Logical Specifications for Functional Programs[⋆]

Theodore S. Norvell and Eric C.R. Hehner
norvell@cs.utoronto.ca hehner@cs.utoronto.ca

Department of Computer Science
University of Toronto

Abstract. We present a formal method of functional program development based on step-by-step transformation.
In their most abstract form, specifications are essentially predicates that relate the result of the specified program to the free variables of that program. In their most concrete form, specifications are simply programs in a functional programming language. Development from abstract specifications to programs is calculational.
Using logic in the specification language has many advantages. Importantly it allows nondeterministic specifications to be given, and thus does not force overspecification.

0 Introduction

A great deal of research has focused on transforming functional programs into equivalent functional programs. The original program can be considered to be an executable specification.

In this paper we wish to consider not only executable specifications, but also implicit specifications that relate the input and result of a functional program in ways that give no indication of any practical way to compute the result. Such a specification can be more abstract and more declarative than an executable specification.

We take the following point of view, applicable to programming in imperative, functional, or any other kind of language: Specifications describe those *observations* that are acceptable and programs are one sort of specification. A specification x can be refined to another specification y if and only if x describes every observation y describes. Within such a framework, nondeterminism presents no difficulty and the validity of refinement is a very simple relationship. In the case of functional (expression) programming each observation consists of the state in which an expression is evaluated and and a value for the whole expression.

To describe acceptable observations, various notations can be used. Common notations include predicate calculus and set notation. Neither of these is satisfactory for expressions, as they disagree with existing notation for deterministic

[⋆] Published in LNCS 669, Bird, Morgan, and Woodcock editors, pp. ??–??, Springer-Verlag, 1993.

expressions. Instead we use a calculus of nondeterministic expressions known as bunch theory. Implicit specifications written in predicate calculus fit well into this calculus.

0.0 The Structure of the Paper

The structure of this paper is as follows. Section 1 presents a theory of nondeterministic expressions that is used throughout the rest of the paper. Section 2 introduces a functional programming language and a specification language. The programming language will be a subset of the specification language. Examples of using this specification language are given in Sect. 3. In Sect. 4 the relation of *refinement* is introduced. A specification y refines a specification x (written $x \sqsupseteq y$) iff every way that y can be satisfied also satisfies x. A program is a specification that can be executed with acceptable efficiency and so needs no further refinement. Section 5 presents a number of theorems that are of help in proving the refinement relation. Section 6 shows how these theorems can be used to derive programs from specifications by a number of small and formally justified steps. Higher order functions are discussed in Sect. 7. Section 8 presents a method of specifying time bounds. In Sect. 9 we look at pattern matching. Finally Sect. 10 discusses related research.

1 Nondeterministic Expressions

We generalize the notion of expression to allow "don't care" nondeterminism (also known as "erratic" nondeterminism). Our generalized expressions are known as *bunch expressions* (Hehner 1984).

Given expressions x and y, the expression $x{,}y$ called the *bunch union* of x and y denotes a value that could be the value of x or could be the value of y. Bunch union is associative, commutative, and idempotent. Ordinary operators distribute over bunch union. For example, the following three expressions are equivalent

$$\begin{gathered}(1,4)+(5,2)\\(1+5),(1+2),(4+5),(4+2)\\3,6,9\end{gathered}$$

The identity of bunch union is written as *null*. It represents the empty bunch. At the opposite end of the spectrum is *all*, representing the union of all expressions.

A bunch x is a *subbunch* of a bunch y if and only if there is a bunch z such that $x{,}z$ is equivalent to y. We write $x : y$. This is a partial order. For all bunch expressions x, we have $null : x$. Equality of bunch expressions will be written as $x \equiv y$, meaning $x : y$ and $y : x$. Ordinary equality is written $x = y$ and differs from $x \equiv y$ in that it distributes over bunch union. Thus

$$(1,2)=2 \quad \equiv \quad \mathbf{true},\mathbf{false}$$

whereas

$$1,2 \quad \not\equiv \quad 2$$

Certain bunch expressions will be called *elements*. Each number is an element as are the constants **true** and **false**. A list of elements is an element. Which functional values are elements will be discussed later. We say e is an element of x if $e : x$ and e is an element.

Bunches may be considered as sets but without the nesting (sets of sets), using a simpler notation (no curly braces), and with distribution of operations over the elements. The main reasons for using bunches rather than sets are notational convenience, and that they specialize properly to deterministic values, whereas sets do not.

2 The Specification Language

Our specification language will be an extension to a simple functional programming language. The expressions of the specification language are bunch expressions.

Specifications may contain free variables. These represent the input to the expression, *i.e.* the state in which it is evaluated. Each variable represents an element.

As a simple example of a specification, $n + 1$ is the specification of a number one greater than state variable n. It happens that this specification is also a program. By using bunch expressions, we allow for choice in the specification. A specification of a number that is one, two, or three greater than n is $n + (1, 2, 3)$.

2.0 The Programming Language Subset

For this paper, we will use the simple language illustrated in Fig. 0. Expressions in this language will be called *programs* to distinguish them from more general specifications.

Types The types of this language are bunches. The bunch *bool* has elements **true** and **false**. The bunch *nat* has elements 0, 1, 2, and so on. A subrange of the naturals is written $i,..j$ for naturals i and j. This subrange includes i but excludes j. Given a type T, the type T^* is the bunch of all finite lists with items (list members) in T. An elementary list is one whose items are all elements.

Expressions The usual boolean and numerical operators are provided, as well as a standard if-expression.

Lists are written in square brackets with semicolons separating the items. A useful notation forms a list of contiguous naturals; $[i;..j]$ begins with i and continues up to (but not including) j. Lists may be catenated using $^+$. List indexing is written as juxtaposition and lists are indexed from 0. A list may be indexed by a list, producing a list of results.

Types		
Naturals	nat	$0, 1, 2, \ldots$
Subranges	$9,..\,12$	$9, 10, 11$
Booleans	$bool$	**true**, **false**
Lists	nat^*	$[\,], [0], [0; 0], \ldots$

Expressions		
Numerical Expressions	$1 + 5$	6
Boolean Expressions	$1 = 5$	**false**
Conditionals	**if** $1 = 5$ **then** 3 **else** 4	4
Lists	$[1; 2; 3; 4]$	$[1; 2; 3; 4]$
Lists	$[9;..\,12]$	$[9; 10; 11]$
List catenation	$[1; 2] \mathbin{^+} [3; 4]$	$[1; 2; 3; 4]$
List indexing	$[1; 2; 3; 4]\ 2$	3
List indexing	$[1; 2; 3; 4]\ [3; 2; 1; 0]$	$[4; 3; 2; 1]$
List length	$\#[1; 2; 3; 4]$	4
Functions	$\lambda m : nat^* \succ\!- \lambda i : nat \succ\!- m\ [0;..\,i]$	
Application	$(\lambda m : nat^* \succ\!- \lambda i : nat \succ\!- m\ [0;..\,i])\ [1; 2; 3; 4]\ 2$	$[1;2]$
Let	**let** $i = 3 \rightarrow [1; 2; 3; 4]\ [0;..\,i]$	$[1; 2; 3]$

Fig. 0. A simple functional programming language

Functions and let-expressions introduce new identifiers which may be used within their bodies. Function application is written as juxtaposition.

