New Generation Computing, 13 (1994) 75-98
OHMSHA, LTD. and Springer-Verlag

Derivation of Algorithms by Introduction of Generation Functions

Liangwei XU
Information Engineering Course, Graduate School, Division of Engineering, The University of Tokyo.
Hideya IWASAKI
Educational Computer Centre, The University of Tokyo.
Masato TAKEICHI
Department of Mathematical Engineering, The University of Tokyo.
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan.

Received 1 April 1993
Revised manuscript received 14 February 1994

Abstract A specification is a mathematical description of the intent we want a program to behave, while an algorithm is a formal description of a program that satisfies the specification. When we want to write an algorithm by defining data domains and functions over these domains, it is common to use recursion equations for function definitions. However, recursion equations lack imaginability such as existential quantifiers in predicate calculus. In this paper we show how executable algorithms are derived from specifications written in ordinary set-theoretic formulas, and illustrate it by deriving several kinds of sorting algorithms as well as a graph algorithm.

Keywords: Formal Program Derivation, Specification, Generation and Test.

§1 Introduction

Where do algorithms come from? Solving problems will result in success after we understand the problem. What we have to do during this stage is to work out for making it clear what should be done, rather than how it should be done. The problem definition written in formal language is called *specification*.

Specifications and implementations have different purposes. Specifications express the programmer's intent as briefly and clearly as possible. Although we may use some basic operators or functions to write specifications,

a specification language is not necessarily an existing computer programming language. In other words, a specification need not be directly executable.

To write a specification we must define data domains, such as numbers, lists, trees and graphs, and functions over these domains. Although there may be various ways to define such functions, recursion equations are widely used for this purpose. The advantages of definition by recursion equations are its extensionability (i.e., it is easily transformed), executability and state-freeness (i.e., equations do not change their truth over time). But such a notation lacks imaginability in specification. It does not allow existential quantifiers, for example. Hence we need to include a more powerful notation such as set abstraction[5] in our specification language which enables us to write down specifications using ordinary set-theoretic formulas.

To illustrate the differences between recursive definition and theoretic definition, consider an example specification. A function of computing all segments of a list may be defined using recursion equations as:

```
segs x = foldr (++) [] (map inits (tails x))
where inits []       = {[]}
      inits x@(_:_)  = {x} ++ inits (init x)
      tails []       = {[]}
      tails x@(_:x') = {x} ++ tails x'
```

Using set-theoretic formula, it can be expressed concisely as

$$Segs\ x = \{y \mid \exists u,\ v(x = u +\!\!+ y +\!\!+ v)\}.$$

We use operators and functions in Haskell.[6] Some operators for lists are overloaded for expressing analogous operations for sets. For example, the list-concatenation operator (++) is used also for set-union, the list-difference operator (\\) is used also for set-difference, and so on.

Because set-theoretic formulas are not executable in general, they should be transformed into a programming language, say Haskell.

In this paper we propose an approach of transforming specifications into executable *generation and test* algorithms by introducing generation functions. A brief description of our idea will be given in the next section. In Section 3 we present the problem with examples. Section 4 deals with the rules of transformation. In Section 5, we illustrate some example transformations. Finally, we summarize what we have done and give remarks on related works.

§2 Outline of Derivation

Suppose that a specification is given in the form

$$Spec :: \alpha \to \{\beta\}$$
$$Spec\ x = \{y \mid Q\ x\ y\}$$

where *Spec* is a specification function which takes a value of type α and gives

the result of type *set of* β. The predicate function Q is defined totally over the domains α and β.

$$Q :: \alpha \rightarrow \beta \rightarrow Bool.$$

If β is an enumerable type of a set b, and Q is computed by a function q, a functional program

```
spec x = {y | y ← b, q x y}
```

may be considered as an algorithm for that specification. We write functional programs in Haskell[6] with slight extensions such as the *set comprehension* notation as in the above definition. The notation of set comprehension consists of an expression representing elements, generators and Boolean expressions, which is similar to the list comprehension notation. The set comprehension in the above example consists of an element expression y, a generator y ← b, and a Boolean expression q x y.

Obviously, the program spec would be unrealistic when β is given a data type such as integer, list, etc., since we do not want to enumerate the whole universal set. In order to obtain more practical programs, we need to construct a smaller set from which elements y can be drawn. In other words, the generation function of y must be derived from the specification.

If we have a pair of functions

$$f :: \alpha \rightarrow \beta \rightarrow \beta$$
$$h :: \alpha \rightarrow \beta \rightarrow \alpha$$

satisfying the condition

$$Q\ x\ y \rightarrow Q\ (h\ x\ y)\ (f\ x\ y),$$

the specification can be rewritten as

$$Spec\ x = \{y \mid \exists z(z \in Spec\ (h\ x\ y);\ y \in Inv\ (f\ x)\ z;\ Q\ x\ y)\}$$

where the function *Inv* denotes the inverse image of a function

$$Inv :: (\alpha \rightarrow \beta) \rightarrow \beta \rightarrow \{\alpha\}$$
$$Inv\ g\ z = \{y \mid z = g\ y\}.$$

This result will be proved in Section 4. Then we have an algorithm

```
spec x = {y | z ← spec (h x y), y ← inv (f x) z, q x y}
```

provided that the sets *Spec* $(h\ x\ y)$ and *Inv* $(f\ x)\ z$ are enumerable.

There may be some problems in this program. Assume that $h\ x\ y = x$ and $f\ x\ y = y$. The above program becomes

```
spec x = {y | z ← spec x, y ← {z}, q x y}
       = {y | y ← spec x, q x y}
```

which is meaningless as an algorithm. So we should choose h such that it satisfies

$$\forall x :: \alpha,\ y :: \beta\colon (h\ x\ y) < x$$

where $<$ is well-founded partial order over domain α. For example, take $h\ x\ y = tail\ x$ for $\alpha = [\gamma]$(lists of γ). Then the above algorithm is transformed into the following one.

```
spec []     = ...
spec (x: xs) = {y | z ← spec xs, y ← inv (f x) z, q x y}.
```

Obviously, $tail\ x$ and x satisfy $(tail\ x) <_{length} x$ where $<_{length}$ is a well-founded partial order and is defined as

$$(x <_{length} y) \leftrightarrow (length\ x < length\ y).$$

The lower bound of it is empty list.

Another problem is that variables y and z depend on each other when h is selected to be depending on y. This problem can be solved if we rewrite *Spec* as*

$$Spec\ x = \{y \mid \exists z,\ w(w \in Spec'\ x;\ z \in Spec\ w;\ y \in Inv\ (f\ x)\ z;\ w = h\ x\ y;\ Q\ x\ y)\}$$

where

$$Spec'\ x = \{w \mid \exists y(w = h\ x\ y;\ Q\ x\ y)\}$$

Here, new variable w is introduced whose generator is $Spec'\ x$. $Spec'\ x$ is a new specification for a subproblem of $Spec\ x$, and it may be transformed in the same manner.

