Type Definitions of Polya

Dennis Volpano
David Gries*

TR 90-1085
February 1990

Department of Computer Science
Cornell University
Ithaca, NY 4853-7501

*The authors acknowledge joint support from the NSF and DARPA under grant ASC-88-00465.

Type Definitions in Polya

Dennis Volpano
David Gries†

Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853-7501

16 February 1990

Abstract

The programming language Polya maintains a clear separation between a type and its implementation through a new construct called the *transform*. Polya allows users to define their own data types and transforms to implement them. The type definition facility of Polya has capabilities not found in existing languages; in short, it allows a more comprehensive description of the properties that determine whether a program is well-formed. Two such properties are the *scope* of variables and the *bounded polymorphic* nature of some operations. One can specify the scope of any local variables that an operation introduces and express that the well-formedness of an operation depends on whether some overloaded function name stands for a function of a certain type. Also novel, is the ability to define literal classes for types and to specify both an abstract and concrete syntax for operations. With these capabilites, it becomes possible to define the syntax of a block-structured language within Polya itself.

D.3 [**Software**]: Programming Languages; D.3.3 [**Programming Languages**]: Language Constructs - *abstract data types*; D.3.4 [**Programming Languages**]: Processors - *compilers*; F.3 [**Theory of Computation**]: Logics and Meanings of Programs; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs - *type structure*

Additional Key Words and Phrases: type definition, syntax, overloading

† The authors acknowledge joint support from the NSF and DARPA under grant ASC-88-00465.

1. Introduction

The imperative programming language Polya maintains a clear separation between a data type and its implementation. Polya allows users to define their own data types and implement them through a new construct called the *transform*. A description of the transform can be found in [2]. This paper discusses type definitions in Polya. It is a chapter taken from the Polya language definition, suitably annotated with rationale for some of the design decisions that were made. It is therefore both expository and technical in nature.

The type definition facility of Polya has capabilities not found in existing languages. It permits a more comprehensive description of the properties that determine whether a program is well-formed, or syntactically correct. An operation that introduces a local variable can be defined with the scope of the variable, as well as its type, explicitly specified. It is also possible to specify that an operation is bounded polymorphic, in that its well-formedness depends on whether some overloaded function name denotes a function of a particular type. Also novel, is the ability to define literal classes for types and to specify both an abstract and concrete syntax for operations. With these capabilites, it becomes possible to define the syntax of a block-structured language within Polya itself.

The ability to define operations and introduce new concrete notations for them may seem alarming, since each programmer can extend the notation to fit their idiosyncrasies. Used in the proper fashion, however, the data type definition can be useful. For example, individuals in a particular area, say physics or computational chemistry, could write a set of basic definitions of types that are used frequently in their work. Their definitions could use the notation typically used in that area. Used in this way, the type definition facility provides a way to tune Polya to a specific domain of discourse.

Before describing type definitions, a brief description of the Polya type system is necessary. A program is judged to be well-typed with respect to an initial type environment, or set of type assumptions, and a set of typing rules. A type environment is a set of assumptions, each having the form $x:\sigma$, which ascribes type σ to identifier x. A_x is environment A with any assumptions about x discarded. An environment has at most one assumption for each identifier unless an identifier is overloaded, in which case it may contain any number of instance assumptions for the identifier, subject to certain restrictions. Typing rules are expressed in terms of typings, each of which has the form $A \vdash e:\sigma$, indicating that under environment A, it can be deduced that expression e has type σ.

There are three kinds of type in Polya: *data*, *constrained*, and *quantified*. These types are described below using a meta-level syntax that shall be used throughout this paper to simplify the specification of Polya syntax. Let [ ] describe a nonempty list, [ ]–s a nonempty list of elements separated by s, { } an option, and | an alternative. Then the data types are described by τ, the constrained types by ρ, and the quantified types by σ:

$$\tau ::= \alpha \ \mid\ \chi\,\{\,(\,[\,\tau\,]\text{–},\,)\,\}\ \mid\ \tau \to \tau' \ \mid\ \textbf{record}\ [\,id:\tau\,]\text{–;}\ \textbf{end}$$
$$\rho ::= (x:\tau)\,.\,\rho \ \mid\ \tau$$
$$\sigma ::= \forall\alpha.\sigma \ \mid\ \rho$$

where α is a type variable, χ a type constructor, like *int* or *bool*, that may have type arguments, and x an overloaded function name. The term $(x:\tau)$ is called a *constraint*. An expression of type $(x:\tau).\,\rho$ has type ρ only if x has type τ. An assumption $x:\tau$ may be shifted from a type environment into the type of an expression as a constraint as long as x is overloaded in the environment.

The initial type environment and typing rules for assigning types to expressions come from data type definitions. A data type definition specifies only the syntax of operations on the type, not their semantics. It has the form

type *id* { ([*type-var*]–,) }
{ **literals** [*lit-class-defn*]–; }
operations [*operation-defn*]–;
end

The name of the type, *id*, is followed by a list of type variables, one for each parameter of the type, if the type is parametric. Literals, or manifest constants, of the type are described by literal class definitions given after **literals**. Literal classes are optional because the manifest constants of some types can be defined instead as functions with no arguments. Operation definitions are given following **operations**. Operations may be functions, expressions, or statements. Every data type definition has global scope.

Type definitions have only one kind of argument, namely type arguments. This simplifies the language definition because no additional rules are needed to ensure that identical type formulas always denote equivalent types, as in Russell [1]. More experience with Polya will determine whether other kinds of arguments should be permitted.

1.1. Literal classes

The definition of a literal class has the form

id { [*lit-comp-decl*] } **as** *lit-concrete-syn*

The name of the literal class, *id*, may be followed by a list of component declarations that describe the components of literals in the class. This part of the definition is regarded as the abstract syntax of the literal class, which is the syntax that would appear in a formal specification, given outside Polya, of the semantics of literals in the class. The concrete syntax for the class is given after **as**. That is, literals in the class are expressed in programs **as** strings in the language defined by the concrete syntax.

Every definition of a literal class introduces another *lexical class* and production into Polya and another type assumption into the initial type environment. The lexical class is defined by a *string description* formed from the concrete syntax for the literal class by replacing each component name by the string description it designates in the abstract syntax. If L is the name of a literal class defined within the definition of type T, then a lexical class called L is introduced, the context-free grammar of Polya is augmented with production $exp = L$, and assumption $L : T$ is added to the initial type environment.

1.1.1. Component declarations

The simplest form of component declaration in a literal class is *s*–*id*, which declares a component named *id*. The component is defined by string description *s*. For example, below are abstract-syntax parts for definitions of literal classes *bit*, *binary*, *octal*, and *hex*. Each class has one component describing a string of digits to be interpreted in a particular base.

bit (0 1)–*bit*
binary [(0 1)]–*bits*
octal [(0..7)]–*octaldigits*
hex [(0..9 *A*..*F*)]–*hexdigits*

The first example illustrates the simplest form of string description: a sequence of characters enclosed in parentheses called a *character class*. Component *bit* may be either character 0 or 1.