Notably absent from this language is any form of recursive definition. Recursion is treated in section 4.1.

Semantics The formal semantics of the operators of the programming language can be given axiomatically. A listing of all the axioms would be rather long. We list a few as examples.

$$\begin{aligned} \textbf{if true then } x \textbf{ else } y &\equiv x \\ \textbf{if false then } x \textbf{ else } y &\equiv y \\ \#[\,] &\equiv 0 \\ \#[x] &\equiv 1 \\ \#(x \mathbin{^+} y) &\equiv \#x + \#y \\ &\vdots \end{aligned}$$

The treatment of errors (for example division by 0) is a matter of some choice. We can treat errors as equivalent to *all*, or we can omit axioms that allow us to reason about erroneous computations. Either way of treating errors is consistent with the rest of this paper.

2.1 Specification Language Extensions

The programming language presented so far can be used to write executable specifications which can then be transformed to more efficient programs using conventional techniques.

Instead of stopping at an executable specification language, we will allow any bunch expression to be used as a specification. In this section, we present a number of constructs that are of use in writing specifications. They extend the programming language to allow greater ease and range of expression.

In this section, P and Q will stand for first order predicates, e for an element, x, y, and z for specifications, and i for an identifier.

Until Sect. 7 we will only consider elements that are first order, that is numbers, booleans, and lists of first order elements. Functional elements will be discussed in Sect. 7.

Predicates are boolean expressions. However the nondeterminism of the specification language is not extended to the predicates. For example $i \leq (2,3)$ is not an acceptable predicate because, in any state where $i = 3$, it is equivalent to $(\mathbf{true}, \mathbf{false})$. In each state, a predicate must be either true or false, never both, never neither (though the logic may not be complete enough to say which).

Programs Any program is also a specification. Furthermore, any way of constructing programs from programs can be used to construct specifications from specifications. Thus

$$[x; y]$$

and

$$\mathbf{if}\ P\ \mathbf{then}\ x\ \mathbf{else}\ y$$

are both specifications provided that P is a predicate and x and y are specifications, even though P, x, and y may not be programs.

Solutions The expression $\S i \cdot P$ is equivalent to the bunch of all elements i for which P is true. For example, $\S i \cdot i : nat \wedge i < 3$ is the bunch $0, 1, 2$. The axiom for this quantifier is:

$$(e : \S i \cdot P) \quad = \quad (\text{Substitute } e \text{ for } i \text{ everywhere in } P)$$

with the usual caveats for substitution.

Null The specification $null$ refines all specifications. This specification is not satisfied by any result. The axiom for $null$ is

$$null \quad \equiv \quad \S i \cdot \mathbf{false}$$

In imperative programming, the corresponding specification is that which has, as its weakest precondition predicate transformer, $\lambda R \cdot \mathbf{true}$.

All The specification all is refined by all specifications. It can be used by the specifier to indicate that she doesn't care about the result. The axiom for all is

$$all \quad \equiv \quad \S i \cdot \mathbf{true}$$

This is the bunch of all elements.

Union and Intersection The specification x,y specifies that at least one of specifications x and y must be met. The specification $x`y$ specifies that both x and y must be met. Their axioms are

$$\begin{aligned} x, y &\equiv \S i \cdot (i : x) \vee (i : y) \\ x`y &\equiv \S i \cdot (i : x) \wedge (i : y) \end{aligned}$$

for i not free in x or y.

Assert The specification $P \succ\!\!- x$ expresses that x must be met when P is true, and otherwise any result will do. Its axiom is

$$P \succ\!\!- x \quad \equiv \quad \textbf{if } P \textbf{ then } x \textbf{ else } \mathit{all}$$

In this usage, P is called an *assertion.*

Guard The specification $P \twoheadrightarrow x$ expresses that x must be met when P is true, and is otherwise impossible to meet. Its axiom is

$$P \twoheadrightarrow x \quad \equiv \quad \textbf{if } P \textbf{ then } x \textbf{ else } \mathit{null}$$

In this usage, P is called a *guard.*

Seen as unary operators, $P \succ\!\!-$ and $P \twoheadrightarrow$ are duals and adjoint.

Try The specification **try** x expresses that x must be met if possible. Its axiom is

$$\textbf{try } x \quad \equiv \quad (x \not\equiv \mathit{null}) \succ\!\!- x \quad \equiv \quad \textbf{if } x \not\equiv \mathit{null} \textbf{ then } x \textbf{ else } \mathit{all}$$

The specification **try** x **else** y expresses that x must be met if possible, and if not, y must be met. Its axiom is

$$\textbf{try } x \textbf{ else } y \quad \equiv \quad \textbf{if } x \not\equiv \mathit{null} \textbf{ then } x \textbf{ else } y$$

This construct expresses a kind of backtracking or dynamic exception handling where failure is expressed by *null*.

Unlike our other specification constructs, **try** and **try else** are not monotonic in all their specification operands, with respect to the subbunch ordering.

Lambda The specification language has a more general abstraction operator than the programming language. For identifier i and expression x, the following is an expression

$$\lambda i \cdot x$$

For any element e

$$(\lambda i \cdot x)\, e \quad \equiv \quad (\text{Substitute } e \text{ for } i \text{ everywhere in } x)$$

Furthermore application distributes over bunch formation, so, for example,

$$f\ null \equiv null$$
$$f\ (y, z) \equiv (f\ y), (f\ z)$$

Thus variables always represent elements.

Lambda abstraction is untyped with respect to the programming language types. However, in order to prevent paradoxical expressions, it is typed with respect to the order of the arguments. Until Sect. 7 all arguments will be first order, that is, nonfunctional.

Let Likewise, the specification language has a more general **let** construct. It is defined by

$$\mathbf{let}\ i \cdot x \quad \equiv \quad (\lambda i \cdot x)\ all$$

Typically x is of the form $P \rightarrow y$, in which case it can be seen that $\mathbf{let}\ i \cdot (P \rightarrow y)$ is the union over all elements i such that P is true, of y.

2.2 Syntactic Issues

Precedence The precedence of operators used in this paper will be first juxtaposition (application and indexing) and then in order

				$=$							
				$\neq$					Binders		
$\#$			$,$	$<$				**if then else**			$\equiv$
$*$	$\times$	$+$	$,..$	$>$	$\neg$	$\wedge$	$\vee$	$\Rightarrow$	**try**	$\rightarrow$	$\sqsupseteq$
$\not{c}$	$/$	$-$	‘	$\leq$				**try else**	$\succ\!-$	$\sqsubseteq$
				$\geq$						$\gg$	
				$:$							

Binders (λ, **let**, §, $\forall$, $\exists$), $\rightarrow$, and $\succ\!-$ are right associative, so that we can write, for example,

$$\lambda i \cdot (P \succ\!- (\mathbf{let}\ j \cdot (Q \rightarrow R \succ\!- x)))$$

as

$$\lambda i \cdot P \succ\!- \mathbf{let}\ j \cdot Q \rightarrow R \succ\!- x$$

Relation operators are *continuing*, so we can write, for example

$$(x \sqsupseteq y) \wedge (y \sqsupseteq z)$$

as

$$x \sqsupseteq y \sqsupseteq z$$

Syntactic Sugar For all binders, we allow the following abbreviation. If the textually first identifier to appear in the body is the bound variable, then the bound variable and the subsequent dot can be omitted. Thus

$$\lambda i \cdot i : T \succ\!\!- x$$

can be written as

$$\lambda i : T \succ\!\!- x$$

and

$$\mathbf{let}\; i \cdot i = e \rightarrow x$$

can be written as

$$\mathbf{let}\; i = e \rightarrow x$$

In programs, we always use the abbreviated notation.