The next problem is the method of computing *Inv*. Although it is difficult to find the inverse image of a function in general, the "standard functions[2]" of functional language usually have the following property.

$$y \in Inv\ (f\ x)\ z \rightarrow y \in r\ (g\ x\ y)\ z$$

where

$$f :: \alpha \rightarrow \beta \rightarrow \beta$$
$$g :: \alpha \rightarrow \beta \rightarrow \gamma$$
$$r :: \gamma \rightarrow \beta \rightarrow \{\beta\}.$$

Table 1 in Section 5 shows functions g's and r's for some standard functions.

If we choose function f satisfying this property,

$$Spec''\ x = \{u \mid \exists y(u = g\ x\ y;\ Q\ x\ y)\}$$

is another subproblem of $Spec\ x$, which is similar to $Spec'\ x$. Using $Spec''\ x$,

* From the result of Lemma 2 in Section 4.

Spec x is transformed into the following.

$$Spec\ x = \{y \mid \exists z,\ u(z \in Spec\ (h\ x\ y);\ u \in Spec''\ x;\ y \in r\ u\ z;\ u = g\ x\ y;\ Q\ x\ y)\}$$

where

$$Spec''\ x = \{u \mid \exists y(u = g\ x\ y;\ Q\ x\ y)\}.$$

In this formula, the generator of y is r and that of u is $Spec''\ x$. Note that $Spec''\ x$ has the same form as $Spec'\ x$.

The last problem is that we leave Q unchanged and use it as a test function in the program, but Q is allowed to involve existential quantifiers which cannot be used directly as a test function in *set comprehension* notation. We should, therefore, rewrite such specification as

$$\begin{aligned} Spec\ x &= \{y \mid \exists z(Q\ x\ y\ z)\} \\ &= map\ fst\ \{(y, z) \mid Q'\ x\ (y, z)\} \end{aligned}$$

where

$$Q\ x\ y\ z \leftrightarrow Q'\ x\ (y, z)$$

and start transformation from specification

$$Spec_1\ x = \{(y, z) \mid Q'\ x\ (y, z)\}.$$

After all transformations described above, we can interpret the result as an algorithm. In the algorithm, each variable is generated by its own generator function which is either a standard function of Haskell or defined in recursive form. Test functions of the algorithm is a combination of $w = h\ x\ y$, $u = g\ x\ y$ and $Q\ x\ y$.

§3 Specifications and Algorithms

A specification of the problem is given in the form

$$\begin{aligned} &Spec \quad :: \alpha \rightarrow \{\beta\} \\ &Spec\ x = \{y \mid Q\ x\ y\} \end{aligned}$$

where

$$Q :: \alpha \rightarrow \beta \rightarrow Bool.$$

Spec is a specification function which takes input of type α and gives the result of type *set of* β. Q is a predicate function totally defined over the domains α and β.

The above specification can be considered as an alternative expression of the equivalent mathematical formula.

$$\forall x :: \alpha,\ y :: \beta:\ y \in Spec\ x \leftrightarrow Q\ x\ y.$$

Some examples follow.

Example 1
Computing all divisions of a list.

$$\begin{aligned} &Divs \quad :: [\alpha] \to \{([\alpha], [\alpha])\} \\ &Divs\ x = \{(y, z) \mid x = y +\!\!+ z\} \end{aligned}$$

For example, we have

$$\begin{aligned} Divs\ [a, b, c] = \{&([], [a, b, c]), ([a], [b, c]), ([a, b], [c]), \\ &([a, b, c], [])\}. \end{aligned}$$

Example 2
Finding the maximum in a set.

$$\begin{aligned} &Max \quad :: \{\alpha\} \to \{\alpha\} \\ &Max\ x = \{y \mid y \in x;\ Maxq\ x\ y\} \end{aligned}$$

where

$$\begin{aligned} &Maxq :: \{\alpha\} \to \alpha \to Bool \\ &Maxq\ x\ y \leftrightarrow \forall a(a \in x \to a \leq y). \end{aligned}$$

For example, $Max\ \{3, 1, 4, 2\} = \{4\}$.

Example 3
Permuting a list.

$$\begin{aligned} &Perms \quad :: [\alpha] \to \{[\alpha]\} \\ &Perms\ x = \{y \mid bag\ x = bag\ y\} \end{aligned}$$

where

$$\begin{aligned} &bag :: [\alpha] \to \{\!\!\{\alpha\}\!\!\} \\ &bag\ [] = \{\!\!\{\}\!\!\} \\ &bag\ (a : x) = \{\!\!\{a\}\!\!\} +\!\!+ bag\ x. \end{aligned}$$

Special braces $\{\!\!\{$ and $\}\!\!\}$ represent a bag constructor, which is similar to [and] for lists, and the append operator $+\!\!+$ for lists is overloaded to express the union operator for bags. The function *bag* takes a list and returns a bag which contains the same elements in the list. For example,

$$\begin{aligned} Perms\ [a, b, c] = \{&[a, b, c], [a, c, b], [b, a, c], [b, c, a], \\ &[c, a, b], [c, b, a]\}. \end{aligned}$$

Example 4
Computing all the segments of a list.

$$\begin{aligned} &Segs \quad :: [\alpha] \to \{[\alpha]\} \\ &Segs\ x = \{y \mid \exists u, v(x = u +\!\!+ y +\!\!+ v)\}. \end{aligned}$$

For example, *Segs* $[a, b, c] = \{[], [a], [b], [c], [a, b], [b, c], [a, b, c]\}$.

Our algorithm is expressed in a *generation and test* style, which is to be expressed in list (set) comprehensions of modern functional languages.[2,6]

The correspondence between specification and algorithm notations follows. A, B_1, ..., B_n are generative domains with corresponding generators a, b1, ..., bn, and P, Q are predicate functions with their counterparts p, q.

(1) $\{y \mid y \in A\} \Leftrightarrow \{y \mid y \leftarrow a\}$

(2) $\{y \mid y \in A;\ Q\ x\ y\} \Leftrightarrow \{y \mid y \leftarrow a,\ q\ x\ y\}$

(3) $\{y \mid \exists z_1, ..., z_n (z_1 \in B_1; ...; z_n \in B_n;\ y \in A;\ P\ x\ z_1 \ldots z_n\ y)\}$
$\Leftrightarrow \{y \mid z1 \leftarrow b1, ..., zn \leftarrow bn,\ y \leftarrow a,\ p\ x\ z1 \ldots zn\ y\}$

(4) $\{y \mid \exists z_1, ..., z_n (z_1 \in B_1; ...; z_n \in B_n;\ y = f\ x\ z_1 ...; z_n;\ P\ x\ z_1 \ldots z_n\ y)\}$
$\Leftrightarrow \{f\ x\ z1 \ldots zn \mid z1 \leftarrow b1, ..., zn \leftarrow bn,\ p\ x\ z1 \ldots zn\ y\}$

§4 Transformation Strategy

We now give a condition for introducing recursion in the specification equation presented above. The following propositions are straightforward from the set theory.

