Generating Abstractors for Abstract Data Types[1]

Lin Huang David Alex Lamb

July 27, 1992
External Technical Report
ISSN-0836-0227-
92-331

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

Version 1.1
Document prepared July 27, 1992

Abstract

Values of an abstract data type (ADT) may be built by some of its functions called constructors. A construction term of an ADT value is an expression which contains only constructors and whose evaluation yields the value. For a given ADT, the abstractor is a function that converts its values to the corresponding construction terms. Abstractors may be used in communicating ADT values in distributed programs.

This paper addresses the problem of generating abstractors of types from their algebraic specifications. We classify specifications into two classes: symmetric and asymmetric. We show that for a given type

- if its specification is symmetric, the abstractor can be automatically generated, and
- if the specification is asymmetric, it is feasible to generate the abstractors when the specification meets certain conditions,

We also present abstractor skeletons for types whose abstractors can be automatically generated.

[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

List of Tables

1 Introduction

An abstract data type (ADT) is characterized by a collection of sorts (types) and a collection of functions. Values of the type are not directly specified; instead, they can be built by calls on a subset of the type's functions, called constructors. A construction term of a value is an expression which contains only constructors and whose evaluation yields the value.

An *abstractor* for 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 (See Subsection 2.4). In such systems, ADT values are exchanged in their construction terms; any value to be communicated must be converted into its corresponding construction term before being transmitted.

This paper addresses the problem of generating abstractors from algebraic specifications of abstract data types. We do not intend to advance the theory of algebraic specifications, rather we use algebraic specifications as a tool for our work. As discussed in Subsection 2.3, the problem is hard. We know of no previous attempts at it. As the first step, this paper focuses on studying syntactic constraints on the form of specifications which guarantee automatical generation of abstractors. We classify specifications into two classes: symmetric and asymmetric. For types with symmetric specifications, we show that the abstractors can be automatically generated; for types with asymmetric specifications, we show that if the specifications meet certain conditions, it is feasible to generate the abstractors. We also present abstractor skeletons for types whose abstractors can be automatically generated. The skeletons are composed only of calls on the functions provided by the types in question. Throughout the paper, we give several examples to show the applications of our results.

The outline of this paper is as follows. Section 2 discusses the problem we are to solve. Sections 3 and 4, the main work of this paper, study the abstractors for types with symmetric and asymmetric specifications respectively. Section 5 summarizes our results.

2 The Problem

In this section, we present the basic idea of algebraic specifications, give a definition of abstractors, describe the problem we are to attack, and discuss an application of abstractors.

2.1 Algebraic Specifications

There is a large literature on algebraic specification methods [GH78, GTW78, EM85, Wir90]. Here we briefly describe the basic idea.

Figure 1 gives an algebraic specification for the well-know type *Stack*. As shown in the figure, an algebraic specification consists of:

TOI describing the type being specified—the *T*ype *O*f *I*nterest.

Base_types describing the types on which the TOI is based. Base types include parameter types and non-parameter types. A parameter type can take on any particular type.

Since the Boolean type is assumed the underlying type of algebraic specification methods, it does not need to be listed in this clause.

Functions describing the functions of the type, including the function symbols, the domains and ranges.

Constructors describing a subset of the type's functions which is used to generate all of its values. The range of a constructor must be the TOI.

Type Stack

TOI

Stack

Base_types

Parameters:	*Ele*
Non-parameters:	*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*

Equations describing the relations among the functions of the type.

In this paper, we choose the initial model[GTW78, EM85] as the meaning of a specification. Informally, this means that two values of TOI are equated only when otherwise the equations may be not satisfied. In the other words, the domain of TOI contains as many different values as possible. Compared with other models, the initial model has two distinct advantages:

- Existence—initial algebras always exist. One may always construct an initial algebra by constructing a quotient algebra[GTW78].
- Uniqueness—all the initial algebras of a specification are isomorphic.

A particular advantage of using the initial model in this work is discussed in Subsection 2.2.

We now define several terms.

A constructor is called a **constant** if its domain is empty, and a **composite constructor** if its domain contains at least two types. For the type *Stack*, *new* is a constant, and *push* is a composite constructor.

A type is called a **composite type** if at least one of its constructors is composite. For example, *Stack* is a composite type, while *Boolean* and *Integer* are not.

Values that are used to construct a value of a composite type are called **components** of the value.

A function is called a **selector** if its range is a type contained in the domain of some composite constructor. Selectors extract components from values. In type *Stack*, *pop* and *top* are selectors.

2.2 Abstractors

A **construction term** of a value is an expression that consists only of constructors and whose evaluation yields the value.

The **abstractor** of type T, denoted by abs_T, takes as input a value and produces as output the corresponding construction term. Abs_T is a function

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

In the sequel, we use the `typewriter font` for terms. We shall assume abstractors for the base types of T are already available.

Consider type *Stack*. Suppose it is implemented by a linked structure. The concrete stack shown in Figure 2 represents a stack that has $e3$ at its top and $e1$ at its bottom. Given this stack, abs_Stack will return the construction term:

```
push(push(push(new, e1), e2), e3)
```

It is trivial to give the specification of an abstractor. This can be done as follows. For every constructor of T

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

an equation

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

is added to the specification. For example, the abstractor of the type *Stack* can be specified as

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

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

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

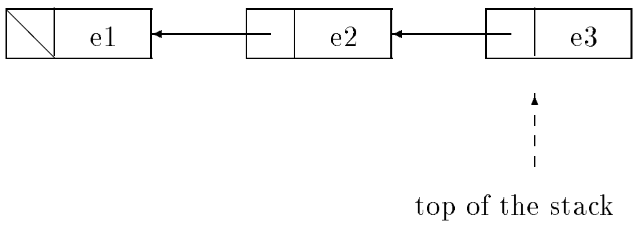


Figure 2: A Concrete Stack

However, as we will see in Subsection 2.3, generating the implementation of an abstractor is not so easy.

Our approach is based on the initial model. The initial algebra, among all the algebras that satisfy a given specification, is the finest-grained in that two values are considered to be different unless they can be proved to be equivalent. Other algebra are coarser-grained since they equate more values than strictly necessary. This means two values equivalent under the initial model are also equivalent under other models. Therefore, abstractors generated in the initial model can be also used in other models.

2.3 The Problem Statement

The problem we are to investigate is how to generate implementations of abstractors.

Basically, abstractors can be generated from types' specifications or from types' implementations. In the method of generation from specifications, a generator analyzes the specification of type T and derives the abstractor for T, which is composed only of calls on the functions of T. On the other hand, in the method of generation from implementations, a particular implementation of T is provided via an abstract model. The generator analyzes the relationship between T and the types used to implement T, and derives the abstractor for this particular implementation.

The major advantage of generation from specifications is that the resulted abstractor is independent of any particular implementation and, therefore, can be used by any implementation of a type. Only one abstractor is needed for a given type. In contrast, the abstractor generated from an implementation is applicable only to that implementation. Each implementation of a type requires its own abstractor. The performance of abstractors generated from implementations, however, might be better because the generator can take into account the particularities of the implementations. Also, as we will see, specifications may not always provide the "right" functions for generating an abstractor.

This paper focuses on generating abstractors from specifications. Like many other problems dealing with semantics of formal specifications, it is in general unsolvable. It is essentially equivalent to the problem of finding the *inverse function* of a given function (constructor), which is in general unsolvable. From a practical view of point, on the other hand, given a value of a type, one does not necessarily have in hand the necessary selector functions to pick out the components from which the value is composed. Fortunately, as shown in the rest of this paper, there exist syntactic constraints of specifications for some types which guarantee the generation of abstractors.

2.4 Term-based Data Exchanges

This subsection discusses a possible application of abstractors to communication of ADT values in distributed programs.

A **distributed program** is composed of a set of modules that reside on nodes of a distributed system and that cooperate to complete a common task. A distributed program is **heterogeneous** if its modules run on different kinds of machines, are written in different languages, and/or use different implementations for the same abstract data type. A **communication** is a process of transmitting data values from one module (the **sender**) to another (the **receiver**). In a communication, data values are transmitted in an **exchange representation** [Lam87], which is the only vehicle the sender and receiver are able to understand.