3 Writing Specifications

In this section several examples are given of using the specification language.

We remind the reader that the free variables together represent the state in which the expression is evaluated and thus each free variable represents an element. Restrictions on these variables, i.e. the type of the state, will be stated informally.

An implementation is obliged to give a result described by the specification. Thus *null* is unimplementable. The specification

$$\mathbf{if}\; x = 0 \;\mathbf{then}\; \mathit{null} \;\mathbf{else}\; 1$$

can be satisfied in states such that $x \neq 0$ but not when $x = 0$. Perhaps the specifier has no intention of providing a state for which the specification is *null*, but to the implementor every input is a possibility. A specification is called *implementable* if there is no state in which it is equivalent to *null*.

3.0 Searching

Suppose that L is a list variable of a type T^* and x is a variable of type T. Informally, we need to find an index of an item x of a list L. A first attempt at formally specifying this is

$$\S i \cdot L\, i = x$$

This says that we want any i such that $L\, i = x$. However, x may not occur in L at all. For such a case, the above specification is *null*, and so the specification is unimplementable. Suppose that we intend to use the specification only when x occurs in the list. Then we don't care what the result would be if x did not occur, and the specification should be

$$(\exists i \cdot L\, i = x) \succ\!\!- (\S i \cdot L\, i = x)$$

This is still not entirely satisfactory if it is not guaranteed by the axioms concerning lists that $L\ i = x$ is false for values of i that are not valid indices of L. The next specification covers this situation

$$(\exists i : 0,..\ \#L \wedge L\ i = x) \succ\!\!- (\S i : 0,..\ \#L \wedge L\ i = x)$$

(Note the use of the syntactic sugar from Sect. 2.2.) The **try** operator can be used to make this more concise:

$$\textbf{try}\ (\S i : 0,..\ \#L \wedge L\ i = x)$$

It is noteworthy that this is a nondeterministic problem. When x appears more than once in the list, the result can be any suitable index. A deterministic specification language would necessitate overspecification.

3.1 Fermat's Last Theorem

Quite often an informal search specification will be of the form "if there is an x such that $P\ x$, then $f\ x$, else y". The **if then else** construct can not be used to formalize this as x will not be available in the then-part. A solution is to use the **try else** construct. For example, the following specification is [ ] if Fermat's Last Theorem is true and is some counterexample otherwise.

$$\begin{array}{l} \textbf{try}\ (\textbf{let}\ n : nat + 3 \to \textbf{let}\ i : nat \to \textbf{let}\ j : nat \to \textbf{let}\ k : nat \to \\ \qquad i^n + j^n = k^n \to [n; i; j; k]\ ) \\ \textbf{else}\ [\,] \end{array}$$

3.2 Sorting

Suppose that $\leq$ is a relation, on a type T, that is reflexive, transitive, and total (that is, for all x and y in T, either $x \leq y$ or $y \leq x$). We wish to specify that, given a list, we want a permutation of it that is sorted with respect to this relation. We will present two equivalent specifications to illustrate the range of styles that the specification language permits.

A Logic Oriented Specification The first specification is more logic oriented. It proceeds by defining a predicate describing the desired relationship between the input and output of the program. First we define a function that returns the number of times an item occurs in a list.

$$count\ L\ x \quad \overset{\text{def}}{\equiv} \quad ¢\S j : 0,..\ \#L \wedge x = L\ j$$

This uses the counting operator ¢ that gives the number of elements in a bunch. Now we define what it is for one list to be a permutation of another

$$Perm\ L\ M \quad \overset{\text{def}}{\equiv} \quad \forall x \cdot count\ L\ x = count\ M\ x$$

Next is a predicate that indicates a list is monotone

$$\mathit{Mono}\ M \quad \overset{\text{def}}{\equiv} \quad \forall j : 1,..\,\#M \Rightarrow M\,(j-1) \leq M\ j$$

The final predicate states that one list is a sorted permutation of another

$$\mathit{Sortof}\ L\ M \quad \overset{\text{def}}{\equiv} \quad \mathit{Perm}\ L\ M \wedge \mathit{Mono}\ M$$

Finally this predicate is used to form the specification:

$$\mathit{sort} \quad \overset{\text{def}}{\equiv} \quad \lambda L : T^* \succ\!\!- \S M : T^* \wedge \mathit{Sortof}\ L\ M$$

An Expression Oriented Specification The second specification is more expression oriented. First we define a permutation function as the smallest function satisfying

$$\begin{aligned}\mathit{perm} \quad \equiv \quad & \lambda L{:}T^* \succ\!\!- L, (\,\mathbf{let}\ M : \mathit{perm}\ L \rightarrow \mathbf{let}\ i \cdot \mathbf{let}\ j \cdot 0 \leq i < j < \#M \rightarrow \\ & \qquad M[0;..\,i]\ {}^+\ [M\ j]\ {}^+\ M[i+1;..\,j]\ {}^+\ [M\ i]\ {}^+\ M[j+1;..\,\#M]\,)\end{aligned}$$

(The meaning of "smallest function" will be explained in section 4.2.) This function nondeterministically returns any permutation of its argument. Next we define the bunch of all ordered lists over T as the smallest bunch satisfying

$$\mathit{ordered} \quad \equiv \quad [\,], (\mathbf{let}\, M{:}\,\mathit{ordered} \rightarrow \mathbf{let}\, t{:}T \wedge (\forall i{:}\,0,..\,\#M \Rightarrow t{\leq}M\,i) \rightarrow [t]^+\,M)$$

Finally one can specify *sort* as

$$\mathit{sort} \quad \overset{\text{def}}{\equiv} \quad \lambda L : T^* \succ\!\!- \mathit{ordered}\ \text{‘}\ \mathit{perm}\ L$$

4 Refinement

4.0 The Refinement Relation

We define the refinement relation $x \sqsupseteq y$ to mean that $y : x$ universally. By "universally" we mean in all states, that is for all assignments of elements to the free variables of expressions x and y. We say x *is refined by* y. For example, that 1,2 is refined by 1 is written

$$1, 2 \quad \sqsupseteq \quad 1$$

For another example,

$$n : \mathit{nat} \succ\!\!- n + (1, 2) \quad \sqsupseteq \quad n + 1$$

The refinement relation is a partial order on specifications.

Programming from a specification x is the finding of a program y such that $x \sqsupseteq y$. To simplify this process, we find a sequence of specifications $x_0 \cdots x_n$ where x_0 is x and x_n is y, and where $x_i \sqsupseteq x_{i+1}$ is a fairly trivial theorem, for each i. This is a formalization of the process of stepwise refinement.

Note that some authors write $x \sqsubseteq y$ for refinement where we write $x \sqsupseteq y$. Perhaps they believe that "bigger is better," but we find the analogy with standard set notation ($\supseteq$) too strong to resist.

