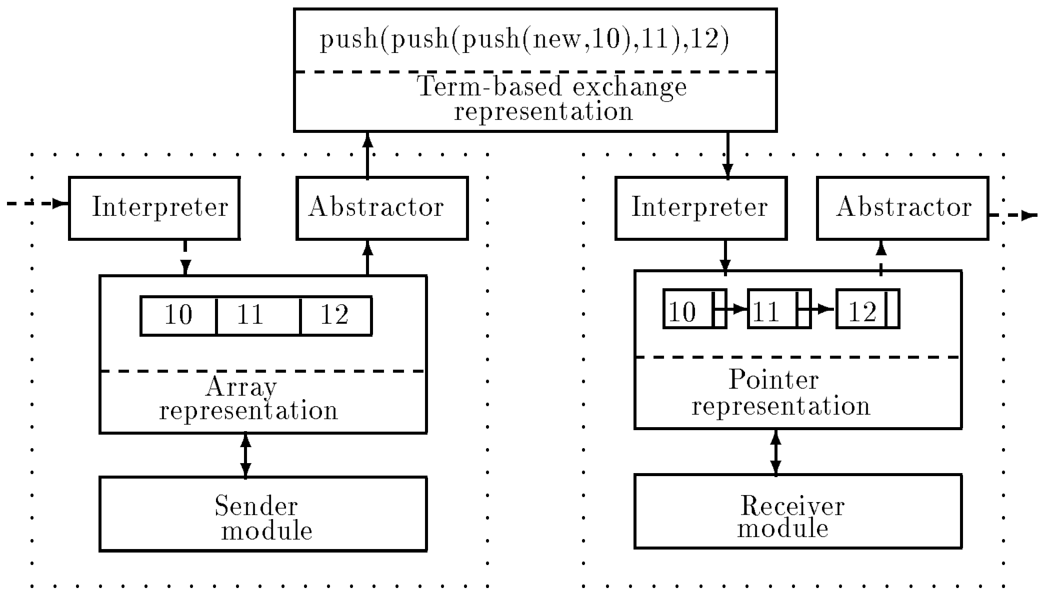
Figure 3: The general communication process

2.3 Communicating Abstract Data Type Values in Heterogeneous Distributed Programs

A distributed program is **heterogeneous** if its modules run on different kinds of machines, use different programming languages, and/or use different implementations for the same abstract data type. In a heterogeneous program, different data representations may be present at its modules. This prevents data from being directly transmitted from one module to another; data conversions are needed somewhere in the process of a communication. One solution is to choose an exchange representation[2] that is acceptable to all the modules of a program and to equip each module with an in-converter and an out-converter (See Figure 3). In this approach, a communication involves three steps, as illustrated by the solid arrows in Figure 3 : the out-converter at the sender transforms the communicated value from the sender's local representation to the exchange representation; the underlying network system transmits the value in the exchange representation from the sender to the receiver; and the in-converter at the receiver transforms the value from the exchange representation into the receiver's local representation.

Particularly, our approach is based on the fact that values of an abstract data type may be built by some of its functions called constructors. A construction term of a value is an expression which contains only constructors and whose evaluations yields the value. We have chosen construction terms as the exchange representation. Accordingly, in-converters can be

[2]Exchange representation is the way to represent values during transmission.



Term-based

Figure 4: The process of communicating a stack

implemented by interpreters of terms and out-converters by abstractors (See Section 4.1 for the definition of abstractors). In a communication, the communicated value is transformed into its construction term by the abstractor at the sender and the construction term is then transmitted to the receiver. The interpreter at the receiver parses the term and produces the local representation by invoking the constructors in the term in an appropriate order.

Let us look at an example. Suppose type *Stack* is implemented by an array at the sender and by a linked structure at the receiver. Figure 4 shows the process of communicating a stack of three elements: 10, 11 and 12, with 10 at the bottom and 12 at the top. The stack (an array) is transformed into its construction term $push(push(push(new, 10), 11), 12)$, then the construction term is transmitted to the receiver, and the local representation (a linked structure) at the receiver is obtained by invoking the operations in the construction term.

Two immediate advantages of this approach are

- The exchange representation is a mathematical notation — independent of any machine, language or implementation — and so is particularly suitable for communication in heterogeneous distributed programs.

- Interpreters are just simple parsers and thus can be easily generated.

This paper studies the problem of generating abstractors from abstraction functions.

3 Abstraction Functions

This section describes the notion of abstraction functions and presents a notation for specifying abstraction functions.