To support communication of ADT values in heterogeneous distributed programs, a system must deal with conversions between different (concrete) data representations. One approach would be to choose an exchange representation that is acceptable to all the modules of a program and to associate each module with an in-converter and an out-converter. In this approach, a communication involves four steps, illustrated by the solid arrows in Figure 3:

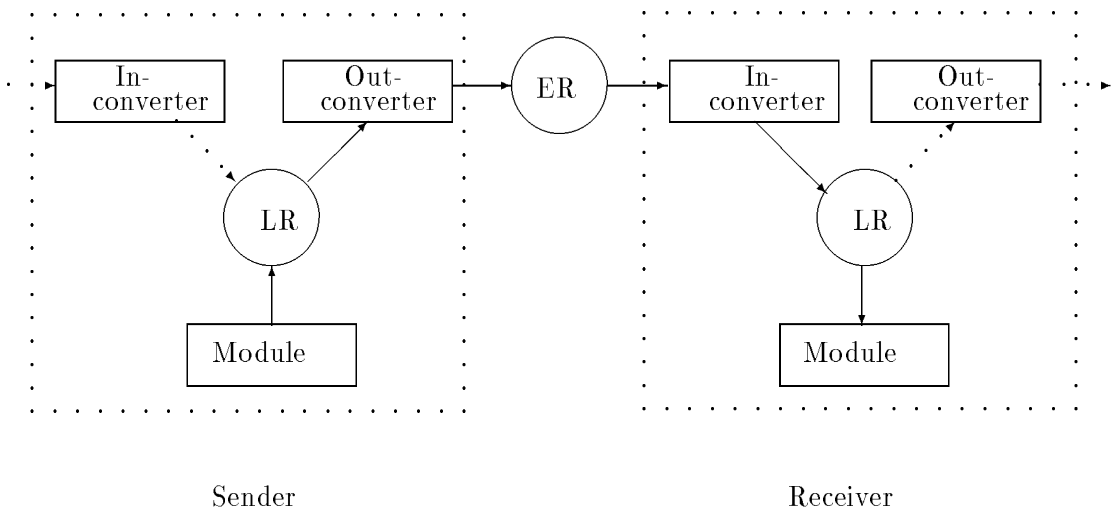
1. The sender initiates a communication.
2. The out-converter at the sender transforms the value to be communicated from the local (concrete) representation used by the sender to the exchange representation.
3. The underlying network system transmits the value in the exchange representation from the sender to the receiver.
4. The in-converter at the receiver transforms the value from the exchange representation to the local (concrete) representation used by the receiver.

The advantages of this approach are

- Representation details are hidden. The details of the local representation used by a module are hidden from other modules. In any communication, the sender is only concerned with how to convert the communicated values from the local representation to the exchange representation and the receiver is concerned with how to convert the values from the exchange representation to the local representation. Neither needs to know the other's local representation.
- Representation changes are localized. Changes in the local representation of a module do not affect other modules' in- or out-converters; only the module's own in- and out-converters need to be changed.
- The number of converters is fixed. Each module uses a fixed number (two) of converters; adding new modules will not cause any new converters to be added to the existing modules.

Two fundamental design issues for this approach are as follows:

- How to choose an appropriate exchange representation?
- How to generate the necessary converters?



ER: Exchange Representation

LR: Local Representation

Figure 3: A Communication Process

One choice of the exchange representation is construction terms. In such systems, a value of an ADT is converted to a construction term before transmission, and the construction term is parsed and evaluated to the local representation used by the receiver upon receipt.

For example, suppose a sender wishes to transmit the stack shown in Figure 2 to a receiver. The out-converter (actually an abstractor) at the sender converts the stack to the construction term

```
push(push(push(new,e1),e2),e3)
```

which is then transmitted to the receiver. The term is automatically parsed by the in-converter (behaving like a syntax-directed translator) at the receiver; and the local representation at the receiver is obtained by calling the appropriate constructors in an appropriate order. Therefore, the in-converter in Figure 3 becomes a simple parser and the out-converter is a set of abstractors.

The advantages of using construction terms as the exchange representation include

- The exchange representation is abstract, since it is a mathematical notation—independent of any particular machine, language, or user. This would increase data portability among different kinds of machines, languages, and ADT implementations.
- The in-converter can be automatically generated since it is just a parser.

Generation of the out-converter (*i.e.*, abstractors) is the topic of this paper.

3 Abstractors for Types with Symmetric Specifications

In this section, we study abstractors for types with symmetric specifications. First, we formulate the definition of symmetric specifications; then, present a procedure to determine if a specification is symmetric; and, finally, give a skeleton of abstractors for types with symmetric specifications.

3.1 Symmetric Specifications

Definition 1 Let T be a type with m constructors $cons_1, \ldots, cons_m$:

$$cons_1 : T_{1,1} \times \ldots \times T_{1,n_1} \longrightarrow T$$

$$\vdots$$

$$cons_m : T_{m,1} \times \ldots \times T_{m,n_m} \longrightarrow T$$

For a given constructor $cons_i$ $(1 \le i \le m)$, if the specification of T contains a function g:

$$g : T \longrightarrow T_{i,j} \ (1 \le j \le n_i)$$

with an equation of the form of

$$g(cons_i(x_1, \ldots, x_j, \ldots, x_{n_i})) = x_j$$

g is called the **j-th normal selector** of $cons_i$. $Cons_i$ is **invertible** if all of its n_i normal selectors are defined in the specification of the type.

In the sequel, the j-th normal selector of $cons_i$ is denoted by $sel_{i,j}$.

Type Binary_tree

TOI

	Bin_tree	

Base_types

	Parameters*:*	*Ele*

Functions

	new:	$\rightarrow$ *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

	new, maketree	

Equations

	left(maketree(l, e, r))	$= l$
	right(maketree(l, e, r))	$= r$
	data(maketree(l, e, r))	$= e$
	isnew(new)	$=$ *true*
	isnew(maketree(l, e, r))	$=$ *false*

Figure 4: Specification of *Binary Tree*

Definition 2 Assume T has m constructors as in Definition 1. If the specification of T contains a function *disc*:

$$disc : T \longrightarrow D$$

where $D = \{d_1, \ldots, d_m\}$ is any m-element set, and if it contains m equations:

$$disc(cons_1(x_{1,1}, \ldots, x_{1,n_1})) = d_1$$
$$\vdots$$
$$disc(cons_i(x_{i,1}, \ldots, x_{i,n_i})) = d_i$$
$$\vdots$$
$$disc(cons_m(x_{m,1}, \ldots, x_{m,n_m})) = d_m$$

then *disc* is called a **discriminator** of T and D is called a **discrimination set.**

By this definition, one may easily verify that *isnew* of type *Stack* is a discriminator.

Definition 3 The specification of a type is **symmetric** if

1. the type has a discriminator, and
2. all the constructors but constants are invertible.

We now present an algorithm to check if the specification of type T is symmetric.

```
For each non-constant constructor $cons$
    For $j$ from 1 to the number of the arguments of $cons$
        If $cons$ does not have the $j$-th normal selector (Definition 1)
            Return "The specification is asymmetric"
    End_for
End_for
If $T$ has a discriminator (Definition 2)
    Return "The specification is symmetric"
Else Return "The specification is asymmetric"
```

Many specifications are symmetric (See Table 1 on Page 26). The specification of type *Stack* is an obvious example. Using the above algorithm, we know the specification of type *Binarytree*(shown in Figure 4) is also symmetric, where the discriminator is *isnew*, and the three normal selectors of *maketree* are *left* , *data*, and *right* respectively. On the other hand, some specifications are asymmetric. The specification of type *Queue* (shown in Figure 5) is such an example; its constructor *enqueue* does not have the normal selectors by Definition 1.

3.2 An Abstractor Skeleton

For any value of a type with symmetric specification, we can determine which constructor has constructed the value by applying the discriminator to the value, and select the components of the value by applying the normal selectors of the corresponding constructor to the value. This is exactly the way in which an abstractor works.

Type Queue

TOI

Queue

Base_types

Parameter:	*Ele*

Functions

new:	$\rightarrow$ *Queue*
enqueue:	*Queue* $\times$ *Ele* $\rightarrow$ *Queue*
front:	*Queue* $\rightarrow$ *Ele*
dequeue:	*Queue* $\rightarrow$ *Queue*
isnew:	*Queue* $\rightarrow$ *Boolean*

Constructors

new, enqueue

Equations

front(enqueue(new, e))	= *e*
front(enqueue(enqueue(q, e), e1))	= *front(enqueue(q, e))*
dequeue(enqueue(new, e))	= *new*
dequeue(enqueue(enqueue(q, e), e1))	= *enqueue(dequeue(enqueue(q, e)), e1)*
isnew(new)	= *true*
isnew(enqueue(q, e))	= *false*

Figure 5: Specification of *Queue*

Proposition 1 *Suppose the specification of T is symmetric. Let the constructors of T*

$$cons_1 : T_{1,1} \times \ldots \times T_{1,n_1} \longrightarrow T$$

$$\vdots$$

$$cons_m : T_{m,1} \times \ldots \times T_{m,n_m} \longrightarrow T$$

Let disc be a discriminator and $\{d_1, \ldots, d_m\}$ *a discrimination set. The abstractor skeleton for T is*

```
abs_T(v:T) =
    if $disc(v)=d_1$
        $cons_1(abs\_T_{1,1}(sel_{1,1}(v)), \ldots, abs\_T_{1,n_1}(sel_{1,n_1}(v)))$
    else if $disc(v)=d_2$
        $cons_2(abs\_T_{2,1}(sel_{2,1}(v)), \ldots, abs\_T_{2,n_2}(sel_{2,n_2}(v)))$
    ⋮
    else
        $cons_m(abs\_T_{m,1}(sel_{m,1}(v)), \ldots, abs\_T_{m,n_m}(sel_{m,n_m}(v)))$
```

If any constructor $cons_i (1 \leq i \leq m)$ is a constant (*i.e.*, its domain is empty), the corresponding statement above would simply be

```
    if $disc(v)=d_i$
        $cons_i$
```

3.3 An Example

We now show the use of the skeleton by an example. Consider the type *Binary_tree* in Figure 4. The correspondence between the functions of *Binary_tree* and those of Proposition 1 is

$cons_1$	:	*new*
$cons_2$	:	*maketree*
$sel_{2,1}$	:	*left*
$sel_{2,2}$	:	*data*
$sel_{2,3}$	:	*right*
disc	:	*isnew*

Accordingly, the abstractor for *Binary_tree* is as follows (Please note, since *isnew* returns a boolean value, we simply write *if isnew(v)* for *if isnew(v)=true*).

```
abs_Bin_tree(v:Bin_tree) =
    if isnew(v)
        new
    else
        maketree(abs_Bin_tree(left(v)),abs_Ele(data(v)),abs_Bin_tree(right(v)))
```

4 Abstractors for Types with Asymmetric Specifications

In practice, many specifications are asymmetric. The specification of type *Queue* in Figure 5 is such an example: the constructor *enqueue* does not have its normal selectors. Generation of abstractors for types with asymmetric specifications is difficult. In this section, we consider types that have only two constructors[1]: one is a constant constructor

$$new : \longrightarrow T$$

and the other is a composite constructor

$$cons : T \times E \longrightarrow T$$

where E is a base type. The construction term for any value of those types would be either

$$\texttt{new} \tag{1}$$

or

$$\text{cons(cons(}\ldots\text{cons(new, } e_1\text{)},\ldots,e_{n-1}\text{), } e_n\text{) for } n \geq 1 \tag{2}$$

In the rest of this section, we introduce a notation for abbreviating construction terms. We then classify types into those with keyed selectors and those with non-keyed selectors, and study their abstractors respectively.