4.1 Programming with Refined Specifications

At this point we can add one final construct to the programming language. Any specification x can be considered to be a program provided a program y is supplied such that $x \sqsupseteq y$. We can think of x as a subprogram name and of y as its subprogram body.

Recursion and mutual recursion are allowed. Since $\sqsupseteq$ is reflexive, it is always possible to refine x with x itself. This leads to correct programs, but ones that take an infinite amount of time to execute. This will be discussed further in Sect. 8.

A programming notation for recursion could be defined, but we have chosen not to do so.

4.2 Function Refinement

Because we wish to speak of refinement of functions, we must extend the sub-bunch relation to functions. This is done by defining

$$(\lambda i \cdot y) \quad : \quad (\lambda i \cdot x)$$

if for all elementary i,

$$y \quad : \quad x$$

Thus if $x \sqsupseteq y$, then $\lambda i \cdot x \sqsupseteq \lambda i \cdot y$.

5 Laws of Programming

In this section we will present a number of theorems that can be used to prove refinement relations. Numerous other theorems could be presented; this is a selection of those most useful for developing programs.

Some of the following laws show mutual refinement, that is both $x \sqsupseteq y$ and $y \sqsupseteq x$; we will use $x \underline{\square} y$ to show this. Some of the following laws apply to both assertions and guards; we will use $\gg$ to mean one of $\succ\!\!-$ or $\twoheadrightarrow$. That is, the laws where $\gg$ appears (even if more than once) each abbreviate exactly two laws, one for $\succ\!\!-$ and one for $\twoheadrightarrow$.

Union elimination: $x, y \sqsupseteq x$
If introduction/elimination: $x \underline{\square}$ **if** P **then** x **else** x
Case analysis:

$$\textbf{if } P \textbf{ then } x \textbf{ else } y \quad \underline{\square} \quad \textbf{if } P \textbf{ then } (P \gg x) \textbf{ else } y$$

$$\textbf{if } P \textbf{ then } x \textbf{ else } y \quad \underline{\square} \quad \textbf{if } P \textbf{ then } x \textbf{ else } (\neg P \gg y)$$

Let introduction/elimination: *If i is not free in x, then*

$$x \quad \underline{\square} \quad \textbf{let } i \cdot x$$

The example law for let : *If e is an element and* (Substitute e for i everywhere in P), *then*

$$\textbf{let}\ i \cdot P \to x \quad \sqsupseteq \quad \textbf{let}\ i \cdot P \to (\text{Substitute } e \text{ for } i \text{ anywhere in } x)$$

The example law for §: *If e is an element and* (Substitute e for i everywhere in P), *then*

$$\S i \cdot P \quad \sqsupseteq \quad e$$

Guard introduction: $x \sqsupseteq P \to x$
Assertion elimination: $P \rightarrowtail x \sqsupseteq x$
Guard strengthening: *If* $(Q \Rightarrow P)$ *universally, then* $P \to x \sqsupseteq Q \to x$
Assertion weakening: *If* $(Q \Rightarrow P)$ *universally, then* $Q \rightarrowtail x \sqsupseteq P \rightarrowtail x$
Assertion/guard use: *If* $(P \Rightarrow y : x)$ *universally, then* $P \gg x \sqsupseteq P \gg y$
Assertion/guard combining/splitting: $P \gg Q \gg x \quad \sqsupseteq\sqsubseteq \quad P \wedge Q \gg x$
Adjunction: $(P \rightarrowtail x \sqsupseteq y) = (x \sqsupseteq P \to y)$
One point: *If e is an element,*

$$i = e \gg x \quad \sqsupseteq\sqsubseteq \quad i = e \gg (\text{Substitute } e \text{ for } i \text{ anywhere in } x)$$

Application introduction/elimination: *If $\lambda i \cdot x$ distributes over bunch union, then*

$$(\lambda i \cdot x)\, y \quad \sqsupseteq\sqsubseteq \quad (\text{Substitute } y \text{ for } i \text{ everywhere in } x)$$

Lambda introduction: *If $x \sqsupseteq y$ then $\lambda i \cdot x \sqsupseteq \lambda i \cdot y$*

There are a great many laws for moving assertions and guards. Inward movement laws say that assertions and guards that apply to a specification apply to any part of the specification. For example,

An example inward movement law: $P \rightarrowtail x, y \quad \sqsupseteq \quad P \rightarrowtail (P \rightarrowtail x), y$

Outward movement laws say that assertions and guards that apply to all parts of a specification apply to the whole specification.

An example outward movement law: *If i is not free in P, then*

$$\textbf{let}\ i \cdot P \to x \quad \sqsupseteq \quad P \to \textbf{let}\ i \cdot x$$

Except for **try** and **try else**, all the operators we have introduced that form specifications from specification operands are monotonic in those operands, with respect to the refinement relation. This gives rise to a number of monotonicity laws that will be used implicitly. For example,

An example monotonicity law: *If $x \sqsupseteq y$, then*

$$\textbf{if}\ P\ \textbf{then}\ x\ \textbf{else}\ z \quad \sqsupseteq \quad \textbf{if}\ P\ \textbf{then}\ y\ \textbf{else}\ z$$

Monotonicity laws allow application of the other laws deep within the structure of a specification.

6 Deriving Programs

In this section, we demonstrate a programming methodology based on the refinement relation.

6.0 Searching

Our searching specification from Sect. 3.0 was

$$(\exists i : 0,..\#L \wedge L\ i = x) \succ (\S i : 0,..\#L \wedge L\ i = x)$$

We add a parameter j so we can specify searching in the part of list L preceding index j

$$\mathit{search\_before} \quad \overset{\text{def}}{\equiv} \quad \lambda j{:}1,..1{+}\#L \succ (\exists i{:}0,..j \wedge L i{=}x) \succ (\S i{:}0,..\#L \wedge L i{=}x)$$

The original specification is refined by

$$\mathit{search\_before}\ (\#L)$$

It remains to supply a program that refines *search_before*. Let j represent any element of type $1,..1{+}\#L$. We start by refining *search_before* j. (Note that hints appear *between* the two specifications they apply to.)

```
    $\mathit{search\_before}\ j$                         **if** introduction and case analysis
$\sqsupseteq$   **if** $L\,(j-1) = x$ **then** $(L\,(j-1) = x \succ \mathit{search\_before}\ j)$
    **else** $(L\,(j-1) \neq x \succ \mathit{search\_before}\ j)$
Definition of *search_before*, assertion combining, and assertion weakening
$\sqsupseteq$   **if** $L\,(j-1) = x$ **then** $(L\,(j-1) = x \succ (\S i : 0,..\#L \wedge L\ i = x))$
    **else** $(L\,(j-1) \neq x \wedge (\exists i : 0,..j \wedge L\ i = x) \succ (\S i : 0,..\#L \wedge L\ i = x))$
Assertion use, example law, and assertion elimination in the then-part
Logic and assertion weakening in the else-part
$\sqsupseteq$   **if** $L\,(j-1) = x$ **then** $j-1$
    **else** $((\exists i : 0,..j{-}1 \wedge L\ i = x) \succ (\S i : 0,..\#L \wedge L\ i = x))$
If there exists an $i$ in $0,..j{-}1$, then $j > 1$
$\sqsupseteq$   **if** $L\,(j-1) = x$ **then** $j-1$
    **else** $((\exists i : 0,..j{-}1 \wedge L\ i = x) \wedge j : 2,..1{+}\#L \succ$
        $(\S i : 0,..\#L \wedge L\ i = x)\ )$
Assertion weakening and assertion splitting
$\sqsupseteq$   **if** $L\,(j-1) = x$ **then** $j-1$
    **else** $((j-1) : 1,..1{+}\#L \succ$
        $(\exists i : 0,..j{-}1 \wedge L\ i = x) \succ$
        $(\S i : 0,..\#L \wedge L\ i = x)\ )$
Definition of *search_before* and application introduction
$\sqsupseteq$   **if** $L\,(j-1) = x$ **then** $j-1$
    **else** $\mathit{search\_before}\ (j-1)$
```