Proposition 1

Given two functions F_1 and F_2 defined by predicates Q_1 and Q_2 as

$$F_1\ x = \{y \mid Q_1\ x\ y\}$$
$$F_2\ x = \{y \mid Q_2\ x\ y\}.$$

If

$$Q_1\ x\ y \rightarrow Q_2\ x\ y$$

holds for any $x :: \alpha$; $y :: \beta$, then

$$F_1\ x \subseteq F_2\ x.$$

Proposition 2

Given two functions F_1 and F_2 defined by predicates Q_1 and Q_2 as

$$F_1\ x = \{y \mid Q_1\ x\ y\}$$
$$F_2\ x = \{y \mid Q_2\ x\ y\}.$$

If

$$F_1\ x \subseteq F_2\ x$$

holds for any $x :: \alpha$, then

$$F_1\ x = \{y \mid y \in F_2\ x;\ Q_1\ x\ y\}.$$

Lemma 1

Given a specification *Spec* $x = \{y \mid Q\ x\ y\}$, and if there are functions

$$f :: \alpha \to \beta \to \beta$$
$$h :: \alpha \to \beta \to \alpha$$

satisfying

$$Q\ x\ y \to Q\ (h\ x\ y)\ (f\ x\ y),$$

then

$$Spec\ x = \{y \mid \exists z(z \in Spec\ (h\ x\ y);\ y \in Inv\ (f\ x)\ z;\ Q\ x\ y)\}.$$

Here, *Inv* denotes the inverse image of a function.

$$Inv :: (\alpha \to \beta) \to \beta \to \{\alpha\}$$
$$Inv\ g\ z = \{y \mid z = g\ y\}.$$

Proof

$$\begin{aligned} Spec\ x &= \{y \mid Q\ x\ y\} && \text{Definition of } Spec \\ &\subseteq \{y \mid Q\ (h\ x\ y)\ (f\ x\ y)\} && \text{Assumption and Proposition 1} \\ &= \{y \mid \exists z(Q\ (h\ x\ y)\ z;\ z = f\ x\ y)\} && \text{Variable introduction} \\ &= \{y \mid \exists z(Q\ (h\ x\ y)\ z;\ y \in Inv(f\ x)\ z)\} && \text{Inversion} \\ &= \{y \mid \exists z(z \in Spec\ (h\ x\ y);\ y \in Inv\ (f\ x)\ z)\} && \text{Definition of } Spec \end{aligned}$$

Putting this and Proposition 2 together, we have

$$Spec\ x = \{y \mid \exists z(z \in Spec\ (h\ x\ y);\ y \in Inv\ (f\ x)\ z;\ Q\ x\ y)\}.\ \square$$

The condition

$$Q\ x\ y \to Q\ (h\ x\ y)\ (f\ x\ y)$$

is an extension of the *continuity condition* presented in Ref. 1) where the predicate Q takes a single argument.

We give two more lemmas used in solving the problems described at Section 2.

Lemma 2

Under the same conditions as Lemma 1, *Spec* x can be further transformed into

$$Spec\ x = \{y \mid \exists z,\ w(w \in Spec'\ x;\ z \in Spec\ w;\ y \in Inv\ (f\ x)\ z;\ w = h\ x\ y;\ Q\ x\ y)\}$$

where

$$Spec'\ x = map\ (h\ x)\ Spec\ x = \{w \mid \exists y(w = h\ x\ y;\ Q\ x\ y)\}.$$

Proof

$$
\begin{aligned}
&Spec\ x \\
&= \{y \mid \exists z(z \in Spec\ (h\ x\ y); \\
&\qquad y \in Inv\ (f\ x)\ z;\ Q\ x\ y)\} && \text{From Lemma 1} \\
&= \{y \mid \exists z,\ w(w = h\ x\ y;\ z \in Spec\ w; \\
&\qquad y \in Inv\ (f\ x)\ z;\ Q\ x\ y)\} && \text{Variable introduction} \\
&= \{y \mid \exists z,\ w(w \in map\ (h\ x)\ (Spec\ x); \\
&\qquad w = h\ x\ y;\ z \in Spec\ w; \\
&\qquad y \in Inv\ (f\ x)\ z;\ Q\ x\ y)\} && y \in Spec\ x \text{ and } map\text{'s definition} \\
&= \{y \mid \exists z,\ w(w \in Spec'\ x;\ w = h\ x\ y; \\
&\qquad z \in Spec\ w;\ y \in Inv\ (f\ x)\ z; \\
&\qquad Q\ x\ y)\} && Spec' \text{ introduction}
\end{aligned}
$$

□

Lemma 3

Under the same conditions as Lemma 1, if there exist functions g and r

$$
\begin{aligned}
&g :: \alpha \to \beta \to \gamma \\
&r :: \gamma \to \beta \to \{\beta\}
\end{aligned}
$$

satisfying

$$\forall\ x, y, z:\ y \in Inv\ (f\ x)\ z \to y \in r\ (g\ x\ y)\ z$$

then

$$
\begin{aligned}
Spec\ x = \{y \mid \exists z,\ u(&z \in Spec\ (h\ x\ y);\ u \in Spec''\ x;\ y \in r\ u\ z; \\
&u = g\ x\ y;\ Q\ x\ y)\}
\end{aligned}
$$

where

$$Spec''\ x = map\ (g\ x)\ Spec\ x = \{u \mid \exists y(u = g\ x\ y;\ Q\ x\ y)\}.$$

Proof

$$
\begin{aligned}
Spec\ x &= \{y \mid \exists z(z \in Spec\ (h\ x\ y); \\
&\qquad y \in Inv\ (f\ x)\ z;\ Q\ x\ y)\} && \text{From Lemma 1} \\
&= \{y \mid \exists z(z \in Spec\ (h\ x\ y); \\
&\qquad y \in r\ (g\ x\ y)\ z;\ Q\ x\ y)\} && \text{Assumption and Propositions 1 and 2} \\
&= \{y \mid \exists z,\ u(z \in Spec\ (h\ x\ y); \\
&\qquad u = g\ x\ y;\ y \in r\ u\ z;\ Q\ x\ y)\} && \text{Variable introduction} \\
&= \{y \mid \exists z,\ u(z \in Spec\ (h\ x\ y); \\
&\qquad u \in map\ (g\ x)\ Spec\ x; \\
&\qquad u = g\ x\ y;\ y \in r\ u\ z;\ Q\ x\ y)\} && y \in Spec\ x \text{ and } map\text{'s definition} \\
&= \{y \mid \exists z,\ u(z \in Spec\ (h\ x\ y); \\
&\qquad u \in Spec''\ x;\ u = g\ x\ y; \\
&\qquad y \in r\ u\ z;\ Q\ x\ y)\} && Spec'' \text{ introduction}
\end{aligned}
$$

□

Using the results of Lemmas 2 and 3, it is easy to show that *Spec x* can

also be transformed as follows:

$$\begin{aligned} Spec\ x\ = \{y \mid \exists z,\ w,\ u(w \in Spec'\ x;\ z \in Spec\ w; \\ u \in Spec''\ x;\ y \in r\ u\ z; \\ u = g\ x\ y;\ w = h\ x\ y;\ Q\ x\ y)\} \end{aligned}$$

where

$$\begin{aligned} Spec'\ x &= \{w \mid \exists y(w = h\ x\ y;\ Q\ x\ y)\} \\ Spec''\ x &= \{u \mid \exists y(u = g\ x\ y;\ Q\ x\ y)\}. \end{aligned}$$

Based on the above lemmas, we can show our transformation steps which was summarized in Section 2.