4.1 A Notation

Let f be a unary function

$$f : T \longrightarrow T$$

In standard mathematic notation, applying f to x n times

$$f(f(\ldots(x)\ldots))$$

would be abbreviated as

$$f^n(x)$$

This can be recursively defined by

$$f^0(x) = x$$
$$f^{n+1}(x) = f(f^n(x)) \text{ if } n > 0$$

We now extend this notation to binary functions. In the following, $\langle e_1, \ldots, e_n \rangle$ denotes a sequence of n elements with e_1 being the first element and e_n the last.

Definition 4 Let

$$f : T \times E \longrightarrow T$$

For $x : T; e, e_1, \ldots, e_n : E$, $f^*(x, \langle e_1, \ldots, e_n \rangle)$ is defined by

$$f^*(x, \langle\rangle) = x$$
$$f^*(x, \langle e_1, \ldots, e_n \rangle) = f(f^*(x, \langle e_1, \ldots, e_{n-1} \rangle), e_n) \text{ if } n > 0$$

[1]We have not yet looked at types with more than two constructors.

In this notation, the expression

$$f(f(\dots f(x, e_1), \dots, e_{n-1}), e_n)$$

can be abbreviated as

$$f^*(x, \langle e_1, \dots, e_n \rangle)$$

Similarly, the construction terms in (1) and (2) can be rewritten in one expression

$$cons^*(new, \langle e_1, \dots, e_n \rangle), n \geq 0$$

Theorem 1 *Let*

$$f : T \times E \longrightarrow T$$

For $n \geq 0$ and $m \geq 0$, we have

$$f^*(f^*(x, \langle e_1, \dots, e_m \rangle), \langle e'_1, \dots, e'_n \rangle) = f^*(x, \langle e_1, \dots, e_m, e'_1, \dots, e'_n \rangle)$$

Proof. By induction on n; see Appendix A.1. □

We may further extend this notation as follows.

Definition 5 Let

$$f : T \times E_1 \times \dots \times E_m \longrightarrow T \ (m \geq 2)$$

The n $(n \geq 0)$ repeated application of f is defined by

$$\begin{aligned} &f^*(x, \langle\rangle) = x \\ &f^*(x, \langle (e_{1,1}, \dots, e_{1,m}), \dots, (e_{n,1}, \dots, e_{n,m}) \rangle) = \\ &\quad f(f^*(x, \langle (e_{1,1}, \dots, e_{1,m}), \dots, (e_{n-1,1}, \dots, e_{n-1,m}) \rangle), \\ &\quad (e_{n,1}, \dots, e_{n,m})) \text{ if } n > 0 \end{aligned}$$

By grouping the arguments of a function together in a sequence, our notation has the following advantages:

- reducing the length of an expression,
- highlighting the arguments of functions in an expression, and
- highlighting the order in which the arguments are applied to functions.

4.2 Abstractors for Types with a Non-keyed Selector

Let v be a value and its construction term be

$$cons^*(new, \langle e_1, \dots, e_n \rangle)$$

In order to convert v to the construction term, we must be able to pick up every e_1 to e_n from v. This is an iteration process on v [Lam90]. In this subsection, we consider using non-keyed selectors to achieve the iteration.

4.2.1 Non-keyed Selectors

A non-keyed selector is of the form

$$sel : T \longrightarrow E$$

It is called a non-keyed selector because it does not use any keys[2] to select components. In the following, we consider two classes of non-keyed selectors: value-sensitive and seniority-sensitive selectors.

Definition 6 A selector *sel* is **value-sensitive** if, for every sequence $\langle e_1, \ldots, e_n \rangle$ there exists some i in $[1 .. n]$ such that, for every permutation $\langle e'_1, \ldots, e'_n \rangle$ of $\langle e_1, \ldots, e_n \rangle$,

$$sel(cons^*(new, \langle e'_1, \ldots, e'_n \rangle)) = sel(cons^*(new, \langle e_1, \ldots, e_n \rangle)) = e_i$$

Definition 7 A selector *sel* is **seniority-sensitive** if, for every sequence $\langle e_1, \ldots, e_n \rangle$ there exists some i in $[1 .. n]$ such that, for every permutation $\langle e'_1, \ldots, e'_n \rangle$ of $\langle e_1, \ldots, e_n \rangle$,

$$sel(cons^*(new, \langle e'_1, \ldots, e'_n \rangle)) = e'_i$$

Given $v = cons^*(new, \langle e_1, \ldots, e_n \rangle)$, a value-sensitive selector selects a component from v based on the component's value, while a seniority-sensitive selects a component based on its position in the sequence $\langle e_1, \ldots, e_n \rangle$, *i.e.*, the seniority (with respect to the chronological order in which the component was added into the sequence) of the component.

We now present a sufficient condition to test if a selector is value-sensitive.

Theorem 2 *Let sel be a selector. If its equations are in the form of:*

$$sel(cons(new, e)) = e$$

$$sel(cons(cons(x, e_1), e_2)) = \begin{cases} e_2 & \text{if } p(e_2, sel(cons(x, e_1))), \\ sel(cons(x, e_1)) & \text{otherwise.} \end{cases}$$

where p is a comparison function on E:

$$p : E \times E \longrightarrow Boolean$$

and if p is a total order[3] *on the base type E, then sel is value-sensitive.*

Proof. See Appendix A.2. □

Consider the specification of the type *Priority_queue* in Figure 6. *head* is a selector, and its equations are in the same form as those of *sel*'s in Theorem 2, and $<$ is a total order on *Integer*. Therefore *head* is a value-sensitive selector.

The following theorem shows a sufficient condition for seniority-sensitive selectors.

Theorem 3 *Let sel be a selector; and c be an integer and* $c \geq 1$. *Sel is seniority-sensitive if its equations are of the following form*

$$sel(cons(new, e_1)) = e_1$$

$$sel(cons^*(new, \langle e_1, e_2 \rangle)) = e_{i_2} \text{ for some } i_2\text{: } 1 \leq i_2 \leq 2$$

[2]See Section 4.3 for the definition of keys.

[3]Although the problem of determining whether a predicate is a total order is in general undecidable, the knowledge of some particular predicates being total orders on some commonly used types, such as $<$ is a total order on *Integer*, can be built into a generator and used to check the total order condition in this theorem.

Type Priority_queue

TOI

PQueue

Base_type

Non-parameter: *Integer*

Functions

new:	$\rightarrow PQueue$
add:	$PQueue \times Integer \rightarrow PQueue$
head:	$PQueue \rightarrow Integer$
tail:	$PQueue \rightarrow PQueue$
isnew:	$PQueue \rightarrow Boolean$

Constructors

new, add

Equation

$$\begin{aligned}
head(add(new, i)) &= i \\
head(add(add(q, i), i1)) &= \begin{cases} i1 & \text{if } i1 < head(add(q, i1)), \\ head(add(q, i)) & \text{otherwise.} \end{cases} \\
tail(add(new, i)) &= new \\
tail(add(add(q, i), i1)) &= \begin{cases} add(q, i) & \text{if } i1 < head(add(q, i1)), \\ add(tail(add(q, i)), i1) & \text{otherwise.} \end{cases} \\
isnew(new) &= true \\
isnew(add(q, i)) &= false \\
add(add(q, i), i1) &= add(add(q, i1), i)
\end{aligned}$$

Figure 6: Specification of *Priority Queue*

$$\vdots$$

$$sel(cons^*(new, \langle e_1, \ldots, e_{c-1}\rangle)) = e_{i_{c-1}} \textit{ for some } i_{c-1} : 1 \leq i_{c-1} \leq c-1$$

and either

$$sel(cons^*(x, \langle e_1, \ldots, e_c\rangle)) = \begin{cases} e_c & \textit{if } x = new, \\ sel(cons^*(x, \langle e_1, \ldots, e_{c-1}\rangle)) & \textit{otherwise.} \end{cases} \tag{3}$$

or

$$sel(cons^*(x, \langle e_1, \ldots, e_c\rangle)) = e_1$$

Proof. See Appendix A.3. $\square$

For example, in the type *Queue* (Figure 5), the equations of *front* are of the form of (3), where $c = 1$. Hence, *front* is a seniority-sensitive selector; and it always selects the first element in a queue.

4.2.2 Deletors

A deletor removes a component from a composite value. In the following, we discuss two kinds of deletors: non-keyed deletors and self-keyed deletors.