3.1 Abstraction Functions

Suppose T' is an implementation type of type T. A value of T' is called a **concrete value** of T if it represents a value of T. Values of T are called **abstract values**.

To prove the implementation satisfies the specification of T, one needs a link between T' and T. This is usually characterized by a so-called **abstraction function**[3][Hoa72, Sha81, LG86, PM90, Par90][4],

$$absF\_T : T' \longrightarrow T$$

which maps concrete values to abstract values.

Two methods, descriptive or constructive, could be used to define abstraction functions.

In the descriptive method, usually a mathematical theory is chosen to be the model of T. $absF\_T$ is declaratively defined by a mapping from concrete values to objects of the theory. In this method, for the implementation of *Stack* shown in Figure 2, one may choose the mathematical concept sequences to be the model of stacks and then define the abstraction function as a mapping from an array to a sequence by[5]

A typical stack is a sequence $\langle e_1 , \ldots , e_n \rangle$
$absF\_Stack : Record \longrightarrow Sequence$
$absF\_Stack(r) = \langle fetch(fetch.buf(r), 1) , \ldots , fetch(fetch.buf(r), fetch.ptr(r)) \rangle$

In the constructive method, on the other hand, $absF\_T$ is explicitly defined by functions of T and T'. The abstraction function in Figure 2 is defined using this method. For comparison, we present it here again:

```
absF_Stack(r: Record[buf:Array[Ele], ptr:Nat]) = abs1(fetch.buf(r), fetch.ptr(r))
where
    abs1(a:Array[Int], n:Nat) =
        if n=0 then
            new
        else
            push(abs1(a,n-1), fetch(a,n))
        end
end
```

where comparison "=" as in "$n = 0$" and subtraction "$-$" as in "$n - 1$" are functions of type Nat.

Hoare[Hoa72] and Shaw[Sha81] use the descriptive method to define abstraction functions; Partsch[Par90] uses the constructive method; and Liskov and Guttag[LG86] mainly use the descriptive method, though they touch on the constructive method once; Parnas and Madey[PM90] do not mention how to describe abstraction functions.

[3] It is called representation function in references[Hoa72, Sha81].

[4] Details of how to prove the correctness of ADT implementations through abstraction functions can be found in references[Hoa72, Sha81, LG86, Par90].

[5] In conventional notations, this would be $absF\_Stack(r) = \langle r.buf[1] , \ldots , r.buf[r.ptr] \rangle$.

3.2 A Notation for Defining Abstraction Functions

This subsection describes our notation for writing abstraction functions. It is based on the constructive method. Appendix A gives a formal description of its syntax.

In our notation, a function f — an abstraction function or an auxiliary function (see Subsubsection 3.2.3 for details about auxiliary functions) — is defined by

$$f(x_1 : T_1, \ldots, x_n : T_n) = FunctionDef \ , \ n \geq 0$$

where $T_1, \ldots, T_n$ are types, and $FunctionDef$ is either a composition definition or a conditional definition, which is described below.

3.2.1 Composition Definition

$FunctionDef$ is a composition definition if it has the form

$$f'(a_1, \ldots, a_m) \ , \ m \geq 0$$

where f' and a_1 , ..., a_m may be the function on the left hand side (recursive definition), constructors of the type in question, functions of the implementation type, or auxiliary functions.

The abstraction function $absF\_Stack$ in Figure 2, for example, is defined by a composition definition consisting of $abs1$ — an auxiliary function, and $fetch.buf$ and $fetch.ptr$ — functions of the implementation type $Record$.

The notation does not allow non-constructors of the type in question. This is due to our purpose of generating abstractors from abstraction functions, which will be explained in Section 4.

3.2.2 Conditional Definition

A function may be defined by a collection of several different function definitions, each associated with a condition. This is called a conditional definition, formed by keywords **if**, **elsif**, and **else**, as shown below:

if B_1 **then**
    f_1
elsif B_2 **then**
    f_2
    ⋮
elsif B_n **then**
    f_n
else
    f_{n+1}
end

where $B_1, \ldots, B_n$ are Boolean functions, $f_1, \ldots, f_{n+1}$ are function definitions, and "elsif" and "else" clauses are optional.

As an example, $abs1$ in $absF\_Stack$ is defined by a conditional definition, where two conditions, $n = 0$ and $n \neq 0$, are used.