[**Step 0**] Preprocess: If the given specification has the form of

$$Spec\ x = \{y \mid \exists z(Q\ x\ y\ z)\},$$

rewrite it as

$$Spec\ x = map\ fst\ (Spec_1\ x)$$

where

$$\begin{aligned} Spec_1\ x &= \{(y, z) \mid Q'\ x\ (y, z)\} \\ Q\ x\ y\ z &\leftrightarrow Q'\ x\ (y, z) \end{aligned}$$

and start transformation from $Spec_1\ x$.

[**Step 1**] Find two functions f and h which satisfy the continuity condition. $h\ x\ y$ and x should satisfy a well-founded partial order.

[**Step 2**] If h does not depend on y, then goto Step 3. Otherwise we must remove the mutual dependency between y and z by the result of Lemma 2. Transform the specification

$$Spec'\ x = \{w \mid \exists y(w = h\ x\ y \wedge Q\ x\ y)\}$$

as a subproblem. Then $Spec\ x$ is rewritten as follows:

$$\begin{aligned} Spec\ x = \{y \mid \exists z,\ w(w \in Spec'\ x;\ z \in spec\ w;\ y \in Inv\ (f\ x)\ z; \\ w = h\ x\ y;\ Q\ x\ y)\} \end{aligned}$$

where

$$Spec'\ x = \{w \mid \exists y(w = h\ x\ y;\ Q\ x\ y)\}.$$

Note that $w = h\ x\ y$ in $Spec\ x$ can be used as a test predicate in the algorithm.

[**Step 3**] Use Lemma 3 to solve the inverse image of $(f\ x)$ and find the generation function of y. That is to find two functions g and r which satisfy

$$y \in Inv\ (f\ x)\ z \rightarrow y \in r\ (g\ x\ y)\ z.$$

Table 1 Inverse Functions.

$(f\ x)\ y$	$g\ x\ y$	$r\ u\ z$
tail y	*head* y	$\{u : z\}$
filter $p\ y$	*filter* $(not \cdot p)\ y$	*merge* $u\ z$
drop $n\ y$	*take* $n\ y$	$\{u + + z\}$
$y \backslash\backslash v$	v	$\{y \mid \exists s(y \in merge\ s\ z;\ s \in perms\ u)\}$ (if *bag* $v \subseteq$ *bag* y) where *merge* $x\ [] = \{x\}$ *merge* $[]\ y = \{y\}$ *merge* $(a : x)\ (b : y)$ $= map(a:)(merge\ x(b : y))$ $+ +\ map(b:)(merge(a : x)y)$.

We can prepare a table of g and r for standard functions described in Ref. 1), and some part of it is shown in Table 1.

After g and r are found, transform the specification

$$Spec''\ x = \{u \mid \exists y(u = g\ x\ y \wedge Q\ x\ y)\}$$

as a subproblem. By Lemma 3, *Spec* x is rewritten as follows:

$$Spec\ x = \{y \mid \exists z,\ u(z \in Spec\ (h\ x\ y);\ u \in Spec''\ x;\ y \in r\ u\ z;\ u = g\ x\ y;\ Q\ x\ y)\}$$

where

$$Spec''\ x = \{u \mid \exists y(u = g\ x\ y;\ Q\ x\ y)\}.$$

Similar to $w = h\ x\ y$ in Step 2, $u = g\ x\ y$ can be used as a test predicate in the algorithm.

If both *Spec'* x and *Spec''* x are introduced as a subproblem,* then the original specification is transformed into the following form.

$$Spec\ x = \{y \mid \exists z,\ w,\ u(w \in Spec'\ x;\ z \in Spec\ w;\ u \in Spec''\ x;\ y \in r\ u\ z;\ u = g\ x\ y;\ w = h\ x\ y;\ Q\ x\ y)\}$$

where

$$Spec'\ x = \{w \mid \exists y(w = h\ x\ y;\ Q\ x\ y)\}$$
$$Spec''\ x = \{u \mid \exists y(u = g\ x\ y;\ Q\ x\ y)\}.$$

[**Step 4**] Put all the results together and remove redundant computation. According to the correspondence between set-theoretic specification notation and algorithm notation given in Section 3, we can get an algorithm satisfying the given specification.

---

* One might think that *Spec'* (or *Spec''*) does not seem to be a "subproblem" of *Spec* because it includes $Q\ x\ y$ in its definition. But $Q\ x\ y$ in *Spec'* (*Spec''*) and $w = h\ x\ y$ ($u = g\ x\ y$) are combined together and transformed into another predicate which is different from $Q\ x\ y$. So *Spec'* (*Spec''*) can be regarded as a new specification.

§5 Examples

The first step to be taken in the transformation is to find the functions h and f satisfying the continuity condition. In general, we usually assume the one first, and then find the other. The *source* domain α and the *target* domain $\{\beta\}$ of the specification usually have some recursive structures, which may be reflected in recursive algorithms. If we first choose h and then fix f according to the continuity condition, we may say that we concentrate on the structure of the source and introduce recursion by the *source structure*. And an approach of choosing f first introduces recursion by the *target structure*. Although the domain usually has some structure, decomposition is not necessarily performed by the structure; it may be done by the values taken from that domain. One of such decomposition methods is to use a filter for selecting elements from sets or lists, and it may introduce recursion by *filtering*. We will illustrate how different algorithms are derived from the same specification by different choices of h and f.

5.1 Permuting and Sorting a List

The specification of the problem of permuting a list has been presented in Section 3.

$$\begin{aligned} &Perms \quad :: [\alpha] \to \{[\alpha]\} \\ &Perms\ x = \{y \mid bag\ x = bag\ y\} \end{aligned}$$

The continuity condition is

$$(bag\ x = bag\ y) \to (bag\ (h\ x\ y) = bag\ (f\ x\ y)),$$

and it would be possible to choose several pairs of h and f.

We get a specification of the sorting problem by placing an additional predicate Ord in the above specification as

$$Sort\ x = \{y \mid bag\ x = bag\ y;\ Ord\ y\}$$

where

$$Ord\ y \leftrightarrow \forall i(1 \leq i \leq length\ y - 1 \to y!!i \leq y!!(i + 1)).$$

The continuity condition of sorting is as follows:

$$\begin{aligned} &(bag\ x = bag\ y) \wedge Ord\ y \\ &\quad \to (bag\ (h\ x\ y) = bag\ (f\ x\ y)) \wedge Ord\ (f\ x\ y). \end{aligned}$$

From this observation, we can transform algorithms for permuting a list into ones for sorting by showing that the condition

$$Ord\ y \to Ord\ (f\ x\ y)$$

holds.