A character class contains printable characters in the ASCII range " "..~. Two characters are special:) and \. In a character class, they must be preceded by the escape character \; e.g. (\\ \) .) is a class containing a backslash, a right parenthesis, and a period.

The brackets [and] in the second example indicate *repetition* : component *bits* is a non-empty string of characters from the class (0 1) .

The third example illustrates a *range* . Component *octaldigits* is a non-empty string of digits in the range 0, 1, 2, . . . , 7 ; *hexdigits* is a non-empty string of digits or letters from the specified ranges.

A range has the form *lo .. hi* where *lo* and *hi* are characters whose ASCII character codes are such that $lo \le hi$. It denotes all characters in the ASCII character set whose character codes lie in the range *lo .. hi* inclusive.

A collection of components may be optional, which is conveyed by enclosing declarations for them in braces { } . Options may be nested, as they are, for instance, in the definition of literal class *float* below. A literal in *float* is to be interpreted as the value $whole.fraction \times 10^{sign\ expon}$.

```
float [(0..9)]-fraction
      [(0..9)]-whole
      {{(+ -)-sign} [(0..9)]-expon}
```

Two of the four components of *float* are optional, namely *sign* and *expon* . Notice that component *whole* is the second component of the abstract syntax even though it appears first in the interpretation of *float* . Because components are identified by name, rather than position, their order in an abstract syntax is irrelevant.

1.1.2. Concrete syntax

A concrete syntax definition for a literal class specifies a notation to be used in programs for expressing literals in the class. It is formulated using options, strings (syntactic sugar), and component names. Names that appear in it must be precisely the component names of the abstract syntax. A string is formed from ASCII characters in the range ! .. ˜ and is delimited by a pair of double quotes. Below are examples of literal classes from type *int* , with the concrete syntax for each class given after **as** .

```
zero                            as "zero" ;
octal [(0..7)]-octaldigits      as "8_" octaldigits ;
hex [(0..9 A..F)]-hexdigits     as "16_" hexdigits ;
```

White-space characters, such as blanks, tabs, and newlines, delimit user-defined literals in programs. Consequently, no white space may appear between the components of a literal in its concrete syntax. For example, "8_" *octaldigits* specifies any string of characters that begins with 8_ , ends with digits in the range 0 .. 7 , and contains no white space.

There are two special characters: " and \ . In a concrete syntax, these characters must be preceded by the escape character \ , as is illustrated below.

```
type char
literals
    printable_chr (" "..˜)-chr as "'" chr "'" ;
    nonprintable_chr (0 1)-octal2 (0..7)-octal1 (0..7)-octal0
        as "'\\" octal2 octal1 octal0 "'" ;
    newline         as "'\\n'" ;
    backslash       as "'\\\\'" ;
    double_quote    as "'\\"'" ;
...
end
```

The longest possible literal is always recognized, so care must be exercised when using blanks in character classes. Allowing blanks to occur in identifiers, for example, as provided by

id (*a* .. *z* *A* .. Z)–*lead* [(*a* .. *z* *A* .. *Z* " ")]–*trail* **as** *lead trail*

would not be wise, for it leads to unexpected consequences like a function application (*f x*) being treated instead as a parenthesized identifier. If blanks no longer always delimit literals, as is the case for *id* , then it is recommended that the concrete syntax provide delimiters, as is done below with a *text* literal, which must be enclosed by a pair of double quotes.

text [(" " .. ~)]–*s* **as** " \" *s* \" "

Options may appear in a concrete syntax definition, but only if the following rule is observed. Every combination of component names possible in the concrete syntax must also be possible in the abstract syntax. For example, below is a definition of literal class *float* .

float [(0 .. 9)]–*fraction* [(0 .. 9)]–*whole* { {(+ –)–*sign*} [(0 .. 9)]–*expon*}
as *whole* "." *fraction* {"E" {*sign*} *expon*} ;

Every combination of component names possible in the concrete syntax is also possible in the abstract syntax. A concrete syntax definition for which this is not true is given below.

{*whole*} "." *fraction* {"E" {*sign*} *expon*}

Now the combination of *fraction* , *sign* , and *expon* is possible in the concrete syntax but not in the abstract syntax, which requires *whole* to be present. This concrete syntax allows literals like .5 , for example, but the abstract syntax of *float* does not, requiring 0.5 instead. Therefore, it is not a concrete syntax for *float* .

From the concrete syntax definition for *float* , a lexical class of the same name is introduced, with the following string description as its definition.

[(0..9)] "." [(0..9)] {"E" {(+ –)} [(0..9)]}

Strings recognized by parts of this string description are associated with components of the abstract syntax for *float* . Any string recognized by [(0 .. 9)] to the left of ".", for instance, is identified with component *whole* , and not with *fraction* or *expon* . Production *exp* = *float* is added to the Polya grammar, and assumption *float*: *real* to the type environment if *float* is defined within type *real* .

Literal classes with distinct names must introduce lexical classes defined by *nonoverlapping* string descriptions; two such descriptions overlap if the sets of strings they describe intersect. This restriction ensures that every literal belongs to a unique lexical class. For example, below is an illegal combination of type definitions.

type *real*	**type** *int*
literals	**literals**
realdecimal [(0..9)]–*r* **as** *r* ;	*intdecimal* [(0..9)]–*i* **as** *i* ;
. . .	. . .
end	**end**

These two type definitions define different literal classes, but the string descriptions formed from their concrete syntax definitions overlap (in fact, are identical).

1.1.3. Overloading

A literal class name may be overloaded, in that it may be defined by literal class definitions within different types. However, it may not have more than one concrete syntax. That is, the string descriptions formed from the various concrete syntax definitions for an overloaded literal class must define the same set of strings. For example, below is a legal combination of type definitions that overload literal class *decimal*.

type *real*	**type** *int*
literals	**literals**
decimal [(0..9)]–*r* **as** *r*	*decimal* [(0..9)]–*i* **as** *i*
...	...
end	**end**

The concrete syntax for *decimal*, as prescribed by *real*, is defined by [(0..9)], as is the syntax prescribed for it by *int*. These type definitions contribute type assumptions *decimal*: *real* and *decimal*: *int* to the initial type environment.

1.2. Operation definitions

The definition of an operation has the form

id { [*opn-comp-decl*] } ({ [*comp-prop*]–; }) { ([*constr*]–;) } { *type* }
    { **as** *opn-concrete-syn* }

The identifier is the name of the operation. It is followed by a list of component declarations that declare the components of the operation. This much of the definition is regarded as the operation's abstract syntax. The operation's concrete syntax is specified after **as**. Type and scope information about the components is given in a list of component properties. Following this list is a list of constraints, each of which specifies a type that some overloaded function name must have in order for an instance of the operation to be well-formed.