Non-keyed Deletors

Definition 8 Let

$$del : T \longrightarrow T$$

del is called a **non-keyed deletor** if for every sequence $\langle e_1, \ldots, e_n\rangle$,

$$del(cons^*(new, \langle e_1, \ldots, e_n\rangle)) = cons^*(new, \langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n\rangle) \text{ for some } 1 \leq i \leq n$$

That is, a non-keyed deletor removes a component from a given value.

For the sake of our research, we are interested in pairs of selectors and deletors rather than just deletors. The following defines complementary pairs of selectors and deletors.

Definition 9 Let *sel* be a non-keyed selector

$$sel : T \longrightarrow E$$

and *del* be a non-keyed deletor

$$del : T \longrightarrow T$$

sel and *del* are a **complementary pair** (complementary to each other), if for any $v : T$, the component selected by $sel(v)$ is the one deleted by $del(v)$. That is, $sel(v)$ and $del(v)$ work on the same component.

We now present two methods to find complementary pairs.

Theorem 4 *For a non-keyed selector sel*

$$sel : T \longrightarrow E$$

and a non-keyed deletor del

$$del : T \longrightarrow T$$

if all the equations of sel and del are in the following form, they are a complementary pair.

$$sel(cons(new, e)) = e$$

$$sel(cons(cons(x, e_1), e_2)) = \begin{cases} e_2 & \text{if } p(e_2, sel(cons(x, e_1))), \\ sel(cons(x, e_1)) & \text{otherwise.} \end{cases}$$

$$del(cons(new, e)) = new$$

$$del(cons(cons(x, e_1), e_2)) = \begin{cases} cons(x, e_1) & \text{if } p(e_2, sel(cons(x, e_1))), \\ cons(del(cons(x, e_1)), e_2) & \text{otherwise.} \end{cases}$$

where p is a total order comparison function on E.

Proof. Since *sel* and *del* have the same form of equations and the conditions in the equations are the same, they work on the same component of a value. □

For example, *head* and *tail* of the priority queue in Figure 6 is a complementary pair.

Theorem 5 *Let c be an integer and $c \geq 1$. Let sel be a non-keyed selector and del a non-keyed deletor. If their equations are in the following form, they are a complementary pair.*

$$sel(cons(new, e_1)) = e_1 \tag{4}$$

$$\vdots$$

$$sel(cons^*(new, \langle e_1, \ldots, e_{c-1}\rangle)) = e_{i_{c-1}}, 1 \leq i_{c-1} \leq c-1 \tag{5}$$

$$del(cons(new, e_1)) = new \tag{6}$$

$$\vdots$$

$$\begin{aligned} & del(cons^*(new, \langle e_1, \ldots, e_{c-1}\rangle)) = \\ & \quad cons^*(new, \langle e_1, \ldots, e_{i_{c-1}-1}, e_{i_{c-1}+1}, \ldots, e_{c-1}\rangle) \end{aligned} \tag{7}$$

and either

$$sel(cons^*(x, \langle e_1, \ldots, e_c\rangle)) = \begin{cases} e_c & \text{if } x = new, \\ sel(cons^*(x, \langle e_1, \ldots, e_{c-1}\rangle)) & \text{otherwise.} \end{cases} \tag{8}$$

$$\begin{aligned} & del(cons^*(x, \langle e_1, \ldots, e_c\rangle)) = \\ & \quad \begin{cases} cons^*(x, \langle e_1, \ldots, e_{c-1}\rangle) & \text{if } x = new, \\ cons(del(cons^*(x, \langle e_1, \ldots, e_{c-1}\rangle)), e_c) & \text{otherwise.} \end{cases} \end{aligned} \tag{9}$$

or

$$sel(cons^*(x, \langle e_1, \ldots, e_c\rangle)) = e_1$$

$$del(cons^*(x, \langle e_1, \ldots, e_c\rangle)) = cons^*(x, \langle e_2, \ldots, e_c\rangle)$$

Proof. Since *sel* and *del* have the same form of equations, they work on the same component of a value. □

Self-keyed Deletors

Definition 10 Let

$$del : T \times E \longrightarrow T$$

del is called a **delete-one self-keyed** deletor if, given $e : E$ and $v : T$, $del(v, e)$ removes one occurrence of e from v; it is called a **delete-all self-keyed** deletor if $del(v, e)$ removes all occurrences of e from v.

The following gives a form of equations which guarantees self-keyed deletors.

Theorem 6 *Let*

$$del : T \times E \longrightarrow T$$

We have

I. Del is a delete-one self-keyed deletor if its equations are in the form of

$$del(new, e) = new$$
$$del(cons(x, e_1), e) = \begin{cases} x & if\ e = e_1, \\ cons(del(x, e), e_1) & otherwise. \end{cases}$$

II. Del is a delete-all self-keyed deletor if its equations are in the form of

$$del(new, e) = new$$
$$del(cons(x, e_1), e) = \begin{cases} del(x, e) & if\ e = e_1, \\ cons(del(x, e), e_1) & otherwise. \end{cases}$$

Proof. See Appendix A.4. □

A self-keyed deletor may be paired with any value-sensitive selector in a sense that they work on the same component of a value.

4.2.3 Abstractor Skeletons

In this subsection, we present abstractor skeletons for several kinds of types. In each abstractor, a pair of selector and deletor are used to iterate all the components of a value, say v. The basic idea is to repeat the following process until v becomes *new*:

```
select a component from v;
delete that component from v;
```

In the following, we assume *isnew* is a discriminator of T and satisfies equations

$$isnew(new) = true$$

and

$$isnew(cons(t, e)) = false$$

Proposition 2 *Let*

$$sel : T \longrightarrow E$$

$$del : T \longrightarrow T$$

If

I. Sel is a value-sensitive selector, del is a non-keyed deletor, and they are a complementary pair.

II. T has an equation

$$cons(cons(x, e_1), e_2) = cons(cons(x, e_2), e_1)$$

then the abstractor skeleton for T is

```
abs_T(v:T) =
    if isnew(v)
        new
    else
        cons(abs_T(del(v)), abs_E(sel(v)))
```

Proof. See Appendix A.5. □

Proposition 2 can be used to handle type *Priority_queue*.

Suppose a selector *sel* and a deletor *del* have equations in the form of (4) to (7) in Theorem 5, *i.e.*, $c > 1$. Given a value v with less than c components, *sel(v)*/*del(v)* chooses/deletes a component from v in a non-uniformed way. Here, we give a skeleton for types whose selector and deletor are only in the form of Equations (8) and (9).

Proposition 3 *Let*

$$sel : T \longrightarrow E$$
$$del : T \longrightarrow T$$

If the equations of sel and del are in the form of Equations (8) and (9) with $c = 1$*, that is, sel is a seniority-sensitive selector, del is a non-keyed deletor, and they are a complementary pair, then the abstractor for T is*

```
    abs_T(v:T) = emit_T(v, new)
where
    emit_T(v:T, s:Term) =
        if isnew(v)
            s
        else
            emit_T(del(v), cons(s, abs_E(sel(v))))
```

Proof. See Appendix A.6. □

Proposition 3 can be used to handle type *Queue*.

Proposition 4 *Let*

$$sel : T \longrightarrow E$$
$$del : T \times E \longrightarrow T$$

If

I. sel is a value-sensitive selector,

II. T has an equation

$$cons(cons(x, e_1), e_2) = cons(cons(x, e_2), e_1)$$

III. either of the following conditions is valid

III–A. del is a delete-one self-keyed deletor, or

III–B. del is a delete-all self-keyed deletor and T has an equation

$$cons(cons(x, e), e) = cons(x, e)$$

then the abstractor skeleton for T is

```
abs_T(v:T) =
    if isnew(v)
        new
    else
        cons(abs_T(del(v, sel(v))), abs_E(sel(v)))
```

Proof. Similar to the proof of Proposition 2. □

Proposition 4 can be used to handle ordinary types *Set* and *Bag*.

4.3 Abstractors for Types with a Keyed Selector

In this section, we study types with keyed selectors. We give a definition of keyed selectors, investigate a special kind of keyed selector, present the abstractor skeleton for types with this kind of selector, and show two application examples.

4.3.1 Keyed Selectors

A keyed selector has the following functionality

$$sel : T \times K_1 \times \ldots \times K_m \longrightarrow E$$

Given a value v and a component c, an m-tuple $(k_1, \ldots, k_m)$ is a key of c if one may select c by $sel(v, k_1, \ldots, k_m)$. That is,

$$sel(v, k_1, \ldots, k_m) = c$$

In this paper, we only consider selectors with single keys[4]:

$$sel : T \times K \longrightarrow E$$

We now show how a certain kind of equations decides the keys of components in a value. In the following, fg denotes the composite of f and g, where f and g are functions. f^{-1} denotes the inverse of f in the ordinary mathematics sense; that is, for f

$$f : D \longrightarrow R$$

its inverse f^{-1} exists if and only if f is bijective.

The conditions in the following theorem are in general undecidable. They are used here only for proving this somewhat general theorem. With some built-in knowledge (See Subsections 4.3.3and 4.3.4), a generator may check those conditions on some commonly used types.

Theorem 7 *Let sel be a selector whose definition equations are in the form of*

$$sel(cons(x, e), k) = \begin{cases} e & \text{if } k = h(cons(x, e)), \\ sel(x, f(k)) & \text{otherwise.} \end{cases}$$

where

$$h : T \longrightarrow K \text{ and } f : K \longrightarrow K.$$

If

[4]We have not yet looked at selectors with multiple keys.