We can add to both sides the range assertion on j and then use the function refinement law of Sect. 4.2 (lambda introduction). This gives us

$$\begin{array}{ll} & \mathit{search\_before} \\ \sqsupseteq & \lambda j : 1,..\, 1{+}\#L \succ \textbf{if } L\,(j-1) = x \textbf{ then } j-1 \\ & \qquad\qquad\qquad \textbf{else } \mathit{search\_before}\,(j-1) \end{array}$$

6.1 Sorting

As a second example of deriving programs we derive a merge sort program from the sorting specification given in Sect. 3.2.

$$\begin{array}{lll} & \mathit{sort}\ L & \text{Definition} \\ \sqsupseteq & \S M \cdot \mathit{Sortof}\ L\ M & \textbf{if} \text{ introduction and case analysis} \\ \sqsupseteq & \textbf{if } \#L \le 1 & \\ & \textbf{then } (\#L \le 1 \succ \S M \cdot \mathit{Sortof}\ L\ M) & \\ & \textbf{else } (\#L > 1 \succ \S M \cdot \mathit{Sortof}\ L\ M) & \end{array}$$

The then-branch is refined as follows

$$\begin{array}{lll} & \#L \le 1 \succ \S M \cdot \mathit{Sortof}\ L\ M & \text{Assertion use and elimination} \\ \sqsupseteq & \S M \cdot \#L \le 1 \Rightarrow \mathit{Sortof}\ L\ M & \text{Example} \\ \sqsupseteq & L & \end{array}$$

We now refine the else-branch. The first idea is to divide and conquer.

$$\begin{array}{lll} & \#L > 1 \succ \S M \cdot \mathit{Sortof}\ L\ M & \\ \multicolumn{3}{l}{\text{Let introduction and assertion elimination}} \\ \sqsupseteq & \textbf{let } T \cdot \textbf{let } U \cdot \S M \cdot \mathit{Sortof}\ L\ M & \text{Guard introduction} \\ \sqsupseteq & \textbf{let } T \cdot \textbf{let } U \cdot L = T \mathbin{{}^+} U \rightarrow \S M \cdot \mathit{Sortof}\ L\ M & \text{One point} \\ \sqsupseteq & \textbf{let } T \cdot \textbf{let } U \cdot L = T \mathbin{{}^+} U \rightarrow \S M \cdot \mathit{Sortof}\ (T \mathbin{{}^+} U)\ M & \end{array}$$

We break off the derivation at this point to consider the next move.

Having divided the list, we will sort the two parts. We need to replace the predicate *Sortof* by one in terms of the sorted parts. We call that predicate *Mergeof* and the desired theorem is

$$\mathit{Mergeof}\ (\mathit{sort}\ T)\ (\mathit{sort}\ U)\ M \Rightarrow \mathit{Sortof}\ (T \mathbin{{}^+} U) M$$

One definition that yields this theorem is

$$\mathit{Mergeof}\ T\ U\ M \quad \overset{\text{def}}{\equiv} \quad (\mathit{Mono}\ T \wedge \mathit{Mono}\ U) \Rightarrow \mathit{Sortof}\ (T \mathbin{{}^+} U)\ M$$

Using the theorem we continue the derivation with

$$\sqsupseteq \quad \textbf{let } T \cdot \textbf{let } U \cdot L = T \mathbin{{}^+} U \rightarrow \S M \cdot \mathit{Mergeof}\ (\mathit{sort}\ T)\ (\mathit{sort}\ U)\ M$$

We defer the implementation of *Mergeof* so for now we just define

$$\mathit{merge} \quad \overset{\text{def}}{\equiv} \quad \lambda X : T^* \succ \lambda Y : T^* \succ \mathit{Mono}\ X \wedge \mathit{Mono}\ Y \succ \S M \cdot \mathit{Mergeof}\ X\ Y\ M$$

So the derivation continues

```
⊒  $\textbf{let}\ T \cdot \textbf{let}\ U \cdot L = T \mathbin{^+} U \rightarrowtail \mathit{merge}\ (\mathit{sort}\ T)\ (\mathit{sort}\ U)$
Guard strengthening
⊒  $\textbf{let}\ T \cdot \textbf{let}\ U \cdot T = L[0;..\ \#L\ \textbf{div}\ 2] \wedge U = L[\#L\ \textbf{div}\ 2;..\ \#L] \rightarrowtail$
       $\mathit{merge}\ (\mathit{sort}\ T)\ (\mathit{sort}\ U)$
Guard movement
⊒  $\textbf{let}\ T = L[0;..\ \#L\ \textbf{div}\ 2] \rightarrowtail$
       $\textbf{let}\ U = L[\#L\ \textbf{div}\ 2;..\ \#L] \rightarrowtail$
       $\mathit{merge}\ (\mathit{sort}\ T)\ (\mathit{sort}\ U)$
```

We now need to refine the *merge* specification. Assuming X and Y of the right types we have

```
       $\mathit{merge}\ X\ Y$                                          Definition
⊒  $\mathit{Mono}\ X \wedge \mathit{Mono}\ Y \succ\!\!- \S M \cdot \mathit{Mergeof}\ X\ Y\ M$
$\textbf{if}$ introduction and case analysis
⊒  $\textbf{if}\ X = [\,]\ \textbf{then}\ (\mathit{Mono}\ Y \wedge X = [\,] \succ\!\!- \S M \cdot \mathit{Mergeof}\ X\ Y\ M)$
       $\textbf{else if}\ Y = [\,]\ \textbf{then}\ (\mathit{Mono}\ X \wedge Y = [\,] \succ\!\!- \S M \cdot \mathit{Mergeof}\ X\ Y\ M)$
       $\textbf{else}\ (\ \mathit{Mono}\ X \wedge \mathit{Mono}\ Y \wedge X \neq [\,] \neq Y \succ\!\!-$
              $\S M \cdot \mathit{Mergeof}\ X\ Y\ M\ )$
Assertion use and example
⊒  $\textbf{if}\ X = [\,]\ \textbf{then}\ Y$                                $\textbf{if}$ introduction
       $\textbf{else if}\ Y = [\,]\ \textbf{then}\ X$
       $\textbf{else}\ (\ \mathit{Mono}\ X \wedge \mathit{Mono}\ Y \wedge X \neq [\,] \neq Y \succ\!\!-$
              $\S M \cdot \mathit{Mergeof}\ X\ Y\ M\ )$
⊒  $\textbf{if}\ X = [\,]\ \textbf{then}\ Y$
       $\textbf{else if}\ Y = [\,]\ \textbf{then}\ X$
       $\textbf{else if}\ X\ 0 \leq Y\ 0\ \textbf{then}\ (\ \mathit{Mono}\ X \wedge \mathit{Mono}\ Y \wedge X \neq [\,] \neq Y \wedge X\ 0 \leq Y\ 0 \succ\!\!-$
                                          $\S M \cdot \mathit{Mergeof}\ X\ Y\ M\ )$
       $\textbf{else}\ (\ \mathit{Mono}\ X \wedge \mathit{Mono}\ Y \wedge X \neq [\,] \neq Y \wedge Y\ 0 \leq X\ 0 \succ\!\!-$
              $\S M \cdot \mathit{Mergeof}\ X\ Y\ M\ )$
Assertion use, example, and definition of $\mathit{merge}$
⊒  $\textbf{if}\ X = [\,]\ \textbf{then}\ Y$
       $\textbf{else if}\ Y = [\,]\ \textbf{then}\ X$
       $\textbf{else if}\ X\ 0 \leq Y\ 0\ \textbf{then}\ [X\ 0] \mathbin{^+} \mathit{merge}\ (X[1;..\ \#X])\ Y$
       $\textbf{else}\ [Y\ 0] \mathbin{^+} \mathit{merge}\ X\ (Y[1;..\ \#Y])$
```