There are three different kinds of operations —functions, expressions, and statements— depending on which options of the above form are exercised:

function:	*id* ({ [*comp-prop*]–; }) { ([*constr*]–;) } *type*
expression:	*id* [*opn-comp-decl*] ({ [*comp-prop*]–; }) { ([*constr*]–;) } *type*
statement:	*id* [*opn-comp-decl*] ({ [*comp-prop*]–; }) { ([*constr*]–;) }

The component properties in the case of a function declare the types of the function's parameters, while *type* is the type of its result. In the case of an expression, properties define not only the types of components, but also the scopes of any local variables the expression might declare; *type* is the type of the expression. Statement definitions are similar except that *type* is omitted.

Every operation definition introduces another *sentential form* and production into Polya. In addition, function and expression definitions also introduce, respectively, type assumptions into the initial type environment and typing rules into the set of Polya typing rules. The sentential form is derived from the operation's concrete syntax definition by replacing each component name with the nonterminal designated by its kind. If S is the sentential form so obtained from the concrete syntax definition given for a function or an expression, then production $exp = S$ is added to the Polya grammar; $stmt = S$ is added if the syntax definition is given for a statement.

1.2.1. Component declarations

The simplest form of component declaration for an operation is *k–id*, which declares a component named *id* of kind *k*. There are five kinds, each denoting the terms derivable from the Polya nonterminal of the same name:

id , **var** , **exp** , **stmt** , **type** .

Below are examples of abstract-syntax parts for definitions of *plus* , *let* , and *block* .

plus **exp**–*x* **exp**–*y*
let **id**–*x* **exp**–*e* **exp**–*b*
block **var**–*v* **type**–*t* **stmt**–*s*

A collection of components is specified as optional by enclosing their declarations in braces { } . Options may be nested. For example, the first block below permits a single variable declaration, which may be omitted; the second block permits the same except that it also allows a variable to be declared without its type.

block { **var**–*v* **type**–*t* } **stmt**–*s*
block { **var**–*v* { **type**–*t* } } **stmt**–*s*

An operation can often be generalized by converting one of its components into a list of components. This is conveyed in its definition by enclosing the declaration for the component within brackets [ ] . For example, replacing **var**–*v* in the preceding definitions of a block statement by [**var**–*v*] means that *v* now denotes a list of one or more variables.

Some operations require two (or more) lists of the same length, whose elements are in a one-to-one correspondence. For example, a block statement may declare a list *v* of variables and a corresponding list *t* of types, with each variable v_i of list *v* having type t_i . However, *v* and *t* denote unrelated lists with the two separate component declarations [**var**–*v*] [**type**–*t*] . To denote corresponding lists of the same length, *v* and *t* must be enclosed within the same bracket pair as in

block [**var**–*v* **type**–*t*] **stmt**–*s* .

A list of lists is described when list brackets are nested in an abstract syntax. For example, with the preceding abstract syntax for a block, a type must be duplicated as many times as there are variables. The block syntax below allows variables with the same type to be listed together, with the type appearing only once.

block [[**var**–*v*] **type**–*t*] **stmt**–*s*

Here *v* denotes a list, each element of which is a list of variables; *v* is still said to denote a **var** list even though it stands for a list of **var** lists.

Additional flexibility is gained by mixing options and list descriptions in an abstract syntax. For example, the following abstract syntax is for a block with an optional list of variable declarations:

block { [**var**–*v* **type**–*t*] } **stmt**–*s*

and below is a block syntax that allows some variables to be declared without types.

block [**var**–v { **type**–t }] **stmt**–s

Recall that because v and t are enclosed by the same pair of list brackets, there is a correspondence between their elements. For some elements of v, there are corresponding types in t, and for others, there are none. For example, v might denote the list $[x, y]$ and t the list $[\_, real]$, where underscore conveys that the type of x has been omitted.

A general rule is needed to interpret any component declaration that describes a list whose elements may be optional. For some $k > 0$, let $[D]^k$ stand for a list of k elements, each described by D, and $[D \mid \_]^k$ for a list of k elements, each of which is a component described by D or an occurrence of placeholder "_". A component declaration of the form $[D_0 \cdots D_{n-1}]$ is interpreted as follows.

$$[D_0 \cdots D_{n-1}] \;=\; C_0^k \cdots C_{n-1}^k \qquad \text{where } C_i = \begin{cases} [D \mid \_] & \text{if } D_i = \{D\} \\ [D] & \text{if } D_i = D \end{cases}$$

For example, the following abstract syntax might be used to describe a procedure block: one that allows multiple procedures to be declared and referenced within a statement body s.

block [**id**–i { [**id**–p { **type**–t }] } **stmt**–b] **stmt**–s

This abstract syntax is interpreted as

block $[$**id**–$i]^k$ $[[$**id**–$p]^j$ $[$**type**–$t \mid \_]^j \mid \_]^k$ $[$**stmt**–$b]^k$ **stmt**–s

for some $j, k > 0$. Here i denotes a list of procedure names, p a list of parameter lists, t a list of type lists, and b a list of procedure bodies. Every procedure name has a corresponding statement body and may have a corresponding parameter list. Each parameter may have a corresponding type. Below is a legal assignment to i, p, t, and b.

$i = [f, g]$
$p = [\_, [x, y]]$
$t = [\_, [int, \_]]$
$b = [body_f, body_g]$

1.2.2. Component properties

Two kinds of properties can be expressed in the definition of an operation: *type* and *scope*. These properties, in the definition of a function or an expression, give rise either to an assumption that augments the initial type environment, or a typing rule that augments the Polya typing rules. A function definition contributes a type assumption, an expression definition a typing rule.

Type

A component of kind **id**, **var** or **exp** in an expression or statement definition must appear in a *type declaration* within that definition. A type declaration of the form $id : \tau$, where the identifier is the name of a component whose kind is **id**, **var**, or **exp**, assigns type τ to the component. To the left below are examples of abstract syntax with type declarations, and to the right, the typing rules or type assumptions they contribute.

plus $(x : int;\ y : int)\ int$ $\qquad\qquad$ $plus : int \rightarrow int \rightarrow int$

identity **exp**–e **type**–t $(e:t)\,t$

$$\frac{A \vdash e : \tau}{A \vdash (identity\ e\ \tau) : \tau}$$

identity **exp**–e $(e:*a)\,*a$

$$\frac{A \vdash e : \tau}{A \vdash (identity\ e\,) : \tau}$$

assign **var**–v **exp**–e $(v:*a;\ \ e:*a)$

All components of functions must be expressions, so that components x and y of *plus* have kind **exp** . In the first version of *identity* , the type of the expression denoted by e is a component. Therefore, a type must be explicit in every instance of this version in a program. The second version drops this restriction, forcing the type to be inferred.

Polya has expansion rules for expanding a type declaration that involves a component list into a list of type declarations. This allows for succinct expression of type assignment. Suppose e is the name of an **id** , **var** , or **exp** list with length m . Then the two expansion rules for type declarations, which may be applied recursively, are

$$e:T \;=\; \begin{cases} e_0:T,\ldots,e_{m-1}:T & \text{if } T \text{ is a type} \\ e_0:T_0,\ldots,e_{m-1}:T_{m-1} & \text{if } T \text{ is a } \textbf{type} \text{ list of length } m \end{cases}$$

Both rules are used in the type declaration of

block {[[**var**–v] **type**–t]} **stmt**–s $(v:t;\ (v)s)$.