3.2.3 Auxiliary Definition

A function definition with k ($k \geq 1$) auxiliary functions has the form

$$
\begin{array}{l}
f(x_1 : T_1, \ldots, x_n : T_n) = FunctionDef \\
\textbf{where} \\
\quad aux_1(x_{1,1} : T_{1,1}, \ldots, x_{1,n_1} : T_{1,n_1}) = FunctionDef_1 \\
\textbf{and} \\
\quad aux_2(x_{2,1} : T_{2,1}, \ldots, x_{2,n_2} : T_{2,n_2}) = FunctionDef_2 \\
\quad \vdots \\
\textbf{and} \\
\quad aux_k(x_{k,1} : T_{k,1}, \ldots, x_{k,n_k} : T_{k,n_k}) = FunctionDef_k \\
\textbf{end}
\end{array}
$$

where aux_1 , ..., aux_k must appear in $FunctionDef$.

Auxiliary functions are defined in the same notation. $FunctionDef_i$ ($1 \leq i \leq k$) is either a composition definition or a conditional definition. It may even contain its own auxiliary functions; thus auxiliary definitions can be nested.

Using auxiliary functions, one may improve clarity of definitions and may easily define some functions which are otherwise difficult to define.

For example, instead of defining *push* by

```
push(s:Stack, e:Ele) = store.buf(store.ptr(s, fetch.ptr(s)+1),
                                 store(fetch.buf(store.ptr(s, fetch.ptr(s)+1)),
                                       fetch.ptr(store.ptr(s, fetch.ptr(s)+1)),
                                       e
                                      )
                                )
```

one may define it by

```
push(s:Stack, e:Ele) = store.buf(s1, store(fetch.buf(s1), fetch.ptr(s1), e))
     where s1 = store.ptr(s, fetch.ptr(s)+1)
```

which is much clearer.

As another example, consider types *Set* and *Bin_tree*, shown in Figure 10 at page 23 and Figure 11 at page 24 respectively. Suppose *Set* is implemented by *Bin_tree*, where each node of a tree stores one element of a set. If function *union* is allowed to be used, the abstraction function could be defined by

```
absF_Set(t:Bin_tree) =
     if isnew(t) then
          new
     else
          insert(union(absF_Set(left(t)), absF_Set(right(t))), value(t))
     end
```

However, since *union* is not a constructor, it is not allowed to be used in the definition of *absF_Set*. To solve this problem, we introduce an auxiliary function *abs*1 as follows:

```
abs1(t:Bin_tree, s:Set) =
    if isnew(t) then
        s
    else
        abs1(right(t), insert(abs1(left(t),s), value(t)))
    end
```

Using $abs1$, we can easily define $absF\_Set$ by:

```
absF_Set(t:Bin_tree) = abs1(t, empty)
where
    abs1(t:Bin_tree, s:Set) =
        if isnew(t) then
            s
        else
            abs1(right(t), insert(abs1(left(t),s), value(t)))
        end
end
```

4 Abstractors

So far, we have described the application of abstractors in communicating ADT values and a notation for writing abstraction functions. In this section, we give a definition of abstractors and show how to derive abstractors from abstraction functions.

4.1 Abstractors

In order to define abstractors, we first need to define construction terms. For a type T, the set of its construction terms, $Term\_T$, is the smallest set of strings which satisfies:

1. If $f : \longrightarrow T$ is a constructor of T, then the symbol `f` is in $Term\_T$.
2. If $f : T_1 \times \ldots \times T_n \longrightarrow T$ is a constructor of T, then for every $\mathtt{t}_1 \in Term\_T_1, \ldots, \mathtt{t}_n \in Term\_T_n$, the string $\mathtt{f(t_1,\ldots,t_n)}$ is in $Term\_T$.

Thus, a construction term is a string built from alphabets including symbols "(", ")", ",", constructor symbols of T, and construction terms of base types of T. To distinguish construction terms from other expressions, in the sequel, we use the `typewriter font` for them.

The **abstractor** of T, denoted by abs_T, is a function

$$abs\_T : T \longrightarrow Term\_T$$

which takes as input a value and produces as output a construction term whose evaluation yields the value. We assume the abstractors of base types already exist. For simplicity, if T' is a base type, we use `x` to represent $abs\_T'(x)$.

