Theta Reference Manual

Preliminary Version

Barbara Liskov
Dorothy Curtis
Mark Day
Sanjay Ghemawat
Robert Gruber
Paul Johnson
Andrew C. Myers

Programming Methodology Group Memo 88
MIT Laboratory for Computer Science
Cambridge, MA 02139

February 8, 1995

This document describes a new programming language called Theta. Theta is a sequential, strongly-typed, object-oriented language. It provides separate mechanisms for type hierarchy, inheritance, and parametric polymorphism. It also provides separate mechanisms for specifications, which define the interfaces of new abstractions, and code that implements the new abstractions, and it allows multiple implementations of types and routines. It has a module mechanism that encapsulates the details of type and routine implementations, while allowing related implementations to share implementation-specific information. Theta is largely derived from CLU, but has also been influenced by Trellis/Owl, Modula-3, C++, and Emerald.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158.

The authors also wish to thank all the members of the Programming Methodology Group at MIT for their help and suggestions; there were many group meetings that helped us refine the Theta design.

Contents

1 Overview of the Language

Theta is an object-oriented language that was developed for use within the Thor object-oriented database system [1, 2], and its origins within Thor have had a major impact on its design. The requirement that code run safely inside Thor has led to the inclusion of several features, such as strong, static type checking and automatic garbage collection.

Theta is an extensible language in which users can define and implement new, abstract types and new routines. New types and routines are defined by *specifications*, which describe the interface of the new abstraction but give no implementation details; a type specification defines the names and signatures of the methods of objects of that type. Types are implemented by *classes*, and a type can have multiple implementations. Classes and routine implementations are grouped within *modules* that encapsulate them. Modules ensure that an object or routine can be accessed only through its public interface, but allow related code to share implementation-specific information.

Theta provides separate mechanisms for type hierarchy, parametric polymorphism, and inheritance. The type hierarchy mechanism allows the definition of families of types with similar behavior; types can have multiple supertypes. The inheritance mechanism is separate from hierarchy, so that related types can have independent implementations and unrelated types can have related implementations; only single inheritance is supported. The parametric polymorphism mechanism supports generic code that is independent of the hierarchy mechanism, and does not require recompilation for different instantiations.

The characteristics of Theta are discussed further in the rest of this chapter.

1.1 Objects

Theta programs run in a universe of objects. Each object in the universe has a unique identity, an encapsulated state, and a set of methods that can be called to interact with it. Objects belong to types that define the names and signatures of their methods.

Theta provides a rich set of built-in types, and users can define new, abstract types. Users can also define new routines. Methods and routines can be either procedures or iterators, and when called may either terminate normally or by signaling an exception. Both types and routines can be parameterized on one or more parameter types.

Objects come into existence as a result of calls to certain methods and routines that create new objects. Storage for objects that can no longer be accessed is reclaimed automatically by the garbage collector.

1.2 The Theta Type System

Theta types are arranged in a hierarchy in which every type can have several supertypes. At the top of the hierarchy is the built-in type any; any is the supertype of all types. When a new type is defined, its definition indicates its supertypes. The Theta compiler ensures *conformance*: a subtype is guaranteed to have the methods of its supertypes, and these methods have compatible signatures. The type any has no methods, and therefore it imposes no constraints on its subtypes. A supertype's methods can be renamed so that the new names match what is needed in the subtype *(9.2)*.

Theta provides strong, static type checking. Every variable has a declared type *(4.5)* that determines the type of object it can refer to. A variable is guaranteed to refer to an object whose actual type is a subtype of the variable's type *(5.1)*. Every routine and method has a declared signature that determines the types of its arguments and results. The compiler ensures that all calls and assignments are consistent with this declared information. The conformance rule ensures that any call that is legal according to a variable's type will be legal for the actual object denoted by the variable. The exact type of a variable's object is not known at compile time, but more exact information can be determined by using the **typecase** statement *(8.12)*, which tests an object's type at runtime.

Type hierarchy provides *subtype polymorphism* and allows the definition of routines that are generic with respect to their arguments and results, and also of data structures that are generic with respect to their elements: in each case the actual objects can belong to a subtype of what is declared.

1.3 Specifications

New types and routines are defined by providing *specifications (9)*. A specification defines interface information; it does not include any information about how the new type or routine is to be implemented. This is given separately, and a type or routine can have many different implementations.