〔1〕 Recursion introduction by target structure

[**Step 1**] Find functions f and h. Choose f such that

$$\begin{aligned} f\ x\ y &= tail\ y, && \text{if } y \neq [] \\ &= [], && \text{otherwise.} \end{aligned}$$

From the specification formula, x must be $[]$ when $y = []$.
For the moment, assume that $y \neq []$. Then the continuity condition for this case is

$$(bag\ x = bag\ y) \rightarrow (bag\ (h\ x\ y) = bag\ (tail\ y)).$$

Since we know that for any $a :: \alpha$,

$$(bag\ x = bag\ y) \rightarrow (bag\ (x \backslash\backslash [a]) = bag\ (y \backslash\backslash [a])),$$

and

$$tail\ y = y \backslash\backslash [head\ y], \qquad \text{if } y \neq [],$$

we can choose the function h as

$$\begin{aligned} h\ x\ y &= x \ominus (head\ y), && \text{if } y \neq [] \\ &= [], && \text{otherwise} \end{aligned}$$

where

$$x \ominus a = x \backslash\backslash [a].$$

Obviously, because $bag\ x = bag\ y$, $head\ y$ is an element of x. Therefore x and $h\ x\ y$ have the relation of

$$length\ (h\ x\ y) < length\ x,$$

and they satisfy the well-founded order $<_{length}$ and the lower bound is empty list.

[**Step 2**] Remove mutual dependency. Because h depends on y, we compute the function $Perms'$ as

$$\begin{aligned} Perms'\ x &= \{w \mid \exists y(w = x \ominus (head\ y);\ bag\ x = bag\ y)\} \\ &= \{x \ominus w_1 \mid \exists y(w_1 = head\ y;\ bag\ x = bag\ y)\} \\ &= \{x \ominus w_1 \mid \exists y_1(bag\ x = bag\ (w_1 : y_1))\} \\ &= \{x \ominus w_1 \mid \exists y_1(w_1 \in x;\ bag\ (x \ominus w_1) = bag\ y_1)\} \\ &= \{x \ominus w_1 \mid w_1 \in x\}. \end{aligned}$$

This subproblem is very simple, and we need not transform it any further.

[**Step 3**] Compute the generation function of y. From the definition of f and Table 1, we have $g\ x\ y = \text{head}\ y$ and $r\ u\ z = \{u : z\}$ such that

$$y \in Inv\ (f\ x)\ z \rightarrow y \in r\ (g\ x\ y)\ z.$$

Through transformations similar to Step 2, we have

$$\begin{aligned} Perms''\ x &= \{u \mid \exists y(u = head\ y;\ bag\ x = bag\ y)\} \\ &= \{u \mid u \in x\}. \end{aligned}$$

[**Step 4**] Put all the results together and remove redundant computations. We have

$$\mathsf{perms}\ [\,] = \{[\,]\}$$

from the original specification. And according to the lemmas we stated in Section 4, we can combine the above results as follows.

$$\begin{aligned} Perms\ x &= \{y \mid \exists z,\ w,\ u\ (w \in Perms'\ x;\ z \in Perms\ w; \\ &\qquad u \in Perms''\ x;\ y \in r\ u\ z; \\ &\qquad u = g\ x\ y;\ w = h\ x\ y;\ bag\ x = bag\ y)\} \\ &= \{y \mid \exists z,\ w,\ w_1,\ u\ (w = x \ominus w_1;\ w_1 \in x;\ z \in Perms\ w; \\ &\qquad u \in x;\ \underline{y = u : z;} \\ &\qquad \underline{u = head\ y;}\ \underline{w = x \ominus (head\ y);} \\ &\qquad bag\ x = bag\ y)\} \\ &= \{y \mid \exists z,\ w,\ w_1,\ u\ (\underline{w = x \ominus w_1;}\ \underline{w_1 \in x;}\ z \in Perms\ w; \\ &\qquad \underline{u \in x;}\ y = u : z; \\ &\qquad \underline{w = x \ominus u;}\ bag\ x = bag\ y)\} \\ &= \{y \mid \exists z,\ w,\ u\ (z \in Perms\ w; \\ &\qquad u \in x;\ y = u : z; \\ &\qquad w = x \ominus u;\ bag\ x = bag\ y)\} \\ &= \{y \mid \exists z,\ u\ (z \in Perms\ (x \ominus u); \\ &\qquad u \in x;\ y = u : z; \\ &\qquad bag\ x = bag\ y)\} \end{aligned}$$

We have reached an algorithm where the generators of y, z and u are $u : z$, $Perms\ (x \ominus u)$ and x, respectively. The predicate $bag\ x = bag\ y$ is left as an test predicate.

A careful observation tells us that the algorithm has some redundancy. As we see that

$$\forall x,\ z,\ u\colon bag\ (x \ominus u) = bag\ z \wedge u \in x \rightarrow bag\ x = bag\ (u : z),$$

y produced by the above generator $(u : z)$ always satisfies $bag\ x = bag\ y$. Therefore we can omit the predicate and the final result will be:

$$\mathsf{perms\ x@(\_:\_) = \{u : z \mid u \leftarrow x,\ z \leftarrow perms\ (x\backslash\backslash[u])\}}.$$

From the above result, we obtain a version of the *Selection Sort* algorithm.

```
sort [] = {[]}
sort x@(_:_) = {u:z | u ← [minimum x], z ← sort (x\\[u])}
```

〔2〕 Another recursion introduction by target structure

[**Step 1**] Suppose that

$$f\ x\ y = drop\ (length\ x/2)\ y.$$

Since *bag* satisfies

$$(bag\ x = bag\ y) \rightarrow (bag(x\backslash\backslash w) = bag\ (y\backslash\backslash w))$$

for any list w and the equation

$$drop\ (length\ x/2)\ y = y\backslash\backslash take\ (length\ x/2)\ y$$

holds, we can show that the function h defined as

$$h\ x\ y = x\backslash\backslash take\ (length\ x/2)\ y$$

satisfies the continuity condition. Obviously, under the condition of $bag\ x = bag\ y$, x and $h\ x\ y$ have the relation of

$$length\ (h\ x\ y) < length\ x.$$

They satisfy a well-founded order $<_{length}$ and the lower bounds are the empty list and singleton list.