In this example, v denotes a list of **var** lists of length k say, so by the second rule, $v:t$ expands to $v_0:t_0,\ldots,v_{k-1}:t_{k-1}$. But v_i is also a list of length m (say); call it w . So $w:t_i$ expands to $w_0:t_i,\ldots,w_{m-1}:t_i$ by the first rule. The expansion rules are summarized in the table below.

	list v of length m	non-list v
list T of length m	$v:T = v_0:T_0,\ldots,v_{m-1}:T_{m-1}$	$v:T$ is undefined
non-list T	$v:T = v_0:T,\ldots,v_{m-1}:T$	$v:T = v:T$

A list of distinct type variables is created by enclosing a single type variable within brackets; e.g. $[*a]$ produces $*a_0,\ *a_1,\ \cdots$. A list of type variables may be used to ascribe a type to a list of components of the appropriate kind, as in $v:[*a]$, where v names a list; the second expansion rule then applies. The length of $[*a]$ is inherited from the length of v , and every occurrence of $[*a]$ in a definition leads to the same list of unique type variables.

For example, suppose we wish to define a multiple assignment statement in which the types of the target variables may be different. The following is incorrect because, under the first expansion rule, it prescribes the same type for each target variable.

assign [**var**–v **exp**–e] $(v:*a;\ \ e:*a)$

Correct type declarations are given in

assign [**var**–v **exp**–e] $(v:[*a];\ e:[*a])$.

Here, the type of each variable v_i and the corresponding expression e_i have to be the same $(*a_i)$, but the types of the v_i may be different.

Scope

A component of kind **id** or **var** in an expression or statement definition must appear within a *scope declaration* in that definition if the component signifies the introduction of a new identifier or variable. In a scope declaration of the form $(v)B$, v is the name of an **id** or **var** component and B is the name of an **exp** or **stmt** component. For a given instance of the expression or statement being defined, $(v)B$ indicates that all free occurrences of the variable or identifier named by v in the statement or expression named by B are bound in the instance. Within a single definition, v may appear in more than one scope declaration. The set of all B such that $(v)B$ is a scope declaration determines the scope of the variable or identifier named by v .

For example, consider an instance of the block statement whose abstract syntax is

block **var**–v **type**–t **stmt**–s $(v:t;\ (v)s)$.

If x is the variable denoted by v for this instance, then all free occurrences of x in the statement denoted by s are bound in the instance; each is an occurrence of the x denoted by v . The scope of x is the statement denoted by s . It is important to realize that any identifier denoted by v is *declared* to be a variable (i.e. has kind **var**) as a consequence of the component declaration **var**–v and the scope declaration $(v)s$. In contrast, any identifier denoted by v in the abstract syntax

assign **var**–v **exp**–e $(v:*a;\ e:*a)$

is *required* to be a variable because v is declared as **var**–v and has no scope declaration.

As another example, consider

let **id**–x **exp**–e **exp**–b $(x:*a;\ e:*a;\ (x)b;\ b:*t)*t$.

Note that the scope of the identifier named by x does not include e , which is what one would expect unless the identifier is recursively defined.

For any two scope declarations of the form $(u)B$ and $(v)B$ within the component properties of a definition, u and v are required to denote different identifiers or variables. If u and v could denote the same identifier x , say, then a free occurrence of x in B would be ambiguous, for it could refer to the x denoted by u or the x denoted by v . This restriction would have to be lifted if Polya were to permit the definition of operations that locally overload function names.

Scope declarations in expression definitions give rise to typing rules in which new assumptions about identifiers or variables are added to the type environment in order to arrive at types for the terms in which they appear. For every scope declaration of the form $(v)B$, where B is an **exp** component, in the definition of an expression, there is a sequent $A_v \cup \{v:\tau\} \vdash B:\tau'$ in the antecedent of the expression's typing rule. To the left below are examples of abstract syntax with type and scope declarations; to the right, the typing rules they contribute.

let **id**–x **exp**–e **exp**–b $(x:*a;\ e:*a;\ (x)b;\ b:*t)*t$	$$\frac{A \vdash e:\tau \quad A_x \cup \{x:\tau\} \vdash b:\tau'}{A \vdash (let\ x\ e\ b):\tau'}$$
forall **id**–i **type**–t **exp**–r **exp**–s $(i:t;\ r:bool;\ s:bool;\ (i)r;\ (i)s)bool$	$$\frac{\begin{array}{c}A_i \cup \{i:\tau\} \vdash r:bool \\ A_i \cup \{i:\tau\} \vdash s:bool\end{array}}{A \vdash (forall\ i\ \tau\ r\ s):bool}$$

Note that in each of the rules, an assumption about an identifier is added to an environment identical to A except that any prior assumptions about the identifier are discarded. It is assumed that the semantics of any operation that introduces new variables or identifiers, like *let* and *forall* , requires that they be defined prior to being referenced anywhere in their scope. In the case of *let* , instances of x are bound to instances of e ; in *forall* , instances of i are bound to values in ranges denoted by r . Therefore, assumptions added to the type environment as a result of scope declarations can be discharged in the consequent of a typing rule.

Notice that the polymorphic *let* of Standard ML [3] cannot be defined in Polya as an operation. The reason is that there is no way to define an operation that introduces a local identifier with quantified type, the kind of type that must be assigned to the polymorphic, let-bound identifier. For example, with *let* defined above, the expression

$$let\ f\ \mathbf{fn}(x)x\ \ (\ldots f(0)\ldots f(true)\ldots)$$

is not well-typed because f is not polymorphic, but would be well-typed if *let* were an instance of the ML *let* . Polymorphism in Polya is introduced at the level of constant declarations, so the above expression could be written as

$$\mathbf{const}\ f = \mathbf{fn}(x)x;\ \ \mathbf{yield}\ (\ldots f(0)\ldots f(true)\ldots)$$

with a Polya block expression, but the block expression cannot be defined in Polya. Consequently, Polya cannot be defined in itself.