Summarizing the above, we have proven

```
       $\mathit{sort}$
⊒  $\lambda L : T^* \succ\!\!- \textbf{if}\ \#L \leq 1$
                  $\textbf{then}\ L$
                  $\textbf{else}\ (\ \textbf{let}\ T = L[0;..\ \#L\ \textbf{div}\ 2] \rightarrowtail$
                         $\textbf{let}\ U = L[\#L\ \textbf{div}\ 2;..\ \#L] \rightarrowtail$
                         $\mathit{merge}\ (\mathit{sort}\ T)\ (\mathit{sort}\ U)\ )$
```

and

```
       $\mathit{merge}$
```

```
⊒   $\lambda X : T^* \succ\!\!- \lambda Y : T^* \succ\!\!-$
    **if** $X = [\,]$ **then** $Y$
    **else if** $Y = [\,]$ **then** $X$
    **else if** $X\,0 \le Y\,0$ **then** $[X\,0]\ ^+\ merge\ (X[1;..\#X])\ Y$
    **else** $[Y\,0]\ ^+\ merge\ X\ (Y[1;..\#Y])$
```

7 Higher Order Programming

In Sect. 4.2 we extended the subbunch relation to functions. This allows one to develop functions that have functional results. For example:

$$\begin{array}{ll} & \lambda i : nat \succ\!\!- \lambda j : nat \succ\!\!- i + j + (0, 1, 2) \\ \sqsupseteq & \lambda i : nat \succ\!\!- \lambda j : nat \succ\!\!- i + j + 1 \end{array}$$

We are not yet ready to develop functions that have functional parameters.

Recall that parameters always represent elements. We extend the notion of elementhood to functions before talking about passing functions as arguments.

In order to avoid circularity in the definition of "element" and to preclude paradoxical expressions, we impose a simple type system on bunches (Church 1940). Expressions containing elements of primitive types such as *bool*, *nat*, and lists of such, we say are of type ι. A lambda expression is written $\lambda i_m \cdot x$ where m is a type. If x is of type n, $\lambda i_m \cdot x$ is of type $m \mapsto n$. In determining the type of the body x or any expression within it, it is assumed that i has type m. A function of type $m \mapsto n$, can be applied only to arguments of type m; the type of the application is n.

We say that a lambda expression $\lambda i_m \cdot x$ is an element iff for each element e of type n, $(\lambda i_m \cdot x)e$ is an element. For example, the elements of $\lambda i_\iota \cdot 1, 2$ are $\lambda i_\iota \cdot 1$ and $\lambda i_\iota \cdot 2$.

This definition has the interesting, but not problematic, consequence that there are non-*null* functions that are proper subbunches of elements. For example,

$$(\lambda i_\iota \cdot i = 0 \Rightarrow 0) \quad : \quad (\lambda i_\iota \cdot 0)$$

To avoid cluttering specifications, including programs, with subscripts, we adopt the following convention:

$$\lambda i : x \succ\!\!- z$$

abbreviates

$$\lambda i_m \cdot i : x \succ\!\!- z$$

and

$$\lambda i\ x : y \succ\!\!- z$$

abbreviates

$$\lambda i_{m \mapsto n} \cdot i\ x : y \succ\!\!- z$$

where x is of type m and y is of type n. Similarly for functions of more arguments. This makes sense because a function i that maps elements of x to elements of y is accurately described by the predicate $i\ x : y$.

The definition of application is the same for functional parameters as for non-functional parameters. That is, it is the union over all substitutions of elements of the argument for the parameter.

Let us look at how definitions of application and elementhood affect higher order functions. Suppose we have a higher order function *map* defined by

$$\lambda f\, nat : nat \succ\!\!- \lambda L : nat^* \succ\!\!- \S M : nat^* \wedge \#M = \#L \wedge (\forall i : 0,.. \#L \Rightarrow M\, i = f\,(L\, i))$$

then the application

$$map\ (\lambda i : nat \succ\!\!- i + (1, 2))\ [0; 0]$$

is equivalent to

$$[1; 1], [2; 2]$$

This is perhaps a somewhat surprising consequence, but the alternative of allowing parameters to represent nondeterministic functions has serious pitfalls (see (Meertens 1986) and the discussion in Sect. 10 below).

As the *map* example suggests, the formalism presented here can be used to provide formal definitions of, and prove properties of, higher order operators such as those of Bird (1987).

8 Termination and Timing

As noted previously, programs that are correct according to the calculus given so far in this paper may specify nonterminating computations. This is because any specification x may be used as a program provided it is refined by a program, with recursion allowed. For example, we might refine x by **if** b **then** x **else** x or even by just x.

It is possible (and often reasonable) to verify that a program terminates, or to verify a time bound for it, by analysing the program after it has been derived without explicit consideration of time. If the verification fails, it is back to the drawing board. Such analysis is discussed in, for example, (Sands 1989). In this section we explore an alternative idea, that of incorporating timing (and hence termination) requirements into the original specification and refining such specifications to obtain a program.

8.0 Specifications with Time

Rather than deal with termination and nontermination as a duality, we deal with the time required for a computation to complete. First we must expand the idea of an observation to include the time that is required for a computation. *Specifications with time* are written as $P@T$ where P is a specification of a value and T is a number specification. The kind of numbers used in the T part may include an infinity value. Nondeterministic expressions may be used to give a range of acceptable times. (Syntactically @ binds closer than any operator, even juxtaposition.)

Programming and other operators on specifications are lifted to specifications with time according to a *timing policy.* A timing policy reflects implementation decisions (such as whether operands are evaluated in sequence or parallel), language design decisions (such as strictness), and decisions about how much operations should cost. We will exhibit a particular timing policy based on sequential implementation, strict application, and charging at least one unit of time for each recursive call.