Formally, abs_T can be specified in the following way: for every constructor of T

$$cons : T_1 \times \ldots \times T_n \longrightarrow T$$

define an equation

$$abs\_T(cons(x_1, \ldots, x_n)) = \mathtt{cons(}abs\_T_1(x_1)\mathtt{,} \ldots \mathtt{,}abs\_T_n(x_n)\mathtt{)}$$

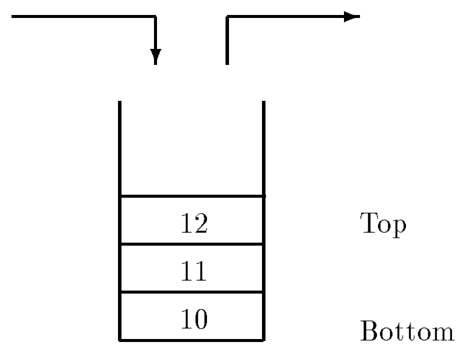


Figure 5: An abstract stack

For example, the abstractor of *Stack* can be specified by

$abs\_Stack : Stack \longrightarrow Term\_Stack$

$abs\_Stack(new) = \texttt{new}$

$abs\_Stack(push(s, e)) = \texttt{push(}abs\_Stack(s)\texttt{,e)}$

where "push($abs\_Stack(s)$,e)" is a concatenation of the string "push(", the string returned by $abs\_Stack(s)$, and the string ",e)".

Consider the stack shown in Figure 5 which has 12 at the top and 10 at the bottom. Taking this stack as an input, *abs_Stack* will return the construction term:

```
push(push(push(new,10),11),12)
```

4.2 General Strategies for Generating Abstractors

Basically, abstractors can be generated from specifications or from implementations. For a given type T, in generation from specifications, a generator analyzes the specification of T and derives the abstractor, which is composed purely of calls on the functions of T. The abstractor is independent of any particular implementation and, therefore, can be used by all of implementations of T. Thus, in this method, only one abstractor is needed for a given type.

For example, the abstractor of *Stack* generated from its specification would be like

```
abs_Stack(v:Stack) =
    if isnew(v)
        new
    else
        push(abs_Stack(pop(v)),top(v))
```

Here *abs_Stack* consists only of functions of *Stack* — *isnew*, *pop*, and *top*; it does not assume any particular implementation, so can be viewed as a built-in function of *Stack* and be exported to the outside.

In generation from implementations, the generator analyzes the relationship between T and a particular implementation type and derives the abstractor for that implementation. Thus, one abstractor is required for each implementation of a type.

As an example, the abstractor for the implementation of *Stack* given in Figure 2 would be like

```
abs_Stack(r: Record[buf:Array[Ele], ptr:Nat]) = abs1(fetch.buf(r), fetch.ptr(r))
where
     abs1(a:Array[Ele], n:Nat) =
          if n=0 then
               new
          else
               push(abs1(a,n-1), fetch(a,n))
          end
end
```

Since *abs_T* involves functions of the implementation type *Record* — $fetch.buf$ and $fetch.ptr$, it can only be used by this implementation. We may view it as a hidden function of the implementation.

This paper focuses on generation from implementations; those interested in details of generation from specifications, please see reference [HL92]. In the next subsection, we shall show how to produce the abstractor of an implementation from its abstraction function.

4.3 Transforming Abstraction Functions into Abstractors

Given an implementation of T, suppose its implementation type is T'. Recall that the abstraction function $absF\_T$ converts concrete values into abstract values, which will be ultimately expressed in terms of functions of T. Since our notation does not allow non-constructor functions of T to be used in the definition of $absF\_T$, the resulting abstract values are actually represented in terms of constructors of T, that is, construction terms. Thus, by replacing every call on a constructor in $absF\_T$ with the corresponding string, $absF\_T$ becomes abs_T.

Below is an algorithm to transform an abstraction function into an abstractor.

Algorithm: Transforming an abstraction function into an abstractor.
Input: The text of an abstraction function $absF\_T$.
Output: The text of the corresponding abstractor abs_T.

For every auxiliary function heading in $absF\_T$
  Replace every occurrance of $x : T$ (if any) with $x : Term\_T$.
For every function definition in $absF\_T$
  Replace every call on a constructor $cons(a_1, \ldots, a_n)$
  (suppose $cons : T_1 \times \cdots \times T_n \longrightarrow T$) with a string `cons(`$abs\_T_1(a_1)$ `,` $\cdots$ `,` $abs\_T_n(a_n)$`)`.