Like type declarations, there are two expansion rules for scope declarations. Suppose v is the name of an **id** or **var** list with length m . Then the two rules are

$$(v)B = \begin{cases} (v_0)B, \ldots, (v_{m-1})B & \text{if } B \text{ is an } \mathbf{exp} \text{ or } \mathbf{stmt} \text{ component} \\ (v_0)B_0, \ldots, (v_{m-1})B_{m-1} & \text{if } B \text{ is an } \mathbf{exp} \text{ or } \mathbf{stmt} \text{ list of length } m \end{cases}$$

Each of these expansion rules may be applied recursively. For example, the scope declaration below is expanded by two applications of the first rule.

$$block\ \ \{[[\mathbf{var}\text{–}v\,]\ \mathbf{type}\text{–}t\,]\,\}\ \ \mathbf{stmt}\text{–}s\ \ ((v:t)s)$$

The expansion rules are summarized in the table below.

	list v of length m	non-list v
list B of length m	$(v)B = (v_0)B_0, \ldots, (v_{m-1})B_{m-1}$	$(v)B$ is undefined
non-list B	$(v)B = (v_0)B, \ldots, (v_{m-1})B$	$(v)B = (v)B$

The expansion rules always lead to a pointwise declaration of scope for variables or identifiers in a list, which may be unacceptable. Suppose, for example, one needs to define an expression that allows for the mutually-recursive definition of some number of identifiers $v_0, \ldots, v_{n-1}$, all of which can be referenced in a body b :

$$\mathbf{letrec}\ v_0:t_0, \ldots, v_{n-1}:t_{n-1}\ \ \mathbf{where}\ v_0 = e_0, \ldots, v_{n-1} = e_{n-1}\ \ \mathbf{in}\ b\,.$$

The definition must prescribe the scope of each v_i as $e_0, \ldots, e_{n-1}$ as well as b . If V is the name of an **id** list corresponding to $v_0, \ldots, v_{n-1}$ and E the name of an **exp** list corresponding to $e_0, \ldots, e_{n-1}$, then the closest we can come to specifying the desired scope declaration for the v_i is $(V)E$ and $(V)b$. However, $(V)E$ prescribes the scope of v_i as e_i only. The correct scope declaration for the v_i requires the list distribution operator / .

In addition to ordinary scope declarations over component lists, the operator / allows distribution of a list over the components of another list to produce a list of scope declarations. Let v be the name of an **id** or **var** component or the name of a list of such components. Then the list distribution operator is defined by the following two rules.

$$(v)\,/\,B \;=\; \begin{cases} (v)B & \text{if } B \text{ is an } \mathbf{exp} \text{ or } \mathbf{stmt} \text{ component} \\ (v)\,/\,B_0\,,\ldots,\,(v)\,/\,B_{m-1} & \text{if } B \text{ is an } \mathbf{exp} \text{ or } \mathbf{stmt} \text{ list of length } m \end{cases}$$

An abstract syntax for *letrec*, with type and scope declarations, can now be given:

$$\mathit{letrec}\ [\,\mathbf{id}\text{–}v\ \ \mathbf{type}\text{–}t\ \ \mathbf{exp}\text{–}e\,]\ \ \mathbf{exp}\text{–}b\ \ (e:t;\ \ v:t;\ \ (v)b;\ \ (v)\,/\,e;\ \ b:{*a})\ {*a}\,.$$

Here, the scope of each v_i is $e_1\,,\ldots,\,e_n$ as well as b, as desired. The distribution rules are summarized in the following table.

	list or non-list v
list B of length m	$(v)\,/\,B \;=\; (v)\,/\,B_0\,,\ldots,\,(v)\,/\,B_{m-1}$
non-list B	$(v)\,/\,B \;=\; (v)B$

Polya permits scope declarations that have parts in common to be abbreviated. Moreover, type declarations may appear in scope declarations. For example, the properties of *let* may be abbreviated

$$\mathit{let}\ \ \mathbf{id}\text{–}x\ \ \mathbf{exp}\text{–}e\ \ \mathbf{exp}\text{–}b\ \ (e:{*a};\ \ (x:{*a})b:{*t})\ {*t}\,.$$

1.2.3. Constraints

Whether an instance of an operation is well-formed may depend on whether an overloaded function name, say f, stands for a function of a particular type, say τ. Such a dependency is expressed in the definition of an operation by including a constraint of the form $f\!:\!\tau$ in a constraint list following the list of component properties. See Sect. 1.2.5, on overloading function names, for examples.

1.2.4. Concrete syntax

The notation used to express an operation in a program is the operation's concrete syntax. Specified as part of the operation's definition, it is formulated from options, lists, strings, and component names. Names that appear in it must be precisely the component names of the abstract syntax. A string is formed from ASCII characters in the range !..~ and is delimited by double quotes, making " a special character along with \. In a string, these characters must be preceded by the escape character \. Below are examples of operation definitions.

mod $(x : \mathit{int};\ \ y : \mathit{int})\ \mathit{int}$ **as** x "mod" y

for **var**–v **exp**–b **stmt**–s $((v : {*a})\,s;\ \ b : \mathit{set}\,({*a}))$
  as "for" "(" v "in" b ";" s ")"

let **id**–x **exp**–e **exp**–b $(e : {*a};\ \ (x : {*a})\,b : {*t})\ {*t}$
  as "let" x "=" e "in" b

Adjacent components in the concrete syntax for an operation may be separated by white space. Thus, the concrete syntax for statement *for* above allows "for" and "(" to be separated by a blank.

But had the concrete syntax been given as "for(" instead, then no white space could separate them.

The concrete syntax specification for a function definition may be omitted, in which case the syntax for an application of it has one of three standard forms. In the definitions below, for example, no concrete syntax is specified, so applications of functions *zero* and *mod* will have the form *zero* and $mod(x, y)$, or $mod.x.y$.

zero ( ) *int*

mod (x : *int*; y : *int*) *int*

Options may appear in a concrete syntax definition only if every combination of component names possible in the concrete syntax is also possible in the abstract syntax. For example, the concrete syntax definition in the block below violates this rule.

block { **var**–v **type**–t } **stmt**–s ((v : t) s)
    as { "var" v { ":" t } } "begin" s "end"

The combination of v (a variable) together with only s (a statement) is possible in the concrete syntax but not in the abstract syntax, which requires that v always be accompanied by a type. Below is a concrete syntax definition that satisfies the rule.

{ "var" v ":" t } "begin" s "end"

From this definition, a sentential form *block*, defined by

{ "var" *var* ":" *type* } "begin" *stmt* "end"

is introduced. Production *stmt* = *block* is added to the Polya grammar.

The Polya grammar must be unambiguous, in that every terminal string has at most one derivation. Consider, for example, the following operation definition.

plus (x : *int*; y : *int*) *int* **as** x "+" y

No information about the associativity of + is given, so the Polya grammar becomes ambiguous with the production contributed by this definition; $x + y + z$ can be parsed as $(x + y) + z$ or $x + (y + z)$.

Lists

Lists specified in a concrete syntax definition have one of two forms: $[S]$ and $[S]\!-\!s$ where S may comprise any element of a concrete syntax definition. The form $[S]$ describes a nonempty list, each element of which is described by S, with no separators, whereas $[S]\!-\!s$ describes the same kind of list except that list elements are separated by s, which may be either a character or a string. If it is a string, it is delimited by double quotes. Every component name in S must denote a list in the corresponding abstract syntax.

Suppose S has the form $w_0\, n_0 \cdots w_{k-1}\, n_{k-1}\, w_k$, where $n_0, \ldots, n_{k-1}$ are all the component names found in S, and $w_0, \ldots, w_k$ represent everything else. Interpretations of $[S]$ and $[S]\!-\!s$ are given below.