The specification of a type defines the supertypes, and the names and signatures of the methods. A type can have multiple supertypes, and the methods of the supertype can be renamed in the subtype.

A type specification only defines the methods of the type's objects and thus provides no way to create new objects "from scratch." Creation is accomplished through the use of routines that are not part of the type. Keeping creation separate from the type means that a subtype's creators need not be related to those of its supertypes, and it allows different implementations of the type to create objects differently. For example, a hash-table based implementation of a map abstraction would have a creator that takes the hashing function as an argument, while an array-based implementation would have a creator that takes no arguments.

1.4 Classes and Inheritance

A type is implemented by one or more *classes (10.4)*. A class contains some instance variables that store the state of objects of the type, and routines that implement the object's methods. The class can also contain implementations of "private" methods that are not available to external code that uses its objects.

A class can inherit from a single superclass *(10.5)*. Objects of a subclass contain superclass instance variables and methods. Superclass methods can be renamed in the subclass. Code within a subclass cannot access the inherited instance variables directly; instead, superclass methods are used to access these variables.

A class definition indicates whether subclasses are permitted, and if so, what methods and associated routines they can use *(10.4)*. A special kind of operator called a *maker* must be exported so that subclasses can initialize the inherited instance variables *(10.5.2)*; in addition private methods can be exported, and public methods can be hidden *(10.5.1)*. This mechanism allows users to provide a rich interface for subclasses, while ensuring that subclasses cannot damage superclass objects *(10.5.4)*.

1.5 Modules

Classes and routine implementations are placed inside modules *(10.1)*. Each module *(10)* implements one or more specifications. For example, a typical module contains a class implementing a type plus implementations of routines that create objects of the type.

A module encapsulates the contained code; details of any classes it contains are visible only to other code in the module. However, these details are fully exposed to code within a module. Thus, routine implementations in a module have full access to internal details of classes within the module. Similarly, classes within the same module can take advantage of implementation-specific details of one another.

1.6 Parametric Polymorphism

In addition to subtype polymorphism, Theta supports *parametric polymorphism*, in which a routine, type, or class definition is parameterized by one or more types *(9.3)*. Parametric polymorphism allows the actual parameter type to be selected by the user, when the type or routine is *instantiated (3.2.2)*. For example, array is a parameterized type, with instantiations array[int], array[char], and so on.

Parameters can be constrained to be types whose objects have certain methods with certain signatures. These constraints permit parameterized code to be compiled without knowledge of the actual instantiation types. Generic code needs to be compiled only once.

Subtype polymorphism is useful for defining generic behavior over a set of related types, while parametric polymorphism is useful for defining generic behavior where the actual parameters need not be related in the type hierarchy.

1.7 Programs

A Theta program consists of a group of *program units*. A program unit is a specification *(9)*, equate *(4.6)*, or module *(10.1)*. Specifications define abstract types and routines; modules provide implementations of these types and routines. Equates provide abbreviations for constants, e.g., the name pi might denote the number 3.1416.

Within program units, *external names (4.3)* are used to refer to specifications and equates. External names can be chosen locally to fit the needs of the unit that uses them; this allows different units to define entities with the same name without causing a global name conflict. For example, two different specifications could both define a type named int_set.

Theta compilers provide separate compilation for program units. The compiler makes use of the specifications denoted by external names of the unit being compiled to ensure that the type or routine is used in a type-correct manner. During the *linking* process, an implementation is selected for each of these types and routines.

This manual does not define the mechanisms for interpreting external names during compilation and linking. A compiler might process a file containing many units such that every external name is defined somewhere in the file, and every specification in the file is implemented in some module in the file. Alternatively, objects recording the meanings of exported names might be stored persistently (*e.g.*, in Thor); in this case the compiler might process an individual unit in a context that associates its external names with appropriate specification objects, and the linker might select routine implementations using a context that associates external names with appropriate implementation objects.

2 Lexical Considerations

2.1 Notation

We use an extended BNF grammar to define the syntax. The general form of a production is:

$$\textit{nonterminal} \quad \rightarrow \quad \textit{alternative} \mid \textit{alternative} \mid \textit{alternative} \mid \ldots$$

The following form denotes that *optional* can appear 0 or 1 times.

[*optional* **]**

The following form denotes that *c*an appear 0 or more times.