As an example, given the abstraction function $absF\_Set$ in Subsection 3.2.3, the above algorithm will produce the abstractor *abs_Set* (based on the implementation type Bin_tree):

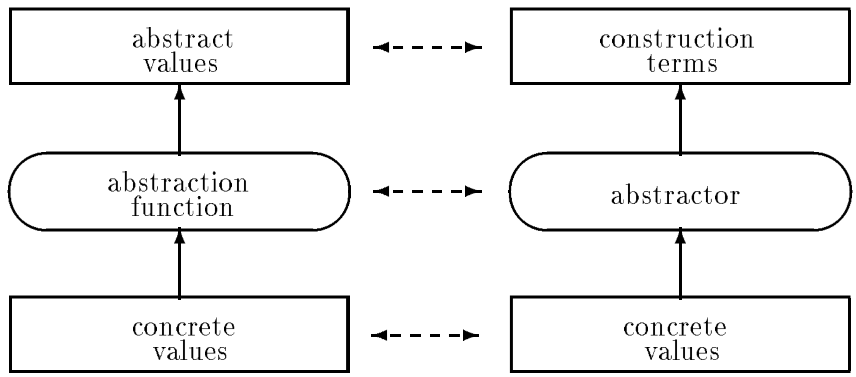


Figure 6: Relationship between abstraction functions and abstractors

absF_Set(t:Bin_tree) = *abs1(t,* `empty`*)*
where
  abs1(t:Bin_tree, s:Term_Set) =
    if *isnew(t)* **then**
      s
    else
      abs1(right(t), `insert`*(abs1(left(t),s), abs_Ele(value(t))))*
    end
end

Figure 6 shows the relationship between an abstraction function and the corresponding abstractor. Any input to the abstraction function (a concrete value) can be an input to the abstractor; any output from the abstraction function (a abstract value) has a corresponding construction term (an output from the abstractor). Thus, for a pair of input and output of the abstraction function, there exists a pair of input and output of the abstractor. The abstraction function and abstractor are naturally related to each other.

4.4 An Example

Let us look at an example of the computation process of abstractors. Consider a concrete stack shown in Figure 7. Its construction term for the stack is

```
push(push(push(new,10),11),12)
```

Now we show that given the concrete stack, *abs_Stack* defined in Subsection 4.2 will return this construction term.

Denoting the concrete stack by r, we have

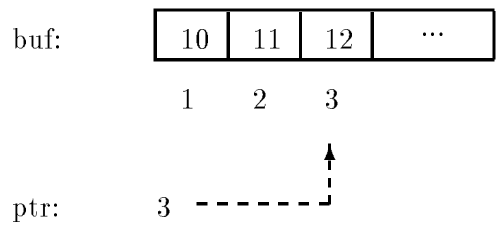


Figure 7: A concrete stack

$$
\begin{aligned}
& abs\_Stack(r) \\
& \quad \{ \text{ by the definition of } abs\_Stack \ \} \\
= \ & abs1(fetch.buf(r), fetch.ptr(r)) \\
& \quad \{ \text{ by } fetch.ptr(r) = 3 \ \} \\
= \ & abs1(fetch.buf(r), 3) \\
& \quad \{ \text{ by the } else \text{ clause of } abs1 \ \} \\
= \ & \mathtt{push}(abs1(fetch.buf(r), 3-1),\ fetch(fetch.buf(r), 3)) \\
& \quad \{ \text{ by } fetch(fetch.buf(r), 3) = 12 \ \} \\
= \ & \mathtt{push}(abs1(fetch.buf(r), 2)\mathtt{,12)} \\
& \quad \{ \text{ by the } else \text{ clause of } abs1 \ \} \\
= \ & \mathtt{push(push(}abs1(fetch.buf(r), 2-1),\ fetch(fetch.buf(r), 2))\mathtt{,12)} \\
& \quad \{ \text{ by } fetch(fetch.buf(r), 2) = 11 \ \} \\
= \ & \mathtt{push(push(}abs1(fetch.buf(r), 1)\mathtt{,11),12)} \\
& \quad \{ \text{ by the } else \text{ clause of } abs1 \ \} \\
= \ & \mathtt{push(push(push(}abs1(fetch.buf(r), 1-1),\ fetch(fetch.buf(r), 1))\mathtt{,11),12)} \\
& \quad \{ \text{ by } fetch(fetch.buf(r), 1) = 10 \ \} \\
= \ & \mathtt{push(push(push(}abs1(fetch.buf(r), 0)\mathtt{,10),11),12)} \\
& \quad \{ \text{ by the } if \text{ clause of } abs1 \ \} \\
= \ & \mathtt{push(push(push(new,10),11),12)}
\end{aligned}
$$

5 Discussions and Conclusions

As far as we know, in the literature, abstraction functions are only used in proving the correctness of implementations and in helping understand implementations. In this paper, we explore a new use of abstraction functions — in generating abstractors. Moreover, the generation is quite simple.