$[S]$ $\quad (w_0\, n_{0_0} \cdots w_{k-1}\, n_{k-1_0}\, w_k) \cdots (w_0\, n_{0_m} \cdots w_{k-1}\, n_{k-1_m}\, w_k)$

$[S]\!-\!s$ $\quad (w_0\, n_{0_0} \cdots w_{k-1}\, n_{k-1_0}\, w_k)\ s \cdots s\ (w_0\, n_{0_m} \cdots w_{k-1}\, n_{k-1_m}\, w_k)$

By the above interpretations, it is an error to have a component name in S if it is not a list in the abstract syntax. Below are examples of definitions with lists.

> *letrec* [**id**–v **type**–t **exp**–e] **exp**–b $(e:t;\ (v:t)b:*a;\ (v)/e)*a$
> **as** "letrec" [v ":" t]–, "where" [v "=" e]–, "in" b
>
> *block* { [[**var**–w] **type**–t] } **stmt**–s $((w:t)s)$
> **as** { "var" [[w]–, ":" t]–; } "begin" s "end"

Below are the interpretations of the lists occurring in the preceding definitions.

[v ":" t]–,	$v_0 : t_0\ ,\ \cdots,\ v_m : t_m$
[v "=" e]–,	$v_0 = e_0\ ,\ \cdots,\ v_m = e_m$
[[w]–, ":" t]–;	$[w_0]\text{–},:t_0\ ;\ \cdots;\ [w_n]\text{–},:t_n$

In the definition of *letrec* , the list interpretation requires v , t , and e to be lists of equal length, which they are, as specified in the abstract syntax. The list interpretation requires w to be a list of lists in the definition of *block* , which it is by virtue of being declared a list of **var** lists in the abstract syntax.

The abstract syntax of a definition may require the lengths of some lists to be equal when the concrete syntax does not. For example, in the following, where v and t must denote lists of equal length,

> *block* { [**var**–v **type**–t] } **stmt**–s $((v:t)s)$
> **as** { "var" [v ":" t]–; } "begin" s "end"

the concrete syntax requires the lists denoted by v and t to have the same length, but the concrete syntax in the following definition does not.

> *block* { [**var**–v **type**–t] } **stmt**–s $((v:t)s)$
> **as** { "var" [v]–, ":" [t]–, } "begin" s "end"

Although the concrete syntax definition does not enforce the equality, it is still observed in that lists described by [v]–, and [t]–, must have the same length.

The complement of the above case is also allowed, in that an abstract syntax may not require the lengths of some lists to be equal when the concrete syntax does. For example, in the following, where v and t can denote lists of different lengths,

> *block* { [**var**–v] [**type**–t] } **stmt**–s $((v:t)s)$
> **as** { "var" [v ":" t]–; } "begin" s "end"

the concrete syntax requires the lists denoted by v and t to have the same length while the abstract syntax does not. This simply means that the syntax of programs that use this block will be more restricted than necessary.

Operators

Consider again, the definition

> *plus* $(x:int;\ y:int)\ int$ **as** x "+" y

As indicated earlier, this definition leads to an ambiguous Polya grammar. Although the definitions below do not cause ambiguity, they do not allow *plus* to be expressed as an infix operator.

plus $(x : int;\ y : int)$ *int*

plus $(x : int;\ y : int)$ *int* as "+" "(" x "," y ")"

There is an alternative to the above definitions that allows us to regain the flexibility of the first definition (unparenthesized, infix use of +) without ambiguity. It involves defining *plus* as an *operator*.

An operator is a unary or binary function whose concrete syntax is given as a symbol together with a declaration of the symbol's position relative to its operands. There are three positions: **infix**, which applies to binary operators, and **prefix** and **postfix**, which apply to unary operators only. Below are examples of functions defined as operators.

plus $(x : int;\ y : int)$ *int*	**infix** "+"
mod $(x : int;\ y : int)$ *int*	**infix** "mod"
factorial $(i : int)$ *int*	**postfix** "!"
not $(b : bool)$ *bool*	**prefix** "~"

The precedence and associativity of operators is given in a *precedence declaration* which is given separately from type definitions. There is one such declaration for all operators; it is a list of the form

[*assoc* [*op-name*]]

where *assoc*, which is either **left** or **right**, is the associativity of all operators in the list of operator names that follow it. Operators in the first list of a precedence declaration have highest precedence, those in the second, next highest, and so on. For example, below is a precedence declaration.

left *factorial*
left *not*
left *mod and*
left *plus minus*

Here *factorial* has highest precedence followed by *not*. With this declaration, $x + y + z$ is parsed as $(x + y) + z$ because + associates to the left, and $x\ mod\, y + 1$ is parsed as $(x\ mod\, y) + 1$ because *mod* has higher precedence than + . A precedence declaration is always given in terms of operator names from the abstract syntax, not symbols of the concrete syntax.

Binary predicates defined as operators, such as *less* (<), are often used in mathematical notation with implicit conjunction as in $a < b < c$, which means $a < b \wedge b < c$. Polya allows a binary predicate to be declared as a conjunctive operator, so that when it occurs opposite another conjunctive operator in an expression, a conjunction of the two predicates is assumed. This is done by specifying **conj** in place of associativity in a precedence declaration, as is illustrated below.

conj *less greater*
left *and*
left *plus or*

All operators of the same precedence have the same associativity. There are expressions that cannot be parsed without this restriction. For example, if + and − have the same precedence but − associates to the left and + to the right, $x - y + z$ cannot be parsed because of the conflict in associativity. The restriction is observed unobtrusively anytime a precedence declaration is given by virtue of its form.

1.2.5. Overloading

A function name may be *overloaded*, in that it may stand for functions of different types. For example, in the definitions of types *int* and *real*, there might be two separate definitions for multiplication:

$times: (i: int;\ \ j: int)\, int$ **infix** "*"
$times: (i: real;\ \ j: real)\, real$ **infix** "*"

so that *times* becomes overloaded.

As mentioned in Sect. 1.2.3, it is possible to express that an operation is well-formed only if each overloaded function name, in a collection of such names, stands for a function of a particular type. For example, consider an exponentiation operation *expon* such that $expon\,(e,\ j)$ is e multiplied by itself j times, for some natural number j. This operation is meaningful for any e whose type is one for which *times* is defined, so it can be used to raise matrices, say, to some power, as well as real numbers. Therefore, we say it is bounded polymorphic. Below is a Polya definition of *expon* that captures the fact that an instance $expon\,(e,\ j)$ is well-formed only if *times* is defined on the type of e:

$expon\,(e: *t;\ j: nat)\,(times: *t \rightarrow *t \rightarrow *t)\, *t$ **infix** "**"

The term $(times: *t \rightarrow *t \rightarrow *t)$ is called a *constraint*.

A definition of an overloaded name may involve a constraint on itself. For example, we might overload relational operator *at_most* ($\leq$) to also stand for the lexicographic ordering on a sequence, but only if *at_most* is defined on the elements of the sequence:

$at\_most\ \ (x: seq(*t);\ y: seq(*t))\,(at\_most: *t \rightarrow *t \rightarrow bool)\, bool$ **infix** "<="

The following type assumptions correspond to the definitions of *expon* and *at_most*.