[*optional* **]***

The full Theta reference grammar is given in Appendix A. Productions from this grammar are used freely in the following chapters to give the general form for different Theta constructs in a concise manner. When the productions relevant to a given construct are presented, some nonterminals are naturally left undefined; the interested reader can find the productions for these nonterminals in the reference grammar.

2.2 Lexical Considerations

2.2.1 Case Insensitivity

Case does not matter in Theta. For example, THEN, then, and Then are all the same reserved word.

2.2.2 Tokens and Separators

A module is written as a sequence of tokens and separators. A *token* is a sequence of "printing" ASCII characters representing a reserved word, an identifier, a literal, an operator, or a punctuation symbol. A *separator* is a "blank" character (a space, vertical tab, horizontal tab, carriage return, newline, or form feed) or a comment. Zero or more separators may appear between any two tokens, where at least one separator is required between any two adjacent non-self-terminating tokens: reserved words, identifiers, integer literals, and real literals. Tokens are described in more detail in the following sections.

2.2.3 Comments

A *comment* begins with a percent sign (%) and ends with the first "line-ending" character (a vertical tab, carriage return, newline, or form feed). The enclosed characters serve as a single separator and are otherwise ignored by the compiler.

2.3 Reserved Words

The following tokens are reserved words:

begin	**if**	**returns**
bind	**implements**	**same_object**
break	**in**	**self**
class	**inherits**	**signal**
continue	**iter**	**signals**
do	**make**	**tagcase**
else	**makes**	**then**
elseif	**module**	**type**
end	**others**	**typecase**
except	**proc**	**when**
exit	**provides**	**where**
for	**resignal**	**while**
has	**return**	**yield**
hides		**yields**

In addition, the following tokens are reserved words because they are the names of the built-in types and parameterized types, or the names of literals for these types:

any	maybe	sequence
array	nil	string
bool	null	struct
char	oneof	true
false	real	vector
int	record	

The token failure has a special reserved meaning when used as an exception name *(8.14)* but is not a reserved word.

2.4 Identifiers

Identifiers are sequences of letters, digits, and underscores that begin with a letter or underscore.

2.5 Literals

There are literals for naming objects of the built-in types null, bool, int, real, char, and string (see Section 7.1 and also Appendix B).

2.6 Operators and Punctuation Symbols

A number of tokens are used as operators and punctuation symbols. The following table lists all of the operators and punctuation of Theta. Many of these tokens are used as a shorthand for various method invocations *(7.11)*. For each single-character token, the table gives its hexadecimal ASCII code. (The notation b_c is the Theta notation for integer constant c given in base b *(B.4)*.)

<table>
<tr><td>"</td><td>16_22</td><td><</td><td>16_3c</td></tr>
<tr><td>&</td><td>16_26</td><td><=</td><td></td></tr>
<tr><td>'</td><td>16_27</td><td>=</td><td>16_3d</td></tr>
<tr><td>(</td><td>16_28</td><td>></td><td>16_3e</td></tr>
<tr><td>)</td><td>16_29</td><td>>=</td><td></td></tr>
<tr><td>*</td><td>16_2a</td><td>[</td><td>16_5b</td></tr>
<tr><td>**</td><td></td><td>]</td><td>16_5d</td></tr>
<tr><td>+</td><td>16_2b</td><td>^</td><td>16_5e</td></tr>
<tr><td>,</td><td>16_2c</td><td>;</td><td>16_6b</td></tr>
<tr><td>-</td><td>16_2d</td><td>{</td><td>16_7b</td></tr>
<tr><td>.</td><td>16_2e</td><td>|</td><td>16_7c</td></tr>
<tr><td>..</td><td></td><td>||</td><td></td></tr>
<tr><td>/</td><td>16_2f</td><td>}</td><td>16_7d</td></tr>
<tr><td>//</td><td></td><td>~</td><td>16_7e</td></tr>
<tr><td>:</td><td>16_3a</td><td>~=</td><td></td></tr>
<tr><td>:=</td><td></td><td>\</td><td>16_5c</td></tr>
</table>

The following "printing" ASCII characters are not used in the language:

!	exclamation	16_21	?	question-mark	16_3f
#	pound-sign	16_23	@	at-sign	16_40
$	dollar-sign	16_24	`	back-quote	16_60

The ASCII character % (ASCII 16_25) is used for comments *(2.2.3)*.