I. there exists a function $g : K \longrightarrow K$ such that for every $x' : T$ and $e' : E$, the following equation is valid:

$$h(cons(x', e')) = g(h(x'))$$

II. f^{-1} and g^{-1} exist,

III. f^{-1} and g^{-1} are commutative, i.e., $f^{-1}g^{-1} = g^{-1}f^{-1}$, and

IV. for every $n, i \geq 1$ and $i < n$,

$$(f^{-1}g^{-1})^{n-i}(h(cons^*(new, \langle e_1, \ldots, e_n \rangle))) \neq h(cons^*(new, \langle e_1, \ldots, e_n \rangle))$$

then for a value v whose construction term is $cons^*(new, \langle e_1, \ldots, e_n \rangle)$,

I. e_n can be selected by a key $h(v)$. That is,

$$sel(v, h(v)) = e_n$$

II. $e_i (1 \leq i < n)$ can be selected by a key $(f^{-1}g^{-1})^{n-i}(h(v))$. That is,

$$sel(v, (f^{-1}g^{-1})^{n-i}(h(v))) = e_i$$

Proof. See Appendix A.7. □

At the first glance, the conditions in this theorem are very strict. In practice, however, it is likely that h is a constant function (See Subsection 4.3.3 or f an identity function (See Subsection 4.3.4). When h is a constant function, g will be an identity function and so will be g^{-1}; hence, $f^{-1}g^{-1}$ becomes a single function f^{-1}. When f is an identity function, f^{-1} will be an identity function too; thus, $f^{-1}g^{-1}$ becomes a single function g^{-1}. In either case, $f^{-1}g^{-1}$ becomes a single function, making it easier to verify the conditions of Theorem 7.

In addition, we believe that the conditions in Theorem 7 might be natural for keyed selectors, since these conditions ensure that a selector associates a unique key for each component in a value.

4.3.2 An Abstractor Skeleton

This subsection presents an abstractor skeleton for types which satisfy the conditions of Theorem 7. Suppose T is such a type. Given any value v of T whose construction term is $cons^*(new, \langle e_1, \ldots, e_n \rangle)$, according to the theorem, one can pick up e_n to e_1 by using the keys:

$$h(v), f^{-1}(g^{-1}(h(v))), \ldots, (f^{-1}g^{-1})^{n-1}(h(v))$$

The following abstractor is based on this idea to select all the components in a value.

Proposition 5 *Assume T satisfies the conditions in Theorem 7. If T has a function*

$$size : T \longrightarrow Nat$$

with equations

$$size(new) = 0$$
$$size(cons(x, e)) = size(x) + 1$$

then the abstractor skeleton is as follows.

Type Sequence

TOI

Sequence

Base_types

	Parameters:	*Ele*
	Non-parameters:	*Nat*

Functions

	new:	$\rightarrow$ *Sequence*
	insert:	*Sequence* $\times$ *Ele* $\rightarrow$ *Sequence*
	retrieve:	*Sequence* $\times$ *Nat* $\rightarrow$ *Ele*
	length:	*Sequence* $\rightarrow$ *Nat*

Constructors

new, insert

Equations

$$retrieve(insert(s,\ e),\ n) = \begin{cases} e & \text{if } n = 0, \\ retrieve(s,\ n\text{-}1) & \text{otherwise.} \end{cases}$$

$$length(new) = 0$$

$$length(insert(s,\ e)) = length(s)+1$$

Figure 7: Specification of *Sequence*

```
    abs_T(v:T) =
        if isnew(v)
            new
        else
            iter_T(v, size(v), h(v))
where
    iter_T(v:T, m:Nat, k:K) =
        if m=1
            cons(new, abs_E(sel(v, k)) )
        else
            cons(iter_T(v, m-1, $f^{-1}g^{-1}(k)$), abs_E(sel(v, k)))
```

Proof. See Appendix A.8. $\Box$

Note that *size* computes the number of components of a value and is used in the abstractor to control the number of the iteration.

4.3.3 Example: *Sequence*

First we look at the type *Sequence* in Figure 7. In the specification

$$h : Sequence \longrightarrow Nat$$
$$h(s) = 0$$

$f : Nat \longrightarrow Nat$

$f(n) = n - 1$

Since $h(s) = 0$, it is a constant function. We have

$g : Nat \longrightarrow Nat$

and

$g(n) = n$

Therefore,

$f^{-1}(n) = n + 1$

$g^{-1}(n) = n$

$(f^{-1}g^{-1})(n) = n + 1$

The abstractor for *Sequence* would be

```
abs_Sequence(v:Sequence) =
    if isnew(v)
        new
    else
        iter_Sequence(v, length(v), 0)
```

where

```
iter_Sequence(v:Sequence, m:Nat, k:Nat) =
    if m=1
        insert(new, abs_Ele(retrieve(v, k)) )
    else
        insert(iter_Sequence(v, m-1, k+1), abs_Ele(retrieve(v, k)))
```

One may note that given a sequence of $\langle e_1, \ldots, e_n \rangle$, the specification of *Sequence* associates $e_n, \ldots, e_1$ with keys:

$0, 1, \ldots, n - 1$

respectively.

4.3.4 Example: *Tree*

Now turn to a harder example, *Tree* in Figure 8, which is adapted from the reference [GH91]. In the specification

$h : Tree \longrightarrow Nat$

$h(t) = numChildren(t) - 1$

$f : Nat \longrightarrow Nat$

$f(n) = n$

Since

$$\begin{aligned} & h(addChild(t1, t2)) \\ = & \; numChildren(addChild(t1, t2)) - 1 \\ = & \; numChildren(t1) + 1 - 1 \\ = & \; numChildren(t1) - 1 + 1 \\ = & \; (h(t1)) + 1 \end{aligned}$$

Type Tree

TOI

	Tree	

Base_types

	Parameters*:*	*Ele*
	Non-parameters*:*	*Nat*

Functions

	node:	$Ele \rightarrow Tree$
	addChild:	$Tree \times Tree \rightarrow Tree$
	child:	$Tree \times Nat \rightarrow Tree$
	content:	$Tree \rightarrow Ele$
	numChildren:	$Tree \rightarrow Nat$
	isnode:	$Tree \rightarrow Boolean$

Constructors

node, addChild

Equations

$$child(addChild(t1,\ t2),\ n) = \begin{cases} t2 & \text{if } n{=}numChildren(addChild(t1,\ t2))\ \text{-}1, \\ child(t1,\ n) & \text{otherwise.} \end{cases}$$

$$\begin{aligned}
content(node(e)) &= e \\
content(addChild(t1,\ t2)) &= content(t1) \\
numChildren(node(e)) &= 0 \\
numChildren(addChild(t1,\ t2)) &= numChildren(t1){+}1 \\
isnode(node(e)) &= true \\
isnode(addChild(t1,t2)) &= false
\end{aligned}$$

Figure 8: Specification of *Tree*

Adapted from "An LSL Handbook"[GH91]

we have

$$g : Nat \longrightarrow Nat$$

and

$$g(n) = n + 1$$

Therefore,

$$f^{-1}(n) = n$$
$$g^{-1}(n) = n - 1$$
$$(f^{-1}g^{-1})(n) = n - 1$$

The abstractor for *Tree* would be

```
abs_Tree(v:Tree) =
     if isnode(v)
          node(abs_Ele(content(v)))
     else
          iter_Tree(v, numChildren(v), numChildren(v)-1)
where
iter_Tree(v:Tree, m:Nat, k:Nat) =
     if m=1
          addChild(abs_Tree(child(v,k)))
     else
          addChild(iter_Tree(v, m-1, k-1), abs_Tree(child(v, k)))
```

5 Conclusions and Future Work

We have proposed a method to systematically generate abstractors from some algebraic specifications. The method has been applied to the types contained in the *LSL (Larch Shared Language) Handbook* [GH91], some of which have been shown in this paper. Table 1 lists the results[5].

We now summarize our contributions:

- We have classified types into the symmetric and asymmetric classes. We have shown abstractors for symmetric types can be automatically generated. We have identified some conditions which allow one to automatically generate abstractors for asymmetric types.

- We have designed abstractor skeletons for types whose abstractors can be generated. The abstractors contain only the functions provided by the types and are composed in a purely functional way. As a result, the abstractors have the same style as other functions.

- We have extended the repeated function application notation to n-ary functions. The new notation has proven useful in expressing construction terms.

[5]Only composite types appear in the table.

LSL Type	*Proposition/Page*	*Kind*	*Selector*	*Deletor*
Array	not handled			
Bag	prop. 4/pp. 19	asymmetric	value-sensitive	self-keyed
Deque	prop. 1/pp. 11	symmetric		
Graph	not handled			
List	prop. 1/pp. 11	symmetric		
Map	not handled			
PriorityQueue	prop. 2/pp. 18	asymmetric	value-sensitive	non-keyed
Relation	not handled			
Sequence	prop. 5/pp. 21	asymmetric	keyed	
Set	prop. 4/pp. 19	asymmetric	value-sensitive	self-keyed
Queue	prop. 3/pp. 19	asymmetric	seniority-sensitive	non-keyed
Simpletree	prop. 5/pp. 21	asymmetric	keyed	
Stack	prop. 1/pp. 11	symmetric		
String	prop. 5/pp. 21	asymmetric	keyed	

Table 1: Results of Handling the LSL Types

This paper assumes a purely functional system. In a direct translation to an imperative system, data structures would be rebuilt after (often destructively) abstracting them. Although the performance would not be good, it is shown that only an abstractor and a parser would be enough to convert different representations of ADT values. When performance is a priority, one may consider using nondestructive iterators [Lam90] in an imperative language.