[**Step 2**] From the definition of h, we have

$$\begin{aligned}
&Perms'\ x\\
&\quad = \{w \mid \exists y(w = x\backslash\backslash take\ (length\ x/2)\ y;\ bag\ x = bag\ y)\}\\
&\quad = \{x\backslash\backslash w_1 \mid \exists y(w_1 = take\ (length\ x/2)\ y;\ bag\ x = bag\ y)\}\\
&\quad = \{x\backslash\backslash w_1 \mid \exists y_1(bag\ x = bag\ (w_1 +\!\!+ y_1);\\
&\qquad\qquad length\ w_1 = length\ x/2)\}\\
&\quad = map\ (x\backslash\backslash)\ PPerms'\ x
\end{aligned}$$

where

$$\begin{aligned}
PPerms'\ x = \{w_1 \mid \exists y_1(&bag\ x = bag\ (w_1 +\!\!+ y_1);\\
&length\ w_1 = length\ x/2)\}.
\end{aligned}$$

Because *PPerms'* has an existential quantifier, it should be transformed through Step 0.

$$\begin{aligned}
&PPerms'\ x\\
&\quad = map\ fst\ \{(w_1, y_1) \mid bag\ x = bag\ (w_1 +\!\!+ y_1);\\
&\qquad\qquad length\ w_1 = length\ x/2\}\\
&\quad = map\ fst\ (PPPerms'\ x)
\end{aligned}$$

where

$$\begin{aligned}
PPPerms'\ x = \{(w_1, y_1) \mid &bag\ x = bag\ (w_1 +\!\!+ y_1);\\
&length\ w_1 = length\ x/2\}
\end{aligned}$$

Now the transformation strategy must be applied to *PPPerms'*. The continuity functions of *PPPerms'* should satisfy

$$Q_1\ x\ (w_1, y_1) \rightarrow Q_1\ (h_1\ x\ (w_1, y_1))\ (f_1\ x\ (w_1, y_1))$$

where

$$Q_1\ x\ (w_1, y_1) \leftrightarrow bag\ x = bag\ (w_1 +\!\!+ y_1) \wedge length\ w_1 = length\ x/2$$

We firstly suppose that

$$h_1\ x\ (w_1, y_1) = x \ominus (head\ w_1).$$

According to the predicate Q_1 and its continuity condition, f_1 should be

$$\begin{aligned} f_1\ x\ (w_1, y_1) &= (tail\ w_1, y_1), && \text{if } even(length\ x) \\ &= ((tail\ w_1) +\!\!+ [head\ y_1], tail\ y_1), && \text{otherwise} \end{aligned}$$

Through Step 2 to Step 4, we have

$$\begin{aligned} &PPPerms'\ x \\ &= \{((a : u), v) \mid a \in x; (u, v) \in PPPerms'\ (x \ominus a)\}, && \text{if } even(length\ x) \\ &= \{(a : (init\ u), (last\ u) : v) \mid a \in x; \\ &\quad (u, v) \in PPPerms'\ (x \ominus a)\}, && \text{otherwise} \end{aligned}$$

Hence

$$\begin{aligned} PPerms'\ x &= \{a : u \mid a \in x; u \in PPerms'\ (x \ominus a)\}, && \text{if } even(length\ x) \\ &= \{a : (init\ u) \mid a \in x; u \in PPerms'\ (x \ominus a)\}, && \text{otherwise} \end{aligned}$$

[**Step 3**] From the definition of f and Table 1, we have $g\ x\ y = take\ (length\ x/2)\ y$ and $r\ u\ z = \{u +\!\!+ z\}$. And

$$Perms''\ x = \{u \mid \exists y(u = take\ (length\ x/2)\ y; bag\ x = bag\ y)\}.$$

This is the same set as *PPerms'* x!

[**Step 4**] Put all the results together and remove redundant computation. We have

```
perms []          = {[]}
perms [a]         = {[a]}
perms x@(_:_:_)   = {y ++ z | y ← (h x), z ← perms (x\\y)}

h [a] =    {[]}
h x@(_:_:_)
  |even (length x) = {a:z | a ← x, z ← h (x\\[a])}
  |otherwise       = {a:(init z) | a ← x, z ← h (x\\[a])}
```

This algorithm says that to get permutations of a list we firstly make all the permutations of half of the list through function h and then append with all the

permutations of the other half.*
A sorting algorithm based on this result is as follows.

```
sort []          = {[]}
sort [a]         = {[a]}
sort x@(_:_:_)   = {y ++ z | y ← sh x, z ← sort (x\\y)}

sh [a] =    {[]}
sh x@(_:_:_)
    | even (length x) = {a : z | a ← [minimum x], z ← sh (x\\[a])}
    | otherwise       = {a : (init z) | a ← [minimum x], z ← sh (x\\[a])}
```

[3] Recursion introduction by source structure

[Step 1] First, we choose the function h as

$$h\ x\ y = drop\ n\ x \qquad (1 \leq n < length\ x).$$

Note that the value n is not specified here, and it can be treated as a value which depends on the length of x. Observe that the function *bag* satisfies

$$(bag\ x = bag\ y) \rightarrow (bag\ (x\backslash\backslash w) = bag\ (y\backslash\backslash w))$$

for any list w. From the above relation and the equation

$$drop\ n\ x = x\backslash\backslash take\ n\ x,$$

we can show that the function f defined as

$$f\ x\ y = y\backslash\backslash take\ n\ x$$

satisfies the continuity condition. Obviously, x and $h\ x\ y$ have the relation of

$$length\ (h\ x\ y) < length\ x.$$

They satisfy a well-founded order $<_{length}$ and the lower bound is empty list and singleton list.

[Step 2] Because h does not depend on y, we can proceed to the next step.

[Step 3] Because of $bag\ x = bag\ y$ we have $bag\ (take\ n\ x) \subseteq bag\ y$. From the definition of f and Table 1, we select $g\ x\ y = take\ n\ x$ and

$$r\ u\ z = \{y \mid \exists s(y \in merge\ s\ z;\ s \in perms\ u)\}.$$

They satisfy

$$y \in Inv\ (f\ x)\ z \rightarrow y \in r\ (g\ x\ y)\ z,$$

---

* Instead of defining $f\ x\ y = drop\ (length\ x/2)\ y$ at Step 1 of this transformation, we might generally define f something like

$$f\ x\ y = drop\ n\ y \qquad (1 \leq n < length\ y)$$

as we will define h in the next subsection. Unfortunately, under such definition it is difficult to find functions f_i and h_i which depend not only on their arguments but also on n in general.

and the subproblem *Perms″ x* becomes

$$Perms''\ x = \{take\ n\ x\}.$$

The function *merge* is defined as follows.

```
merge x [] = {x}
merge [] y@(_ : _) = {y}
merge x@(x′ : x″) y@(y′ : y″) =
    map (x′:) (merge x″ y) ++ map (y′:) (merge x y″)
```

[Step 4] As in the previous derivation, we have

```
perms [] = {[]}
perms [a] = {[a]}
```

from the original specification. We get the result formula for the specification as follows.

```
perms x@(_ : _) = {y | z ← perms (drop n x),
                       s ← perms (take n x),
                       y ← merge s z}
```

As mentioned earlier, we are free to choose the parameter n in the definition of the function h.