We also propose a notation for writing abstraction functions, which has not been seen in the literature.

Our approach does not place an excessive burden on the user. In the literature, much work calls for the user to provide the abstraction function for every implementation of a type. For example, abstraction functions are an integral part of *Alphard* programs[Sha81]; they are an essential piece of information contained in internal module documents[PM90]; and they are strongly recommended to be provided as comments in *CLU* programs[LG86]. In addition, defining abstraction functions itself should not be a problem to the user, since when designing an implementation of a type, he at least in his mind has the relationship between the implementation type and the type to be implemented.

In systems which use construction terms as exchange representations to communicate ADT values, since abstraction functions can be easily transformed into conversion routines (abstractors), the user is freed from the burden of writing complex conversion routines that translate ADT values from one representation to another.

Our notation requires abstraction functions to be defined in purely functional style. This may sacrifice performance for clarity. One possible direction of future work would be in exploring ways to transform abstraction functions from functional style to imperative style, so as to improve the performance.

Acknowledgments We wish to thank Tianling Lu for her many helpful comments on two earlier drafts of this paper, which, among other things, make the presentation much clearer.

References

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specifications of abstract data types. *Acta Informatica*, 10(1):27–52, 1978.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Current Trends in Programming Methodology, Vol.4 Data Structuring*, pages 80–149. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[HL92] L. Huang and D.A. Lamb. Generating abstractors for abstract data types. Technical Report 92-331, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, July 1992.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press and McGraw-Hill Book Company, 1986.

[Par90] H.A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag Berlin Heidelberg, 1990.

[PM90] D.L. Parnas and J. Madey. Functional documentation for computer systems engineering. Technical Report 90-287, Department of Computing and Information, and TRIO, Queen's University, Kingston, Ontario, September 1990.

[Sha81] M. Shaw, editor. *ALPHARD: Form and Content.* Springer-Verlag New York Inc., 1981.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier Science Publishers B.V./The MIT Press, 1990.

A The Syntax of the Notation for Specifying Abstraction Functions

This section describes the syntax of the specification language for abstraction functions.

We use a variant of Backus-Naur-Form. The particular differences are

1. [X] means X is optional.
2. {X} denotes zero or more repetitions of X.
3. Items enclosed in single quotes ‘ ’ are terminals.

The syntax is as follows.

```
FunctionDef      ::= Id ‘=’ Exp [AuxiliaryDef]
Exp              ::= CompositionExp
                   | ConditionExp
CompositionExp   ::= Id [‘(’ CompositionExp {‘,’ CompositionExp} ‘)’]
ConditionExp     ::= ‘if’ CompositionExp ‘then’ CompositionExp
                     {‘elsif’ CompositionExp ‘then’ CompositionExp}
                     [‘else’ CompositionExp]
                     ‘end’
AuxiliaryDef     ::= ‘where’ FunctionDef {‘and’ FunctionDef} ‘end’
Id               ::= Letter {Letter | Digit | ‘_’}
Letter           ::= ‘A’ | ... | ‘Z’ | ‘a’ | ... | ‘z’
Digit            ::= ‘0’ | ... | ‘9’
```

B Example Abstraction Functions

This section presents example abstraction functions for some common implementations of several types.

For the ease of reading, we use the conventional notation to write functions of types *Array* and *Record*. In particular, we use $A[i]$ to access the element of array A with index i and $R.f$ to access the field f of record R.

B.1 Set

Here we shall consider two implementations of *Set*; its specification is given in Figure 10 at page 23.

B.1.1 A *Record* Representation of *Set*

Suppose *Set* is represented by

Representation *Set=Record[buf:Array[Ele], size:Nat]*

The elements of a set are stored in buf at indexes $1, 2, \ldots, size$; and $size$ is initialized to 0.

The abstraction function is as follows.

```
absF_Set(r:Record[buf:Array[Ele], size:Nat]) = abs1(r.buf, r.size)
where
    abs1(a:Array[Ele], n:Nat) =
        if n=0 then
            empty
        else
            insert(abs1(a,n-1), a[n])
        end
end
```

B.1.2 Another *Record* Representation of *Set*

Suppose *Set* is represented by

Representation *Set=Record[buf:Array[Ele], low:Nat, high:Nat]*

The elements of a set are stored in buf at indexes $low, low+1, \ldots, high$; initially, low and $high$ are set to be equal.