$expon: \forall\alpha.\,(times: \alpha \rightarrow \alpha \rightarrow \alpha).\, \alpha \rightarrow nat \rightarrow \alpha$
$at\_most: \forall\alpha.\,(at\_most: \alpha \rightarrow \alpha \rightarrow bool).\, seq\,(\alpha) \rightarrow seq\,(\alpha) \rightarrow bool$

Not all type assumptions created through overloading are admitted to the type environment. There are four conditions that a type assumption must satisfy before it is admitted:

(1) Its addition to the environment must not create an *overlapping* set of type assumptions. This ensures that every occurrence of an overloaded function name in a program can be resolved uniquely. Two assumptions $h: \forall\bar{\alpha}.\,\rho\,.\tau$ and $h: \forall\bar{\gamma}.\,\rho'.\tau'$ overlap if τ and τ' are unifiable. For example, the type assumptions below overlap.

$expon: \forall\alpha.\,(times: \alpha \rightarrow \alpha \rightarrow \alpha).\, \alpha \rightarrow nat \rightarrow \alpha$
$expon: real \rightarrow nat \rightarrow real$

(2) It must be free of *unnecessary constraints*. An assumption $h: \forall\bar{\alpha}.\,\rho\,.\tau$ is free of unnecessary constraints if for every constraint $x: \tau'$ in ρ, τ' contains at least one type variable, and at least one type variable that it contains occurs in τ. If neither of these conditions holds then it can be decided whether the constraint is satisfiable by inspecting all definitions of x. For example, the constraint in the assumption

$expon: \forall\alpha.\,(times: \alpha \rightarrow \alpha \rightarrow \alpha).\, real \rightarrow real \rightarrow real$

is unnecessary because the bound variable α appears in the constraint only. The constraint demands that *times* be of type $\tau \rightarrow \tau \rightarrow \tau$ for some type τ, which can be decided by inspecting all definitions of *times*. With respect to the previous two definitions of *times*, for example, we see that the constraint can be satisfied with $\tau = int$ or $\tau = real$. Likewise, it would be pointless to specify the constraint $times : real \rightarrow real \rightarrow real$ because all definitions of *times* can be examined to see if any one of them has this type. Unnecessary constraints might be automatically eliminated by the system as a preprocessing step.

(3) Its addition to the environment must not create a *cyclic* (mutually-recursive) set of type assumptions. A cyclic set of assumptions is one that contains a sequence of assumptions of the form

$$h_0 : \forall\bar{\alpha}. (h_1 : \tau_0). \rho_0 \,,\quad h_1 : \forall\bar{\alpha}. (h_2 : \tau_1). \rho_1 \,, \ldots, \quad h_{n-1} : \forall\bar{\alpha}. (h_0 : \tau_{n-1}). \rho_{n-1}$$

for $n > 1$, and where the h_i's are not necessarily distinct names. See (4) below for the condition that applies when $n = 1$.

(4) If it is recursive, that is, has the form $h : \forall\bar{\alpha}. (h : \tau). \rho$, then τ is not a *substitution instance* of τ', where τ' is the constraint-free (data-type) part of ρ. In other words, there is no substitution of types for type variables that when applied to τ' yields τ. Recursive assumptions for which this is not true are useless. Suppose τ is a substitution instance of τ' and that assumption $h : \forall\bar{\alpha}. (h : \tau). \rho$ is in the type environment. Any attempt to add an assumption of the form $h : \forall\bar{\gamma}. \rho'. \tau''$ to the environment will fail if τ and τ'' are unifiable. The reason is because τ' and τ'' can be unified as a consequence of τ and τ'' being unifiable and τ being an instance of τ'. So the new assumption fails to satisfy condition (1). As a result, it is undecidable whether constraint $(h : \tau)$ of the recursive assumption is satisfiable, for the only assumption that can be used to determine this is the recursive assumption itself.

An overloaded function name may have at most one concrete syntax. That is, the sentential forms derived from the various concrete syntax definitions for an overloaded function name must generate the same set of terminal strings. This rule isn't as restrictive as it appears. After all, the purpose of overloading function names is to be able to use a single concrete syntax to stand for similar operations on different types. Specifying different concrete syntax definitions for instances of an overloaded name defeats the purpose of overloading.

One of our design objectives was to ensure that an expression has a uniform syntactic interpretation in all contexts. With $+$ and $*$ each overloaded, for example, whether the expression $x + y * z$ associates to the left or right should not depend on the types of x, y, and z. One should be able to parse it regardless of type. To this end, different precedences cannot be assigned to instances of an overloaded function name. An overloaded name may appear only once in a precedence declaration which effectively specifies the same precedence for all instances.

Acknowledgments

We would like to thank Geoffrey Smith for his help in designing the Polya type system, and Steve Jackson and Hal Perkins for their comments on the design of this facility.

References

[1] Demers, A. and Donahue, J. Data types, parameters and type checking *7th ACM Symposium on Principles of Programming Languages*, 1980.

[2] Gries, D. and Volpano, D. The transform - a new language construct *Structured Programming*, vol. 11, 1, January 1990.

[3] Mitchell, J. and Harper, R. The essence of ML *15th ACM Symposium on Principles of Programming Languages*, 1988.

Appendix

Polya has no built-in data types. Commonly-used types, often built into the definitions of other imperative languages, have no special status and are defined with type definitions like any other data type. Below are examples of type definitions for some commonly-used data types.

```
type char
literals
printable_chr (" "..~)–chr as "'" chr "'" ;          A printable ASCII character
nonprintable_chr (0 1)–octal2                         A nonprintable ASCII character
                 (0..7)–octal1
                 (0..7)–octal0
  as "'\\" octal2 octal1 octal0 "'" ;
newline as "'\\n'" ;                                  A newline \012
backslash as "'\\\\'" ;                               A backslash \134
double_quote as "'\\"'" ;                             A double quote \042
form_feed as "'\\f'" ;                                A form feed \014
tab as "'\\t'" ;                                      A tab \011
carriage_return as "'\\r'"                            A carriage return \015

operations
{Relational operators defined relative to the total ordering of ASCII
character codes. Note that these are conjunctive, e.g. a < b ≤ c ≡
a < b ∧ b ≤ c }
------------------------------------------------------------------------------
less (i:char; j:char) bool infix "<" ;                <
at_most (i:char; j:char) bool infix "<=" ;            ≤
greater (i:char; j:char) bool infix ">" ;             >
at_least (i:char; j:char) bool infix "=>" ;           ≥
equal (i:char; j:char) bool infix "=" ;               =
unequal (i:char; j:char) bool infix "!="              ≠
end
```

type *real*
literals

decimal [(0..9)]–*i* **as** *i*;	Conventional integer in decimal notation
float [(0..9)]–*whole* [(0..9)]–*fraction* { { (+ –)–*sign* } [(0..9)]–*expon* } **as** *whole* "." *fraction* { "E" { *sign* } *expon* }	Conventional floating-point format

operations

{Unary operators}

negate (*i*:*real*) *real* **prefix** "–" ;	Unary minus

{Binary operators}

plus (*i*:*real*; *j*:*real*) *real* **infix** "+" ;	Addition
minus (*i*:*real*; *j*:*real*) *real* **infix** "–" ;	Subtraction
times (*i*:*real*; *j*:*real*) *real* **infix** "*" ;	Multiplication
divide (*i*:*real*; *j*:*real*) *real* **infix** "/" ;	Division
expon (*i*:*real*; *j*:*real*) *real* **infix** "**" ;	Exponentiation