Primitives such as multiplication are lifted to specifications with time as

$$x@a \times y@b \quad \equiv \quad (x \times y)@(a+b)$$

The **if** is lifted as

$$\mathbf{if}\ x@a\ \mathbf{then}\ y@b\ \mathbf{else}\ z@c \quad \equiv \quad (\mathbf{if}\ x\ \mathbf{then}\ y\ \mathbf{else}\ z)@(a + \mathbf{if}\ x\ \mathbf{then}\ b\ \mathbf{else}\ c)$$

Specifications of functions with time specify both the time required to produce the functions and the time required to apply it (as a function of its argument). The specification $(\lambda i \cdot x@a)@b$ specifies a function that takes b time units to produce and $(\lambda i \cdot a)y$ time units to apply to y. The following way of lifting application models eager evaluation where the cost of evaluating the argument is assessed at the point of application. Let $(\lambda i \cdot x@a)^{\nu}$ mean $\lambda i \cdot x$ and $(\lambda i \cdot x@a)^{\tau}$ mean $\lambda i \cdot a$. Now

$$f@b\ y@c \quad \equiv \quad (f^{\nu}\ y)@(b + c + f^{\tau}\ y)$$

Reference to a refined specification is allowed as a programming construct (Sect. 4.1), but extra time may be optionally added. For example, if $x@0$ is a refined specification, one may make reference to $x@1$. In any loop of references, by this timing policy, at least 1 time unit must be added in the loop. Thus the observation that $x@a \sqsupseteq x@a$, although true, does not allow us to use $x@a$ in a program. On the other hand, if $x@a \sqsupseteq x@(a+1)$ is true, $x@a$ (or $x@(a+1)$) may be used as a program. For example, the observation that $x@\infty \sqsupseteq x@(\infty + 1)$ means that $x@\infty$ may be used as a program; but $x@\infty$ is not a very useful specification. Recursive reference should be a bit clearer with an example.

Since the suffix @0 occurs quite frequently we will take the liberty of not writing it, leaving it implicit.

8.1 An Example

Let ΣL be the sum of the elements of a list of naturals L. Our specification with time of a summation function is

$$sum \quad \overset{\text{def}}{\equiv} \quad \lambda L : nat^{*} \succ (\Sigma L)@(\#L)$$

The time required to produce the summation function must be 0, that is no recursive calls are allowed, by the convention of not writing @0. This is easily achieved if we write the function as a constant. The $@\#L$ means that the time required to apply the summation function to a list L is $\#L$. We will write sum'

for the same specification with the (implicit) @0 replaced by @1. However $\varSigma L$ is specified in detail, the following should hold

$$L = [\,] \succ\!\!- \varSigma L \sqsupseteq 0$$
$$L \neq [\,] \succ\!\!- \varSigma L \sqsupseteq L\,0 + \varSigma(L[1;..\#L])$$

The following are also true

$$L = [\,] \succ\!\!- \#L \sqsupseteq 0$$
$$L \neq [\,] \succ\!\!- \#L \sqsupseteq 1 + \#(L[1;..\#L])$$

With these theorems we can quickly derive the obvious program

$$\begin{array}{ll}
 & \mathit{sum} \qquad \textbf{if}\text{ introduction; case analysis; first and third theorems} \\
\sqsupseteq & \lambda L : \mathit{nat}^* \succ\!\!- \textbf{if } L = [\,] \textbf{ then } 0@0 \\
 & \qquad\qquad \textbf{else } (L \neq [\,] \succ\!\!- (\varSigma L)@(\#L)) \\
\multicolumn{2}{l}{\text{Second and fourth theorems}} \\
\sqsupseteq & \lambda L : \mathit{nat}^* \succ\!\!- \textbf{if } L = [\,] \textbf{ then } 0 \\
 & \qquad\qquad \textbf{else } (L\,0 + \varSigma(L[1;..\#L]))@(1 + \#(L[1;..\#L])) \\
\multicolumn{2}{l}{\text{Application for specifications with time}} \\
\sqsupseteq & \lambda L : \mathit{nat}^* \succ\!\!- \textbf{if } L = [\,] \textbf{ then } 0 \textbf{ else } L\,0 + \mathit{sum}'\,(L[1;..\#L])
\end{array}$$

8.2 Higher Order Specifications with Time

The time taken to apply a function obtained from an application of a higher-order function may well depend on the time to apply a closure. The τ and ν notation allows specification of such functions. An efficient map function is specified by

$$\begin{array}{l}
\lambda f\ \mathit{nat} : \mathit{nat} \succ\!\!- \lambda L : \mathit{nat}^* \succ\!\!- \\
(\S M : \mathit{nat}^* \wedge \#M = \#L \wedge (\forall i : 0,..\#L \Rightarrow M\ i = f^{\nu}\ (L\ i)))@(\sum_{i:0,..\#L} 1 + f^{\tau}\ (L\ i))
\end{array}$$

9 Pattern Matching

In modern function programming languages, functions are generally defined by a sequence of equations with the appropriate definition being picked according to pattern matching. Likewise the **case** construct of, for example, Haskell works by pattern matching. We look here at how this syntactic device can be given a semantics using the notation and theory presented earlier.

Since function definition by pattern matching can be understood in terms of the **case** construct we discuss only that. Consider the **case** expression

$$\begin{array}{l}
\textbf{case } x \textbf{ of } \{f\ i \rightarrow y; \\
\qquad\qquad\ g\ j \rightarrow z\ \}
\end{array}$$

Where f and g are functions mapping types T and U respectively to a third type. The case expression can be understood as the specification

$$\begin{array}{l}
x : f\,T, g\,U \succ\!\!- \textbf{let } k : x \twoheadrightarrow (\textbf{let } i \cdot k : f\ i \twoheadrightarrow y), \\
\qquad\qquad\qquad\qquad\qquad (\textbf{let } j \cdot k : g\ j \twoheadrightarrow z)
\end{array}$$

This interpretation of the case statement is nondeterministic when patterns overlap. Sequential pattern matching is modelled somewhat differently. The above case expression can be modelled as

$$\begin{aligned}&\mathbf{let}\ k : x \rightarrow \mathbf{if}\ k : f\ T\ \mathbf{then}\ (\mathbf{let}\ i \cdot k : f\ i \rightarrow y)\\ &\qquad\qquad \mathbf{else\ if}\ k : g\ U\ \mathbf{then}\ (\mathbf{let}\ j \cdot k : g\ j \rightarrow z)\\ &\qquad\qquad \mathbf{else}\ all\end{aligned}$$

10 Related Work

The use of logic to express the relationship between input and output dates back to work by Turing (Morris and Jones 1984), and is more recently found in the work of, for example, Hoare (1969) and Dijkstra (1975).

The uniform treatment of abstract specifications and programs is becoming common in imperative programming methodologies. Back (1987), Hehner (1984), Morgan (1988), and Morris (1990), building on the work of Dijkstra (1975), all extend imperative languages to include arbitrary specifications. A new methodology of Hehner (1990) treats the programming language as a subset of logic and uses logic as the full specification language.

Some of the specification constructs presented here are based on constructs that have been used in imperative specification. The $\rightarrow$, **try**, and **try else** operators, for example, are similar to operators described by Morgan (1988) and/or Nelson (1987).