In this paper, we are only concerned with generation from specifications. As discussed in Section 3, abstractors could be generated from implementations as well. It would be easier to do so and the performance of the resulted abstractors might be better. How to generate abstractors from implementations has yet to be investigated.

Clearly, much work is needed on looking for conditions that guarantee automatic generation of abstractors for more asymmetric types. We are investigating ways to generate abstractors for the LSL types which we currently cannot handle.

Acknowledgments We are very grateful to Tianling Lu and Andrew Malton for many critical and valuable comments on earlier drafts of this paper.

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.

[GH91] J.V. Guttag and J.J. Horning. An LSL handbook. Digital Equipment Corporation, 1991.

[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.

[Lam87] D.A. Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):267–318, July 1987.

[Lam90] D.A. Lamb. Specification of iterators. *IEEE Transactions on Software Engineering*, 16(12):1352–1359, December 1990.

[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 Proofs of Several Theorems and Propositions

Here we supply the proofs that were skipped in the main body of this paper.

A.1 A Theorem for Abbreviating Expressions of Repeated Function Applications

The following theorem shows how two sequences of arguments in an expression are concatenated into a single sequence.

Theorem 1 *Let*

$$f : T \times E \longrightarrow T$$

For let $m \geq 0$ and $n \geq 0$, we have

$$f^*(f^*(x, \langle e_1, \ldots, e_m \rangle), \langle e'_1, \ldots, e'_n \rangle) = f^*(x, \langle e_1, \ldots, e_m, e'_1, \ldots, e'_n \rangle) \tag{10}$$

Proof. We prove (10) by induction on n.

Base step: $n = 0$

$$\begin{aligned}
& f^*(f^*(x, \langle e_1, \ldots, e_m \rangle), \langle e'_1, \ldots, e'_n \rangle) \\
& \{ \text{ by } n = 0 \ \} \\
= \ & f^*(f^*(x, \langle e_1, \ldots, e_m \rangle), \langle \rangle) \\
& \{ \text{ by Definition 4 } \} \\
= \ & f^*(x, \langle e_1, \ldots, e_m \rangle) \\
& \{ \text{ by simple calculations } \} \\
= \ & f^*(x, \langle e_1, \ldots, e_m \rangle)
\end{aligned}$$

Induction Step: Assume (10) holds for $n, (n > 0)$.

$$\begin{aligned}
& f^*(f^*(x, \langle e_1, \ldots, e_m \rangle), \langle e'_1, \ldots, e'_{n+1} \rangle) \\
& \{ \text{ by Definition 4 } \} \\
= \ & f(f^*(f^*(x, \langle e_1, \ldots, e_m \rangle), \langle e'_1, \ldots, e'_{n+1} \rangle), e_{n+1}) \\
& \{ \text{ by induction hypothesis} \} \\
= \ & f(f^*(x, \langle e_1, \ldots, e_m, e'_1, \ldots, e'_n \rangle), e_{n+1}) \\
& \{ \text{ by Definition 4 } \} \\
= \ & f^*(x, \langle e_1, \ldots, e_m, e'_1, \ldots, e'_n, e_{n+1} \rangle)
\end{aligned}$$

□

A.2 A Sufficient Condition for Value-sensitivity

The following theorem presents a sufficient condition to test if a selector is value-sensitive.

Theorem 2 *Let sel be a selector. If its equations are in the form of:*

$$sel(cons(new, e)) = e \tag{11}$$

$$sel(cons(cons(x, e_1), e_2)) = \begin{cases} e_2 & \text{if } p(e_2, sel(cons(x, e_1))), \\ sel(cons(x, e_1)) & \text{otherwise.} \end{cases} \tag{12}$$

where p is a comparison function on E:

$$p : E \times E \longrightarrow Boolean$$

and if p is a total order on E, sel is value-sensitive.

Proof. First we prove that for any $n (n \geq 1)$

$$sel(cons^*(new, \langle e_1, \ldots, e_n \rangle)) = min(e_1, \ldots, e_n) \tag{13}$$

where $min(e_1, \ldots, e_n)$ denotes the minimum element, under the total order p, of the set $\{e_1, \ldots, e_n\}$.

We prove (13) by induction on n.

Base step: $n = 1$

$$\begin{aligned}
& sel(cons(new, \langle e_1 \rangle)) \\
& \quad \{ \text{ by Definition 4 } \} \\
= \; & sel(cons(new, e_1)) \\
& \quad \{ \text{ by (11)} \} \\
= \; & e_1
\end{aligned}$$

Since e_1 is the only element in e_1, it should also be the smallest one.

Induction step: Assume (13) holds for n. Let

$$e = sel(cons^*(new, \langle e_1, \ldots, e_n \rangle))$$

By the induction hypothesis, e is the minimum element in $\{e_1, \ldots, e_n\}$. We have

$$\begin{aligned}
& sel(cons^*(new, \langle e_1, \ldots, e_{n+1} \rangle)) \\
& \quad \{ \text{ by Definition 4 } \} \\
= \; & sel(cons(cons^*(new, \langle e_1, \ldots, e_n \rangle), e_{n+1})) \\
& \quad \{ \text{ By Equation (12) } \} \\
= \; & \begin{cases} e_{n+1} & \text{if } p(e_{n+1}, e), \\ e & \text{otherwise.} \end{cases}
\end{aligned}$$

- When $p(e_{n+1}, e)$, we have $sel(cons^*(new, \langle e_1, \ldots, e_{n+1} \rangle)) = e_{n+1}$.

  Since

  $$p(e_{n+1}, e) = true \text{ and}$$

  $$p(e, e_i) = true \text{ for any i: } 1 \leq i \leq n$$

  by the transitivity of p,

  $$p(e_{n+1}, e_i) = true \text{ for all i: } 1 \leq i \leq n$$

  Therefore, e_{n+1} is the minimum element in $\{e_1, \ldots, e_{n+1}\}$.

- When $\neg(p(e_{n+1}, e))$, we have $sel(cons^*(new, \langle e_1, \ldots, e_{n+1} \rangle)) = e$.

  Since p is a total order, $\neg(p(e_{n+1}, e))$ implies $p(e, e_{n+1})$ or $e = e_{n+1}$. This means e is also the minimum element in $\{e_1, \ldots, e_{n+1}\}$.

Therefore, (13) is correct.

Similarly, $sel(cons^*(new, \langle e'_1, \ldots, e'_n \rangle))$ will return the minimum element in $\{e'_1, \ldots, e'_n\}$, where $\langle e'_1, \ldots, e'_n \rangle$ is a permutation of $\{e_1, \ldots, e_n\}$. Since $\{e_1, \ldots, e_n\}$ and $\{e'_1, \ldots, e'_n\}$ are the exactly same set and p is a total order, the minima will be the same. Therefore,

$$sel(cons^*(new, \langle e_1, \ldots, e_n \rangle)) = sel(cons^*(new, \langle e'_1, \ldots, e'_n \rangle))$$

By Definition 6, *sel* is value-sensitive. $\square$

A.3 A Sufficient Condition for Seniority-sensitivity

The following theorem shows a sufficient condition for seniority-sensitive selectors.

Theorem 3 *Let sel be a selector; and c be a constant integer and $c \geq 1$. sel is seniority-sensitive, if its equations are in the following form*

$$sel(cons(new, e_1)) = e_1 \tag{14}$$

$$sel(cons^*(new, \langle e_1, e_2 \rangle)) = e_{i_2} \text{ for some } i_2 \text{: } 1 \leq i_2 \leq 2 \tag{15}$$

$$\vdots$$

$$sel(cons^*(new, \langle e_1, \ldots, e_{c-1} \rangle)) = e_{i_{c-1}} \text{ for some } i_{c-1} \text{ :} 1 \leq i_{c-1} \leq c-1 \tag{16}$$

and either

$$sel(cons^*(x, \langle e_1, \ldots, e_c \rangle)) = \begin{cases} e_c & \text{if } x = new, \\ sel(cons^*(x, \langle e_1, \ldots, e_{c-1} \rangle)) & \text{otherwise.} \end{cases} \tag{17}$$

or

$$sel(cons^*(x, \langle e_1, \ldots, e_c \rangle)) = e_1 \tag{18}$$

Proof. Let

$$v = cons^*(new, \langle e_1, \ldots, e_n \rangle), n > 0$$

For $n < c$, the equations (14) to (16) guarantee that $sel(v)$ always returns a component in a fixed position in $\langle e_1, \ldots, e_n \rangle$.

For $n \geq c$, we have two cases:

(i) Equation (17) guarantees that *sel(v)* chooses the c-th component in $\langle e_1, \ldots, e_n \rangle$. That is,

$$sel(cons^*(new, \langle e_1, \ldots, e_n \rangle)) = e_c \text{ if } n \geq c \tag{19}$$

We prove (19) by induction on n.