If we take $n = 1$, $h\ x\ y$ is equivalent to *tail* x and $f\ x\ y$ to *head* x, and the above expression becomes much simpler. We have

```
perms (a : x) = {y | z ← perms x, y ← merge z [a]}
```

or using a specialized *merge* function called *interleave* in Ref. 2),

```
perms (a : x) = {y | z ← perms x, y ← interleave a z}
interleave a [] = {[a]}
interleave a (z : zs) = {a : z : zs} ++ map (z:) (interleave a zs).
```

Next consider the case $n = length\ x/2$. We have

```
perms x@(_ : _) = {y | z ← perms x2, z1 ← perms x1, y ← merge z z1}
                  where n = length x/2
                        x1 = drop n x
                        x2 = take n x
```

Combining these results with the continuity condition of *Ord*, we get the *Insertion Sort*

```
sort []     = {[]}
sort (a : x) = {y | z ← sort x, y ← insertion a z}

insertion a [] = {[a]}
insertion a (z : zs) | a > z      = map (z:) (insertion a zs)
                     | otherwise  = {a : z : zs}
```

and the *Merge Sort*

```
sort []          = {[]}
sort [a]         = {[a]}
sort x@(_ : _)   = {y | z ← sort x2, z1 ← sort x1, y ← smerge z z1}
                     where n = length x/2
                           x1 = drop n x
                           x2 = take n x
smerge x []          = {x}
smerge [] y@(_ : _)  = {y}
smerge x@(x' : x'') y@(y' : y'')
                       | x' > y'     = map (y':) (smerge x y'')
                       | otherwise = map (x':) (smerge x'' y)
```

algorithms, respectively.

〔4〕 Recursion introduction by filtering

[**Step 1**] Suppose that

$$h\ x\ y = \mathit{filter}\ p\ x$$

where $p :: \alpha \rightarrow \mathit{Bool}$ is some predicate function. For any predicate p, function *bag* has the following property.

$$(\mathit{bag}\ x = \mathit{bag}\ y) \rightarrow (\mathit{bag}\ (\mathit{filter}\ p\ x) = \mathit{bag}\ (\mathit{filter}\ p\ y))$$

Hence the function f can be defined as

$$f\ x\ y = \mathit{filter}\ p\ y$$

so that h and f satisfy the continuity condition. Here we should choose the predicate p to satisfy

$$\mathit{length}\ (\mathit{filter}\ p\ x) < \mathit{length}\ x$$

to ensure $h\ x\ y$ and x satisfy a well-founded order $<_{\mathit{length}}$.

[**Step 2**] Because h does not depend on y, we can proceed to the next step.

[**Step 3**] From the definition of f and Table 1, if we select $g\ x\ y = \mathit{filter}\ (\mathit{not} \cdot p)\ y$ and $r\ u\ z = \mathit{merge}\ u\ z$, we have

$$z = f\ x\ y \rightarrow y \in r\ (g\ x\ y)\ z.$$

And

$$\begin{aligned}\mathit{Spec}''\ x &= \{u \mid \exists y(u = \mathit{filter}\ (\mathit{not} \cdot p)\ y;\ \mathit{bag}\ x = \mathit{bag}\ y)\} \\ &= \{u \mid \exists y(\mathit{bag}\ (\mathit{filter}\ (\mathit{not} \cdot p)\ x) = \mathit{bag}\ u; \\ &\qquad \mathit{bag}\ (\mathit{filter}\ p\ x) = \mathit{bag}\ (\mathit{filter}\ p\ y))\} \\ &= \{u \mid \mathit{bag}\ u = \mathit{bag}\ (\mathit{filter}\ (\mathit{not} \cdot p)\ x)\}\end{aligned}$$

$$= Perms\ (filter\ (not \cdot p)\ x).$$

[**Step 4**] From the original specification, we have

```
perms [] = {[]}
```

We can transform the above formula for the specification as follows.

```
perms x@(_ : _) = {y | z ← perms (filter p x),
                       u ← perms (filter (not · p) x),
                       y ← merge u z}
```

A simple choice of the predicate p something like (using the result of Section 5.1[3])

```
perms x@(a : x') = {y | z ← perms (a : filter (<= a) x'),
                        u ← perms (filter (> a) x'),
                        y ← merge u z}
                 = {y | z2 ← perms (filter (<= a) x'),
                        z ← interleave a z2,
                        u ← perms (filter (> a) x'),
                        y ← merge u z}
```

makes the program well-defined.

A sorting algorithm based on this result is a version of the *Quicksort*.

```
sort [] = {[]}
sort (a : x) = {z ++ [a] ++ z1 | z ← sort (filter (<= a) x),
                                 z1 ← sort (filter (> a) x)}
```

5.2 Cycles of a Graph

Our second example is a practical one and the transformation is somewhat lengthy. The problem is to compute all the cycles in a given directed graph. The graph is given as a list of vertices together with a predicate r. $r\ a\ b$ holds just in case (a, b) is a directed arc of a graph. For simplicity, assume that $r\ a\ a$ never holds. A cycle is a connected path of two or more distinct vertices in which the last is connected to the first. Here is our specification.

$$\begin{aligned}
&Cycs\ x && = \{y \mid y \in Seqs\ x;\ cyclic\ y\} \\
&cyclic\ x && = ((x \neq [\,]) \wedge path\ x \wedge r\ (last\ x)\ (head\ x)) \\
&path\ x && = \forall i(1 \leq i \leq length\ x - 1 \rightarrow r\ (x!!i)\ (x!!(i+1))) \\
&Seqs\ x && = \{y \mid \exists m, z(z = m \diamond x \wedge bag\ y = bag\ z)\} \\
&\diamond && :: ([Bool], [a]) \rightarrow [a] \\
&m \diamond [\,] && = [\,] \\
&[\,] \diamond x && = x \\
&(a : m) \diamond (b : x) && = b : (m \diamond x) \quad \text{if } a \\
& && = m \diamond x \quad \text{otherwise}
\end{aligned}$$

The continuity condition of *Cycs* is

$$y \in Seqs\ x \wedge cyclic\ y \rightarrow (f\ x\ y) \in Seqs\ (h\ x\ y) \wedge cyclic\ (f\ x\ y)$$

Although it is difficult to find a function f which satisfies

$$cyclic\ y \rightarrow cyclic\ (f\ x\ y).$$

it is an easy task to show that a slightly different condition

$$path\ x \rightarrow path\ (tail\ x)$$

holds. It means that the continuity condition is satisfiable in the case of *Seqs* and *path*. Hence we define another specification

$$Seqq\ x = \{y \mid y \in Seqs\ x;\ path\ y\}$$

whose predicate is a little weaker than that of *Cycs* x. We can select f and h for this specification:

$$\begin{array}{lll} f\ x\ y = tail\ y, & & \text{if } y \neq [] \\ \quad = [], & & \text{otherwise} \\ h\ x\ y = x\backslash\backslash[head\ y], & & \text{if } y \neq [] \\ \quad = [], & & \text{otherwise} \end{array}$$

Through Step 1 to Step 4 of the transformation strategy, we have

$$Seqq\ x = \{[]\} +\!\!+ \{y \mid \exists z,\ a(y = a : z;\ z \in Seqq\ (x\backslash\backslash[a]);\ y \in Seqs\ x;\ path\ y)\}.$$

Under the condition

$$y = a : z \wedge z \in Seqq\ (x\backslash\backslash[a]),$$

the following relations holds.