The abstraction function is as follows.

```
absF_Set(r: Record[buf:Array[Ele], low:Nat, high:Nat]) = abs1(r.buf, r.low, r.high)
where
    abs1(a:Array[Ele], l:Nat, h:Nat) =
        if l=h then
            empty
        else
            insert(abs1(a, l+1, h), a[l])
        end
end
```

B.2 Bin_tree

For the specification of *Bin_tree*, see Figure 11 at page 24.

B.2.1 A *Record* Representation of *Bin_tree*

Suppose *Bin_tree* is represented by

Representation *Bin_tree=Record[data:Ele, left:Bin_tree, right:Bin_tree]*

where $data$ stores the data item of the root of a tree, and $left$ and $right$ point to the left subtree and the right subtree respectively.

The abstraction function would be

```
absF_Bin_tree(r: Record[data:Ele, left:Bin_tree, right:Bin_tree])=
   if isnewRec(r) then
      newTree
   else
      maketree(r.data, abs1(r.left), abs1(r.right))
   end
```

B.2.2 An *Array* Representation of *Bin_tree*

Suppose *Bin_tree* is represented by

Representation *Bin_tree=Array[Ele]*

Initially, the root of a tree is at index 1 in the array; and, for a subtree with root at index i, the roots of its left and right subtrees are at $2 \times i$ and $2 \times i + 1$ respectively.

We assume the elements of the array are initialized to a special value, say "null". The abstraction function would be

```
absF_Bin_tree(a:Array[Ele]) = abs1(a, 1)
where
   abs1(a:Array[Ele], n:Nat) =
      if A[n]=null then
         newTree
      else
         maketree(a[n], abs1(a, 2*n), abs1(a, 2*n+1))
      end
end
```

B.3 Graph

To save space, we do not give a specification of *Graph* here. We assume the constructors of *Graph* are

- *newGraph*: creates an empty graph.
- $addNode(g : Graph,\ n : Nat)$: returns a graph which consists of the graph g and the node n if n was not in g, otherwise returns g.
- $addEdge(g : Graph,\ n1 : Nat,\ n2 : Nat)$: returns a graph which consists of the graph g and an edge between $n1$ and $n2$ if no edge between $n1$ and $n2$ was in g, otherwise returns g.

B.3.1 An Adjacency Matrix Representation of *Graph*

An adjacency matrix A is a square matrix of boolean values, where $A[i, j] = True$ means there is an edge from node i to node j, and $A[i, j] = False$ means there is not.

Assume *Array*1 is a two dimensional array type. The representation of *Graph* by adjacency matrix is

Representation Graph=Array1[Boolean]

Suppose the nodes of a graph are numbered from 1 to max. The abstraction function would look like

```
absF_Graph(a: Array1[Boolean]) = abs1(a, max)
abs1(a:Array1[Boolean], n:Nat) = rows(a,n,n,nodes(n))
where
    rows(a:Array1[Boolean], m:Nat, n:Nat, g:Graph) =
        if m=0 then
            g
        else
            columns(a, m, n, rows(a,m-1,n,g))
        end
    where
        columns(a:Array1[Boolean], m:Nat, n:Nat, g:Graph) =
            if n=0 then
                g
            elsif a[m,n]=true then
                addEdge(columns(a,m,n-1,g),m,n)
            else
                columns(a,m,n-1,g)
            end
    end
and
    nodes(n:Nat) =
        if n=0 then
            newGraph
        else
            addNode(nodes(n-1), n)
        end
end
```

where $nodes(n)$ returns

$$addNode(\ldots addNode(newGraph, 1)\ldots n)$$

and $columns(a, m, n, g)$ returns

$$addEdge(\ldots addEdge(g, m, n_1)\ldots m, n_i)$$

whenever $a[m, n_1]$, $\ldots$, $a[m, n_i]$ are true for $0 < n_1 < \ldots < n_i \leq n$.

B.3.2 An Adjacency List Implementation of *Graph*

Now consider another representation of *Graph*, adjacency list, where a graph is represented as a list of nodes, each of which maintains a list of its neighbours. The representation would be

```
Representation Graph=Record[nd:Node, next:Graph]
        where Node=Record[id:Nat, nbours:Node]
```