Base step: $n = c$

$$\begin{aligned} & sel(cons^*(new, \langle e_1, \ldots, e_n \rangle)) \\ & \quad \{ \text{ by n=c } \} \\ = \; & sel(cons^*(new, \langle e_1, \ldots, e_c \rangle)) \\ & \quad \{ \text{ by (17) } \} \\ = \; & e_c \end{aligned}$$

Induction step: Assume (19) holds for $n (n > c)$.

$$\begin{aligned} & sel(cons(new, \langle e_1, \ldots, e_{n+1} \rangle)) \\ & \quad \{ \text{ by Theorem 1 } \} \\ = \; & sel(cons^*(cons^{n+1-c}(new, \langle e_1, \ldots, e_{n-c+1} \rangle), \langle e_{n-c+2}, \ldots, e_{n+1} \rangle)) \\ & \quad \{ \text{ by (17) and } n > c \text{ } \} \\ = \; & sel(cons^*(cons^{n+1-c}(new, \langle e_1, \ldots, e_{n-c+1} \rangle), \langle e_{n-c+2}, \ldots, e_n \rangle)) \\ & \quad \{ \text{ by Theorem 1 } \} \end{aligned}$$

$$= sel(cons^{(n+1-c)+(c-1)}(new, \langle e_1, \ldots, e_{n-c+1}, e_{n-c+2}, \ldots, e_n\rangle))$$
{ by simple calculations }
$$= sel(cons^*(new, \langle e_1, \ldots, e_{n-c+1}, e_{n-c+2}, \ldots, e_n\rangle))$$
{by induction hypothesis }
$$= e_c$$

(ii) Equation (18) guarantees that *sel(v)* chooses the $(n-c+1)$-th component in $\langle e_1, \ldots, e_n\rangle$. That is,

$$sel(cons^*(new, \langle e_1, \ldots, e_n\rangle)) = e_{n-c+1} \text{ for } n \geq c \tag{20}$$

We prove (20) directly.

$$sel(cons^*(new, \langle e_1, \ldots, e_n\rangle))$$
{ by Theorem 1 and $n \leq c$ }
$$= sel(cons^*(cons^{n-c}(new, \langle e_1, \ldots, e_{n-c}\rangle), \langle e_{n-c+1}, \ldots, e_n\rangle))$$
{ by (18) }
$$= e_{n-c+1}$$

Therefore, $sel(v)$ returns the component in a fixed position. □

A.4 A Sufficient Condition for Self-keyed Deletors

The following theorem shows a sufficient condition for self-keyed selectors.

Theorem 6 *Let*

$$del : T \times E \longrightarrow T$$

We have

I. Del is a delete-one self-keyed deletor if its equations are in the form of

$$del(new, e) = new$$
$$del(cons(x, e_1), e) = \begin{cases} x & \text{if } e = e_1, \\ cons(del(x, e), e_1) & \text{otherwise.} \end{cases}$$

II. Del is a delete-all self-keyed deletor if its equations are in the form of

$$del(new, e) = new$$
$$del(cons(x, e_1), e) = \begin{cases} del(x, e) & \text{if } e = e_1, \\ cons(del(x, e), e_1) & \text{otherwise.} \end{cases}$$

Proof. Here we only give the proof of *I*; the proof of *II* is similar. Let

$$v = cons^*(new, \langle e_1, \ldots, e_n\rangle), n > 0$$

We prove by induction on n that for $e : E$, if there exists $1 \leq i \leq n$ such that

$$e_i = e \text{ and } e_j \neq e \text{ for all } i < j \leq n$$

then

$$del(v, e) = cons^*(new, \langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n\rangle) \tag{21}$$

Base step: $n = 1$. It is trivial to show (21) for $n = 1$. We omit the proof here.
Induction Step: Assume (21) holds for n.
Suppose there exists $1 \leq i \leq n+1$ such that

$$e_i = e \text{ and } e_j \neq e \text{ for all } i < j \leq n+1$$

If $i = n+1$,

$$\begin{aligned}
& del(cons^*(new, \langle e_1, \ldots, e_{n+1} \rangle), e) \\
& \{ \text{ by the } if \text{ clause of } del \ \} \\
= \ & cons^*(new, \langle e_1, \ldots, e_n \rangle)
\end{aligned}$$

If $i < n+1$,

$$\begin{aligned}
& del(cons^*(new, \langle e_1, \ldots, e_{n+1} \rangle), e) \\
& \{ \text{ by the } otherwise \text{ clause of } del \ \} \\
= \ & cons(del(cons^*(new, \langle e_1, \ldots, e_n \rangle), e), e_{n+1}) \\
& \{ \text{ by induction hypothesis } \} \\
= \ & cons(cons^*(new, \langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n \rangle), e_{n+1}) \\
& \{ \text{ by simple calculations } \} \\
= \ & cons^*(new, \langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_{n+1} \rangle)
\end{aligned}$$

□

A.5 An Abstractor Skeleton for Types with a Value-sensitive Selector and a Non-keyed Deletor

The following proposition gives an abstractor skeleton for types that have a value-sensitive selector and a non-keyed deletor.

Proposition 2 *Let*

$$sel : T \longrightarrow E$$

$$del : T \longrightarrow T$$

If

I. sel is a value-sensitive selector, del is a non-keyed deletor, and they are a complementary pair.

II. T has an equation

$$cons(cons(x, e_1), e_2) = cons(cons(x, e_2), e_1)$$

then the abstractor skeleton for T is

```
abs_T(v:T) =
    if isnew(v)
        new
    else
        cons(abs_T(del(v)), abs_E(sel(v)))
```

Proof. Let

$$v = cons^*(new, \langle e_1, \ldots, e_n \rangle),\ n \geq 0$$

We need to prove

$$abs\_T(v) = \texttt{cons(}\ldots\texttt{cons(new,}e_1\texttt{)}\ldots e_n\texttt{)} \tag{22}$$

For notation simplicity, in the following, we shall write "e" for $abs\_E(e)$.

We prove (22) by induction on n.

Base step: $n = 0$. It is trivial to show (22) for $n = 0$. We omit the proof here.

Induction Step: Assume (22) holds for n. Let

$$v' = cons^*(new, \langle e_1, \ldots, e_{n+1} \rangle)$$

Suppose

$$del(v') = cons^*(new, \langle e'_1, \ldots, e'_n \rangle) \tag{23}$$

$$sel(v') = e'_{n+1} \tag{24}$$

Since sel and del are a complementary pair, $\langle e'_1, \ldots, e'_n, e'_{n+1} \rangle$ must be a permutation of $\langle e_1, \ldots, e_n, e_{n+1} \rangle$. Hence,

$$\begin{aligned}
& abs\_T(v') \\
& \quad \{ \text{ by the } else \text{ clause of } abs\_T \ \} \\
= \ & \texttt{cons(}abs\_T(del(v'))\texttt{,}abs\_E(sel(v'))\texttt{)} \\
& \quad \{ \text{ by (24) } \} \\
= \ & \texttt{cons(}abs\_T(del(v'))\texttt{,}abs\_E(e'_{n+1})\texttt{)} \\
& \quad \{ \text{ by the notation simplicity assumption } \} \\
= \ & \texttt{cons(}abs\_T(del(v'))\texttt{,}e'_{n+1}\texttt{)} \\
& \quad \{ \text{ by (23) } \} \\
= \ & \texttt{cons(}abs\_T(cons^*(new, \langle e'_1, \ldots, e'_n \rangle))\texttt{,}e'_{n+1}\texttt{)} \\
& \quad \{ \text{ by induction hypothesis } \} \\
= \ & \texttt{cons(cons(}\ldots\texttt{cons(new,}e'_1\texttt{)} \ \ldots e'_n\texttt{),}e'_{n+1}\texttt{)} \\
& \quad \{ \text{ by Condition II } \} \\
= \ & \texttt{cons(cons(}\ldots\texttt{cons(new,}e_1\texttt{)} \ \ldots e_n\texttt{),}e_{n+1}\texttt{)}
\end{aligned}$$

□

A.6 An Abstractor Skeleton for Types with a Seniority-sensitive Selector and a Non-keyed Deletor

The following proposition gives an abstractor skeleton for types that have a seniority-sensitive selector and a non-keyed deletor.

Proposition 3 *Let*

$$sel : T \longrightarrow E$$

$$del : T \longrightarrow T$$

If the equations of sel and del are in the form of Equations (8) and (9) with $c = 1$, that is, sel is a seniority-sensitive selector, del is a non-keyed deletor, and they are a complementary pair, then the abstractor for T is

```
    $abs\_T(v{:}T) = emit\_T(v,$ new$)$
where
    $emit\_T(v{:}T,\ s{:}Term) =$
        if $isnew(v)$
            $s$
        else
            $emit\_T(del(v),$ cons$(s, abs\_E(sel(v))))$
```

Proof. Let

$$v = cons^*(new, \langle e_1, \ldots, e_n \rangle),\ n \geq 0$$

We first prove by induction on n

$$emit\_T(v, s) = \texttt{cons}(\ldots\texttt{cons}(s, e_1)\ldots e_n) \tag{25}$$

Base step: $n = 0$. It is trivial to show (25) for $n = 0$. We omit the proof here.
Induction Step: Assume (25) holds for n. Let

$$v' = cons^*(new, \langle e_1, \ldots, e_{n+1} \rangle)$$

We have

$$\begin{aligned}
& emit\_T(v', s) \\
& \quad \{ \text{ by the } \textit{else} \text{ clause of } emit\_T \ \} \\
= \ & emit\_T(del(v'), \texttt{cons}(s, abs\_E(sel(v')))) \\
& \quad \{ \text{ by Equation (8): } sel(v') = e_1 \} \\
= \ & emit\_T(del(v'), \texttt{cons}(s,\ e_1)) \\
& \quad \{ \text{ by Equation (9): } del(v') = cons^*(new, \langle e_2, \ldots, e_{n+1} \rangle) \} \\
= \ & emit\_T(cons^*(new, \langle e_2, \ldots, e_{n+1} \rangle), \texttt{cons}(s, e_1)) \\
& \quad \{ \text{ by induction hypothesis } \} \\
= \ & \texttt{cons}(\ldots\texttt{cons}(\texttt{cons}(s, e_1), e_2)\ldots e_{n+1})
\end{aligned}$$

Hence,

$$\begin{aligned}
& abs\_T(v) \\
& \quad \{ \text{ by the definition of } abs\_T \ \} \\
= \ & emit\_T(v, \texttt{new}) \\
& \quad \{ \text{ by (25) } \} \\
= \ & \texttt{cons}(\ldots\texttt{cons}(\texttt{new}, e_1)\ldots e_n)
\end{aligned}$$

□

A.7 Sufficient Conditions for Keyed Selectors

The following theorem shows how a certain kind of equations determines the keys of components in a value.