In the functional programming community nondeterministic specifications have been avoided, perhaps because it is feared that nondeterminism does not mix with referential transparency. An exception is the work of Søndergaard and Sestoft (1988, 1990) which explores several varieties of nondeterminism and their relationships to referential transparency. Redelmeier (1984) used a form of weakest precondition semantics to define a programming language, but did not pursue nondeterminism or program derivation. Three bodies of work in functional program transformation do allow nondeterministic specifications. These are the CIP project (Bauer *et al.* 1987), Meertens's essay (Meertens 1986), and Hoogerwoord's thesis (Hoogerwoord 1989).

The CIP project involves not only functional programming, but also algebraic specification of data types and imperative programming. Only the functional programming aspects of CIP are discussed here. CIP is also a transformational approach based on nondeterministic specification. In CIP each specification is associated with a set of values called its breadth. One specification refines another if its breadth is a subset of the other's. CIP includes a **some** quantifier which closely parallels the § quantifier presented here. The significant differences between CIP and the formalism presented here are mainly in the treatment of errors, and predicates.

Errors in CIP are represented by a bottom value. The presence of the bottom value in the breadth of a specification means that the specification may lead to error. Many transformation rules have special side conditions about errors, especially in predicates. In the present formalism, errors are represented by *all* or by incompleteness with a resulting simplification.

Predicates in CIP are simply boolean specifications. This has a unifying effect, but, as with errors, adds side conditions to transformation rules, for example saying that the predicate must be deterministic and must not be in error. In the present formalism, we do not specify the exact language used for predicates, but we do assume that each predicate is either true or false in each state, although the logic may not be complete enough to say which. For example $0/0 = 5$ is not considered to be in error, nor to be nondeterministic. As in CIP, the side conditions about determinism are there, but are somewhat hidden. We are currently looking at allowing nondeterministic predicates without complicating the laws.

Recently Möller (1989) proposed an "assertion" construct for CIP. His construct, $P \ \triangleright x$ is similar to both our guard and assertion in that it is x when P is true, but differs from both our constructs in that it is the bottom (error) value when P is false. It is faithful to the notion of assertions as safety nets. By contrast, our assertion construct is used to represent context. The difference is illustrated by the assertion elimination law, which does not hold for Möller's assertions.

Meertens, in his excellent paper on transformational programming (Meertens 1986), discusses nondeterministic functional programs as a unified notation for specifications and programs. Unfortunately, Meertens confuses *null* (in his notation $[\![/0)$ with the undefined value (the error value). This leads him to choose between rejecting the validity of $x \sqsupseteq \mathit{null}$ and rejecting that $\sqsupseteq$ means "may (as a task) be replaced by." The solution is to accept both, regard *null* as the over-determined value, and use the undetermined value *all* to represent errors.

Meertens uses direct substitution for application. He also adopts the rule $(f, g)\ x \equiv f\ x, g\ x$. He correctly notes that these seemingly reasonable choices lead to contradictions. The following example is given

$$\begin{aligned} f &\overset{\text{def}}{\equiv} \lambda x \cdot x \\ g &\overset{\text{def}}{\equiv} \lambda x \cdot 3 \\ F &\overset{\text{def}}{\equiv} \lambda \phi \cdot \phi\, 1 + \phi\, 2 \end{aligned}$$

then

$$3, 6 \equiv (1{+}2), (3{+}3) \equiv F f, F g \equiv F(f, g) \equiv (f, g)1{+}(f, g)2 \equiv (1, 3){+}(2, 3) \equiv 3, 4, 5, 6$$

Our formalism avoids this paradox by carefully defining elementhood and allowing only elements as the values of parameters.

One outgrowth of Meertens's paper is the so-called Bird-Meertens formalism. Initially, nondeterministic specification was ignored (see e.g. (Bird 1987)). In (Bird 1990), Bird discusses nondeterministic specifications, but not the refinement order on them.

Hoogerwoord in his thesis (Hoogerwoord 1989) develops a calculational method of functional program development based on logical specifications. In contrast to the present paper, he does not treat specifications and expressions as objects of the same sort, and thus does not have a refinement calculus; rather, specifications are predicates that describe the desired expressions. Nondeterminism is not allowed in expressions themselves, but a specification may, of course, underdetermine the meaning of the desired expression.

11 Conclusions

We have presented a simple refinement calculus for functional programming and an attendant programming methodology. The key aspect of this calculus is that it treats specifications and executable expressions uniformly. This allows the programmer to formally express and to verify each step of a stepwise refinement. The calculus includes timing, not just for analysis after program development, but as a guide to development.

Several of the specification operators presented and used herein are new or new to functional programming, as far as we know. These include $\succ\!\!-$, $\rightarrowtail$, **let** , **try**, and **try else**.

The specification language is a small extension to a functional programming language. The extension allows the specifier to state the relationship between the free variables and the result of an expression. Because logic can be used to state this relationship, the language is expressive and natural to anyone familiar with logic. The specifier needs to state exactly the desired relationship and nothing more; there is no requirement that the relationship be functional. Furthermore, the relationship can be expressed in ways that are completely nonalgorithmic.

Acknowledgements We would like to thank Ray Blaak, Andrew Malton, and the referees for helpful comments on earlier drafts. We gratefully acknowledge the support of the Department of Computer Science at the University of Toronto and the Natural Sciences and Engineering Research Council.

References

R.J.R. Back. A calculus of refinement for program derivations. Technical Report 54, Department of Computer Science, Åbo Akademi, Finland, 1987.

F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Puakner, and P. Pepper. *The Munich Project CIP: Volume II: The Program Transformation System CIP-S.* Number 292 in Lecture Notes in Computer Science. Springer-Verlag, 1987.

R.S. Bird. Introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, number 36 in NATO ASI Series F. Springer, 1987.

R.S. Bird. A calculus of functions for program derivation. In David A. Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series. Addison-Wesley, 1990.

Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

Eric C.R. Hehner. *The Logic of Programming.* Prentice-Hall International, 1984.

Eric C.R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14:133–158, 1990.

C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

Rob Hoogerwoord. The design of functional programs: a calculational approach. PhD Thesis, Technische Universiteit Eindhoven, 1989.

Lambert Meertens. Algorithmics. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Mathematics and Computer Science*, number 1 in CWI Monographs. North-Holland, 1986.

Bernhard Möller. Applicative assertions. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

Carroll Morgan. The specification statement. *Trans. on Programming Languages and Systems*, 10(3):403–419, 1988.

F.L. Morris and C.B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, 1984.

Joseph M. Morris. Programs from specifications. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, pages 81–115. Addison-Wesley, 1990.

Greg Nelson. A generalization of Dijkstra's calculus. Technical Report 16, Digital Systems Research Center, Palo Alto, CA, U.S.A., April 1987. Also published in *Trans. on Programming Languages and Systems*, 11(4):517–561, 1989.

D. Hugh Redelmeier. *Towards Practical Functional Programming.* PhD thesis, University of Toronto, 1984.

David Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the 1989 Glasgow Functional Programming Workshop*, Workshops in Computing. Springer-Verlag, 1989.

Harald Søndergaard and Peter Sestoft. Nondeterminism in functional languages. Technical Report 88/18, Department of Computer Science, University of Melbourne, Australia, 1988.

Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27(6):505–518, 1990.

This article was processed using the LaTeX macro package with LLNCS style