The abstraction function would be like

```
absF_Graph(r: Record[nd:Node, next:Graph]) = edges(nodes(newGraph,r),r)
where
    edges(g:Graph, r:Record[nd:Node, next:Graph]) =
        if isnewRec(r) then
            g
        else
            edges(edges1(g, r.nd, r.nd.nbours), r.next)
        end
    where
        edges1(g:Graph, head:Node, nbs:Node) =
            if isnewRec(nbs) then
                g
            else
                edges1(addEdge(g, head.id, nbs.id), head, nbs.nbours)
            end
    end
and
    nodes(g:Graph, r:Record[nd:Node, next:Graph]) =
        if isnewRec(r) then
            g
        else
            nodes(addNode(g, r.nd.id), r.next)
        end
end
```

where $nodes(g, r)$ returns

$$addNode(\ \ldots\ addNode(g, r.nd.id)\ \ldots\ r.next.\cdots.next.nd.id)$$

and $edges1(g, head, nbs)$ returns

$$addEdge(\ \ldots\ addEdge(g, head.id, nbs.id)\ \ldots\ head.id, nbs.nbours\cdots nbours.id)$$

C Specification of Several Types

Figures 8 through 11 present the specifications of *Array*, *Record*, *Set* and *Bin_tree* respectively.

Specification

Array

Parameters

Ele:	*Type*

Declaration

Array[Ele]

Base_types

Ele, Nat

Functions

newArray:	$\rightarrow$ *Array*
store:	*Array* $\times$ *Nat* $\times$ *Ele* $\rightarrow$ *Array*
fetch:	*Array* $\times$ *Nat* $\rightarrow$ *Ele*

Constructors

newArray, store

Equations

$$fetch(store(a,n1,e),\ n2) \quad = \begin{cases} e & if\ n1 = n2, \\ fetch(a,\ n2) & otherwise. \end{cases}$$

Figure 8: Specification of *Array*

Specification

Record

Parameters

$f_1, \ldots, f_n$:	*Identifier*

Declaration

Record$[f_1 : T_1, \ldots, f_n : T_n]$

$T_1, \ldots, T_n$:	*Type*

Base_types

Boolean, $T_1, \ldots, T_n$

Functions

newRec:	$\rightarrow$ *Record*
isnewRec:	*Record* $\rightarrow$ *Boolean*
store.f_i:	*Record* $\times\ T_i \rightarrow$ *Record*
fetch.f_i:	*Record* $\rightarrow T_i$

Constructors

newRec, *store*.f_i

Equations

$$\mathit{fetch}.f_i(\mathit{store}.f_j(r,v)) = \begin{cases} v & \text{if } i = j, \\ \mathit{fetch}.f_i(r) & \text{otherwise.} \end{cases}$$

$$\mathit{isnewRec}(\mathit{newRec}) = \mathit{true}$$

$$\mathit{isnewRec}(\mathit{store}.f_i(r,\ v)) = \mathit{false}$$

Figure 9: Specification of *Record*

Specification

Set

Parameters

Ele:	*Type*

Declaration

Set[Ele]

Base_types

Boolean, Ele

Functions

empty:	$\rightarrow$ *Set*
isempty:	*Set* $\rightarrow$ *Boolean*
insert:	*Set* $\times$ *Ele* $\rightarrow$ *Set*
has:	*Set* $\times$ *Ele* $\rightarrow$ *Boolean*
union:	*Set* $\times$ *Set* $\rightarrow$ *Set*

Constructors

empty, insert

Equations

$$\begin{aligned}
&\textit{has(empty, e)} &&= \textit{true} \\
&\textit{has(insert(s, e1), e2)} &&= \begin{cases} \textit{true} & \textit{if } e1 = e2, \\ \textit{has(s, e2)} & \textit{otherwise.} \end{cases} \\
&\textit{union(s, empty))} &&= s \\
&\textit{union(s1, insert(s2, e))} &&= \textit{insert(union(s1, s2), e)}
\end{aligned}$$

Figure 10: Specification of *Set*

Specification

Bin_tree

Parameters

Ele:	*Type*

Declaration

Bin_tree[Ele]

Base_types

Boolean, Ele

Functions

newTree:	$\rightarrow$ *Bin_tree*
maketree:	*Bin_tree* $\times$ *Ele* $\times$ *Bin_tree* $\rightarrow$ *Bin_tree*
left :	*Bin_tree* $\rightarrow$ *Bin_tree*
right:	*Bin_tree* $\rightarrow$ *Bin_tree*
data:	*Bin_tree* $\rightarrow$ *Ele*
isnew:	*Bin_tree* $\rightarrow$ *Boolean*

Constructors

newTree, maketree

Equations

left(maketree(l, e, r))	= *l*
right(maketree(l, e, r))	= *r*
data(maketree(l, e, r))	= *e*
isnew(newTree)	= *true*
isnewTree(maketree(l, e, r))	= *false*

Figure 11: Specification of *Bin_tree*