$$\begin{array}{l} y \in Seqs\ x \leftrightarrow a \in x \\ path\ y \leftrightarrow (z = []) \vee ((z \neq []) \wedge r\ a\ (head\ z)) \end{array}$$

Hence we can rewrite *Seqq* as

$$Seqq\ x = \{[]\} +\!\!+ \{y \mid \exists z,\ a(y = a : z;\ a \in x;\ z \in Seqq\ (x\backslash\backslash[a]);\ z = [] \vee (z \neq [] \wedge r\ a\ (head\ z)))\}.$$

To avoid redundant computation, we define a new function *Sq* to combine two parts of recursive definition of *Seqq* x. This technique is known as *accumulation*.[1)]

$$\begin{array}{l} Sq\ a\ x = \{z \mid z \in Seqq\ (x\backslash\backslash[a]);\ z = [] \vee (z \neq [] \wedge r\ a\ (head\ z))\} \\ \quad = \{[]\} +\!\!+ \{z \mid z \in Seqq\ (x\backslash\backslash[a]);\ z \neq [];\ r\ a\ (head\ z)\} \end{array}$$

Then *Seqq* x can be expressed as

$$Seqq\ x = \{[]\} +\!\!+ \{y \mid \exists z,\ a(y = a : z;\ a \in x;\ z \in Sq\ a\ x\}.$$

Substituting a with b and x with $x\backslash\backslash[a]$, we have

$$Seqq\ (x\backslash\backslash[a]) = \{[]\} +\!\!+ \{y \mid \exists z,\ b(y = b : z;\ b \in (x\backslash\backslash[a]);\ z \in Sq\ b\ (x\backslash\backslash[a])\}$$

Put this into the definition of $Sq\ a\ x$ we have

$$\begin{aligned} &Sq\ a\ x \\ &\quad = \{[]\} +\!\!+ \{y \mid \exists z,\ b(y = b : z;\ b \in filter\ (r\ a)\ (x\backslash\backslash[a]);\\ &\qquad\qquad z \in Sq\ b\ (x\backslash\backslash[a])\}. \end{aligned}$$

Putting all the functions together, we obtain

$$\begin{aligned} Cycs\ [] &= \{\} \\ Cycs\ x &= \{y \mid y \in Seqs\ x;\ cyclic\ y\} \\ &= \{y \mid y \in Seqq\ x;\ y \neq [];\ r\ (last\ y)\ (head\ y)\} \\ Seqq\ [] &= \{[]\} \\ Seqq\ x &= \{[]\} +\!\!+ \{a : z \mid \exists z,\ a(a \in x;\ z \in Sq\ a\ x)\} \\ Sq\ a\ [] &= \{[]\} \\ Sq\ a\ x &= \{[]\} +\!\!+ \{b : z \mid \exists z,\ a(b \in filter\ (f\ a)\ (x\backslash\backslash[a]);\\ &\qquad\qquad z \in Sq\ b\ (x\backslash\backslash[a])\}. \end{aligned}$$

The algorithm becomes as follows.

```
cycs [] = {[]}
cycs x@(_ : _) = {y | y ← seqq x, y/=[], r (last y) (head y)}

seqq [] = {[]}
seqq x@(_ : _) = {[]} ++ {a : z | a ← x, z ← sq a x}

sq a [] = {[]}
sq a x@(_ : _) = {[]} ++ {b : z | b ← filter (r a) (x\\[a]),
                          z ← sq b (x\\[a])}
```

§6 Conclusion

We believe that there is a common principle which leads us to synthesize an algorithm from a given specification. This may help us not to fall into ad hoc programming.

In Ref. 1), a specification is given in the form of *generation and test*, where the generator is defined in terms of recursive equation. Such a specification lays restriction on the possibility of derivations. For example, the sorting algorithms are almost decided by the recursive pattern of its generator *perms x*. Once the recursive definition of *perms* is given, it can hardly be transformed into another recursive pattern. That is, if the specification is defined as the form of *Insertion Sort*, then it can hardly be transformed into a *Selection Sort* algorithm.

The work by Nogi[8] also tries to derive algorithms from set-theoretic formulas. But his specifications are given in the form of *generation and test* (although the definition of generators are not limited to recursion ones). In our approach, we use solely predicates in set-theoretic specification, which enables us to be free of verifying the uniformity between recursively defined generators and predicates. It is not a trivial task to ensure that recursive definitions for the generator and the predicate are consistent.

We have shown that recursion by filtering comes out in addition to well-known structural recursion in the same framework based on a common continuity condition. And it is worth noting that our continuity condition allows the function h taking both input and output variables. This makes our transformation method applicable to a wider class of problems. We have synthesized various sorting algorithms from a common specification as in Ref. 4) and reported in Ref. 10).

References

1) Bird, R. S., "The Promotion and Accumulation Strategies in Transformational Programming," *ACM Trans. Prog. Lang. Syst.*, *6*, *2*, pp. 487-504, 1984.
2) Bird, R. S., *Introduction to Functional Programming*, Prentice Hall, 1988.
3) Bird, R. S., "An Introduction to the Theory of Lists," in *Logic of Programming and Calculi of Discrete Design* (M. Broy, ed.), *NATO ASI Series F*: *Vol. 36*, Springer-Verlag, 1987.
4) Darlington, J., "A Synthesis of Several Sorting Algorithms," *Acta Informatica*, *11*, *1*, pp. 1-30, 1978.
5) Drake, F. R., *Set Theory*, North-Holland, 1974.
6) Hudak, P. et al., "Report on the Programming Language Haskell (Version 1.1)," *ACM SIGPLAN Notices*, *27*, *5*, 1992.
7) Manna, Z., *The Logical Basis for Computer Programming*, Addison-Wesley, 1985.
8) Nogi, K., "Derivation of Algorithms Based on Structural Induction," *Computer Software*, *7*, *4* pp. 39-59, 1990. [in Japanese]
9) Turner, D. A., "Functional Programs as Executable Specifications," in *Mathematical Logic and Programming Languages* (C. A. R. Hoare and J. C. Shepherdson, eds.), Prentice Hall, 1985.
10) Xu, L., "Derivation of Algorithms by Introducing Generation Functions," *Master's thesis*, Information Engineering Course, University of Tokyo, Japan, 1992.

Liangwei Xu: He is currently a graduate student of Doctor Course of Division of Engineering, The University of Tokyo. He received the B. E. degree from Shanghai Jiao Tong University (P. R. China) in 1982 and the M. E. degree from The University of Tokyo in 1992. His research interests are computational model, program transformation and derivation methodology.

Hideya Iwasaki, Dr. Eng.: He is an Associate Professor of Educational Computer Center, The University of Tokyo. He received the M. E. degree in 1985, the Dr. Eng. degree in 1988 from The University of Tokyo. He has been working on list processing languages, functional languages, and parallel processing.

Masato Takeichi, Dr. Eng.: He is a Professor of Department of Mathematical Engineering, Faculty of Engineering, The University of Tokyo. His research interests are functional programming, language implementation and constructive algorithmics.