Theorem 7 *Let sel be a selector whose definition equations are in the form of*

$$sel(cons(x, e), k) = \begin{cases} e & \textit{if } k = h(cons(x, e)), \\ sel(x, f(k)) & \textit{otherwise.} \end{cases} \tag{26}$$

where

$$h : T \longrightarrow K \text{ and } f : K \longrightarrow K.$$

If

I. *there exists a function* $g : K \longrightarrow K$ *such that for every* $x' : T$ *and* $e' : E$, *the following equation is valid:*

$$h(cons(x', e')) = g(h(x'))$$

II. f^{-1} *and* g^{-1} *exist,*

III. f^{-1} *and* g^{-1} *are commutative, i.e.,* $f^{-1}g^{-1} = g^{-1}f^{-1}$, *and*

IV. *for every* $n, i \geq 1$ *and* $i < n$,

$$(f^{-1}g^{-1})^{n-i}(h(cons^*(new, \langle e_1, \ldots, e_n\rangle))) \neq h(cons^*(new, \langle e_1, \ldots, e_n\rangle))$$

then for a value v *whose construction term is* $cons^*(new, \langle e_1, \ldots, e_n\rangle)$,

I. e_n *can be selected by a key* $h(v)$. *That is,*

$$sel(v, h(v)) = e_n$$

II. $e_i (1 \leq i < n)$ *can be selected by a key* $(f^{-1}g^{-1})^{n-i}(h(v))$. *That is,*

$$sel(v, (f^{-1}g^{-1})^{n-i}(h(v))) = e_i$$

Proof. The result (a) directly follows from the definition of *sel*. We only need to prove the result (b): for any n and $1 \leq i < n$ the following is valid

$$sel(cons^*(new, \langle e_1, \ldots, e_n\rangle), (f^{-1}g^{-1})^{n-i}(h(cons^*(new, \langle e_1, \ldots, e_n\rangle)))) = e_i \qquad (27)$$

We prove (27) by induction on n. Since $i < n$, if $n = 1$, i must be *0*. Therefore, we start from $n = 2$.

Base step: $n = 2$

Since $1 \leq i < n$, i can only be *1*.

$$\begin{aligned}
& sel(cons^*(new, \langle e_1, e_2\rangle), (f^{-1}g^{-1})^{2-1}(h(cons^*(new, \langle e_1, e_2\rangle)))) \\
& \quad \{ \text{ by condition IV and (26) } \} \\
= \; & sel(cons(new, e_1), f((f^{-1}g^{-1})(h(cons^*(new, \langle e_1, e_2\rangle))))) \\
& \quad \{ \text{ by condition I } \} \\
= \; & sel(cons(new, e_1), f((f^{-1}g^{-1})g(h(cons(new, e_1))))) \\
& \quad \{ \text{ by simple calculations } \} \\
= \; & sel(cons(new, e_1), h(cons(new, e_1))) \\
& \quad \{ \text{ by (26) } \} \\
= \; & e_1
\end{aligned}$$

Induction step: Assume (27) holds for $n (n > 2)$.

$$
\begin{aligned}
& sel(cons^*(new, \langle e_1, \ldots, e_{n+1}\rangle), (f^{-1}g^{-1})^{(n+1)-i}(h(cons^*(new, \langle e_1, \ldots, e_{n+1}\rangle))))\\
& \quad \{ \text{ by Theorem 1 } \}\\
= & sel(cons(cons^*(new, \langle e_1, \ldots, e_n\rangle), e_{n+1}), (f^{-1}g^{-1})^{n-i+1}(h(cons(new, \langle e_1, \ldots, e_{n+1}\rangle))))\\
& \quad \{ \text{ by condition IV and (26) } \}\\
= & sel(cons^*(new, \langle e_1, \ldots, e_n\rangle), f((f^{-1}g^{-1})^{n-i+1}(h(cons(new, \langle e_1, \ldots, e_{n+1}\rangle)))))\\
& \quad \{ \text{ by condition I } \}\\
= & sel(cons^*(new, \langle e_1, \ldots, e_n\rangle), f((f^{-1}g^{-1})^{n-i+1}(g(h(cons^*(new, \langle e_1, \ldots, e_n\rangle))))))\\
& \quad \{ \text{ condition III } \}\\
= & sel(cons^*(new, \langle e_1, \ldots, e_n\rangle), (f^{-1}g^{-1})^{n-i}(h(cons^*(new, \langle e_1, \ldots, e_n\rangle))))\\
& \quad \{ \text{ by induction hypothesis } \}\\
= & e_i
\end{aligned}
$$

□

A.8 An Abstractor Skeleton for Types with a Keyed Selector

The following proposition gives an abstractor skeleton for types that have a keyed selector.

Proposition 5 *Assume T satisfies the conditions in Theorem 7. If T has a function*

$$size : T \longrightarrow Nat$$

with equations

$$size(new) = 0$$
$$size(cons(x, e)) = size(x) + 1$$

then the abstractor skeleton is as follows.

```
    abs_T(v:T) =
        if isnew(v)
            new
        else
            iter_T(v, size(v), h(v))
where
    iter_T(v:T, m:Nat, k:K) =
        if m=1
            cons(new, abs_E(sel(v, k)))
        else
            cons(iter_T(v, m-1, $f^{-1}g^{-1}$(k)), abs_E(sel(v, k)))
```

Proof. Let

$$v = cons^*(new, \langle e_1, \ldots, e_n\rangle),\ n \geq 0$$

We first prove that for $n > 0$, if for a given $k : K$

$$sel(v, k) = e_j \text{ and } j \leq n$$

then for $1 \leq m \leq n$

$$iter\_T(v, m, k) = \texttt{cons(}\ldots\texttt{cons(new, } e_{j-m+1}\texttt{)}\ldots e_j\texttt{)} \tag{28}$$

We prove (28) by induction on m.

Base step: $m = 1$. (28) immediately follows from the true branch of the *if* clause in *iter_T*.

Induction Step: Assume (28) holds for m.

Note that by Theorem 7,

$$sel(v, f^{-1}g^{-1}(k)) = e_{j-1} \tag{29}$$

Now we have

$$
\begin{aligned}
& iter\_T(v, m+1, k) \\
& \quad \{ \text{ by the } else \text{ clause of } iter\_T \ \} \\
= \ & \texttt{cons}(iter\_T(v, m, f^{-1}g^{-1}(k)), abs\_E(sel(v,k))) \\
& \quad \{ \text{ by } sel(v,k) = e_j \ \} \\
= \ & \texttt{cons}(iter\_T(v, m, f^{-1}g^{-1}(k)), abs\_E(e_j)) \\
& \quad \{ \text{ by notation simplicity assumption } \} \\
= \ & \texttt{cons}(iter\_T(v, m, f^{-1}g^{-1}(k)), e_j) \\
& \quad \{ \text{ by induction hypothesis and (29) } \} \\
= \ & \texttt{cons(cons(}\ldots\texttt{cons(new, } e_{(j-1)-m+1}\texttt{)}\ldots e_{j-1}\texttt{)}, e_j\texttt{)} \\
& \quad \{ \text{ by simple calculations } \} \\
= \ & \texttt{cons(cons(}\ldots\texttt{cons(new, } e_{j-(m+1)+1}\texttt{)}\ldots e_{j-1}\texttt{)}, e_j\texttt{)}
\end{aligned}
$$

Hence, for $v = new$,

$$
\begin{aligned}
& abs\_T(v) \\
& \quad \{ \text{ by the } if \text{ clause of } abs\_T \ \} \\
= \ & \texttt{new}
\end{aligned}
$$

and for $v \neq new$,

$$
\begin{aligned}
& abs\_T(v) \\
& \quad \{ \text{ by the } else \text{ clause of } abs\_T \ \} \\
= \ & iter\_T(v, size(v), h(v)) \\
& \quad \{ \text{ by } size(v) = n \ \} \\
= \ & iter\_T(v, n, h(v)) \\
& \quad \{ \text{ by (28) with } m = n \text{ and } j = n. \text{ Note: } j = n \text{ is due to } sel(v, h(v)) = e_n. \ \} \\
= \ & \texttt{cons(}\ldots\texttt{cons(new, } e_1\texttt{)}\ldots, e_n\texttt{)}
\end{aligned}
$$

□