{Relational operators. Note that these are conjunctive, e.g. $a < b \le c \;\equiv\; a < b \wedge b \le c$ }

less (*i*:*real*; *j*:*real*) *bool* **infix** "<" ;	$<$
at_most (*i*:*real*; *j*:*real*) *bool* **infix** "<=" ;	$\le$
greater (*i*:*real*; *j*:*real*) *bool* **infix** ">" ;	$>$
at_least (*i*:*real*; *j*:*real*) *bool* **infix** "=>" ;	$\ge$
equal (*i*:*real*; *j*:*real*) *bool* **infix** "=" ;	$=$
unequal (*i*:*real*; *j*:*real*) *bool* **infix** "!=" ;	$\ne$

{Supplementary functions}

abs (*i*:*real*) *real*;	Absolute value
max (*i*:*real*; *j*:*real*) *real* **infix** "*max*"	Maximum of i and j
min (*i*:*real*; *j*:*real*) *real* **infix** "*min*" ;	Minimum of i and j
maxlist [**exp**–*i*] (*i*:*real*) *real* **as** *MAX* "(" [*i*]–, ")" ;	Maximum of a list of values
minlist [**exp**–*i*] (*i*:*real*) *real* **as** *MIN* "(" [*i*]–, ")" ;	Minimum of a list of values

{Quantifications. In the following, i stands for a list of bound variables, or "dummies", r for "range", and t for "term". For each expression, we give an informal English wording of it.}

sum [**id**–*i* **type**–*s*] **exp**–*r* **exp**–*t* (*i*:*s*; *r*:*bool*; *t*:*real*; (*i*)*r*; (*i*)*t*) *real* **as** "*SUM*" "(" [*i* ":" *s*] ":" *r* ":" *t* ")";	Sum of t for all i in range r: $+ / \{\lvert i : r : t \rvert\}$.
product [**id**–*i* **type**–*s*] **exp**–*r* **exp**–*t* (*i*:*s*; *r*:*bool*; *t*:*real*; (*i*)*r*; (*i*)*t*) *real* **as** "*PROD*" "(" [*i* ":" *s*] ":" *r* ":" *t* ")";	Product of t for all i in range r: $* / \{\lvert i : r : t \rvert\}$.
max [**id**–*i* **type**–*s*] **exp**–*r* **exp**–*t* (*i*:*s*; *r*:*bool*; *t*:*real*; (*i*)*r*; (*i*)*t*) *real* **as** "*MAX*" "(" [*i* ":" *s*] ":" *r* ":" *t* ")";	Maximum of t for all i in range r: $\mathbf{max} / \{\lvert i : r : t \rvert\}$.
min [**id**–*i* **type**–*s*] **exp**–*r* **exp**–*t* (*i*:*s*; *r*:*bool*; *t*:*real*; (*i*)*r*; (*i*)*t*) *real* **as** "*MIN*" "(" [*i* ":" *s*] ":" *r* ":" *t* ")"	Minimum of t for all i in range r: $\mathbf{min} / \{\lvert i : r : t \rvert\}$.

end

A sequence s of type $seq(t)$ is thought of as a function from a prefix of the natural numbers to type t. The size $\#s$ of s is the length of the prefix. Hence, the sequence is viewed as $s.0, s.1, \ldots, s.(\#s-1)$.

type *seq* (*t*)

operations

{Sequence construction}	
make_seq { [**exp**−*e*] } (*e* :*t*) *seq*(*t*) **as** "<" { [*e*]−, } ">";	Sequence consisting of the expressions *e*
make_seq [**id**−*i* **type**−*s*] **exp**−*r* **exp**−*t* (*i* :*s*; *r* :*bool*; *t*:**tl*; (*i*)*r*; (*i*)*t*) *seq*(**tl*) **as** "<" [*i* ":" *s*] ":" *r* ":" *t* ">" ;	The sequence consisting of the values t for all i such that r holds. The elements are in ascending ordered with respect to the lexicographic ordering of elements in $s_0 \times \cdots \times s_{n-1}$.

{Unary operators}	
size (*s* :*seq*(*t*)) *nat* **prefix** "#" ;	Number of elements in s

{Binary operators}	
cat (*e* :*seq*(*t*) ; *s* :*seq*(*t*)) *seq*(*t*) **infix** " ^ " ;	Catenation of sequences
occurrences (*v* :*t*; *e* :*seq*(*t*)) *nat* **infix** "#"	Number of occurrences of v in e

{Relational operators. Note that these are conjunctive}	
member (*e* :*t* *s* :*seq*(*t*)) *bool* **infix** "*in*" ;	Is member of: $e \; in \; s \equiv 0 < e \, \# s$
not_member (*e* :*t*; *s* :*seq*(*t*)) *bool* **infix** "*notin*";	Is not member of: $e \; notin \; s \equiv \neg(v \; in \; s)$

{Index operations}	
index (*s* :*seq*(*t*); *i* :*nat*) *t* **as** *s* "(" *i* ")";	Element number i of s
subseq (*s* :*seq* (*t*); *i* :*int*; *j* :*int*) *seq* (*t*); as *s* "(" *i* ".." *j* ")";	Subsequence of s
subseqlast (*s* :*seq* (*t*); *i* :*int*) *seq*(*t*); **as** *s* "(" *i* ".." ")";	$s\,(i..)$
first (*s* :*seq*(*t*)) *t*;	$= s.0$
rest (*s* :*seq*(*t*)) *seq*(*t*);	$= s.(1..)$
last (*s* :*seq*(*t*)) *t*;	$= s.(\#s-1)$

{Distributive operators}	
distr (*f* :*t*−>*t*−>*t*; *b* :*seq*(*t*)) *t* **infix** "/" ;	Let $f : t \to t \to t$ be associative. Let s be a nonempty sequence of type $seq(t)$, with elements $s.0, \; s.1, \; \ldots, \; s.(\#s-1)$. Then $f / s = s.0 \, f \, s.1 \, f \cdots f \, s.(\#s-1)$. In addition, if f has a unit e (say), then $f / < > = e$.
distru (*f* :*t*−>*t*−>*t*) *seq*(*t*)−>*t* **prefix** "/"	If f is associative and symmetric, $/f$ yields a function that distributes f over a sequence: $/f = \mathbf{fn}(s : seq\,(t)) f / s)$.

end