3 Types and Parameterized Types

A *type* consists of a set of *objects*, along with a collection of *methods* that belong to each of these objects. A method can access and manipulate its object's state. If none of an object's methods modify its *abstract state* (the state that can be observed using its methods), we say the object is *immutable*; otherwise it is *mutable*. We say a type is immutable (mutable) if its objects are immutable (mutable).

A *parameterized type* defines a family of related types. For example, the array parameterized type defines the types array[int], array[char], array[array[int]], and so on. Each type in the family is obtained by *instantiating* the parameterized type *(3.2.2)*, producing a type referred to as an *instantiation*.

Theta has a number of built-in types and parameterized types. In addition, users can define new abstract types and parameterized types *(9, 10)*.

Theta has strong static type checking. Types are arranged in a type hierarchy: a type can be a subtype of several other types *(3.4)*. The hierarchy for the built-in types is quite flat *(3.4)*, except that there is a rich hierarchy for routine types *(3.4.1)*. Users define the hierarchy for the user-defined types *(9.2)*. The type hierarchy determines the legality of assignment *(5)* and invocation *(6.1)*.

A *type designator* denotes a type. For example, a type designator can be the name of an ordinary type (int, color), or an instantiation of a parameterized type (array[int], maybe[employee]).

The Theta compiler requires complete information about the type denote by a type designator. For example it needs to know all methods of the type and their signatures, and also all supertypes of the type. This means that any identifiers used as type designators must either be defined in the current program unit or in the external compilation environment. If the designator denotes a user-defined type, all supertypes of the type must also be defined either locally or externally.

3.1 Built-in Types

The built-in types are null, any, bool, char, int, real, string, and all routine types. The built-in parameterized types are array, sequence, vector, record, struct, oneof, and maybe.

The type any has no methods and is used as the top of the type hierarchy *(3.4)*: all types, both built-in and user-defined, are subtypes of type any.

The type null has one literal, nil; it is typically used as a placeholder in oneof types. The type bool is a conventional Boolean type, with literals true and false. The type int represents a subrange of the mathematical integers, and the type real represents a finite-precision approximation to a subrange of the mathematical real numbers. The type char represents characters (typically ASCII), and the type string represents strings of characters (again, typically ASCII).

The parameterized types array, vector, and sequence are homogeneous collections indexed by consecutive integers. Array and vector are mutable types; a vector has a fixed size, while an array can grow and shrink dynamically. Sequence is an immutable type. The special types record and struct are heterogeneous tuples with names as selectors of fields; record is a mutable type, while struct is immutable. The special type oneof is a named, immutable, discriminated union. The parameterized type maybe is a special case of oneof which is used to represent an object that contains either an object of the parameter type, or the value nil.

Routine types are also built-in types in Theta. Routines (i.e., the objects of these types) are defined by routine specifications *(9)* and routine implementations *(10)*. Routines are first-class objects that can be stored in data structures and passed as arguments and results.

The full specifications of the built-in types appear in Appendix B. Every Theta implementation will provide implementations of these types. Appendix C contains specifications of some additional types that will be provided by most Theta implementations.

3.2 Type Designators

There are four different kinds of types in Theta, each with its own particular type designator form. These forms are explained in the following sections. Type designators are also defined by *equates (4.6)*.

3.2.1 Simple Types

Simple types are defined by non-parameterized type specifications *(9)*. They include all non-parameterized user-defined types and several of the built-in types. A simple type is designated by its name:

simple_type_desig → *idn* | null | bool | char | int | real | string | any

3.2.2 Parameterized Type Instantiations

A parameterized type has one or more type parameters. A type designator for such a type denotes an instantiation by providing an actual type for each type parameter:

parm_type_desig → *parm_type* [*type_list*]

where

parm_type → *idn* | **array** | **sequence** | **vector** | **maybe**

type_list → *type_designator* **[** , *type_designator* **]***

The specification *(9.3)* of a parameterized type can use **where** clauses to require that actual parameters have certain methods. An instantiation of a parameterized type is legal provided it has the right number of actual parameters, and the actual parameters satisfy the restrictions of the **where** clauses. For example, consider a user-defined parameterized type, set[T]; since sets do not contain duplicate, the specification for set permits instantiation only if the argument type provides an equality method (which allows duplicates to be recognized):

set = **type** [T] **where** T **has** equal: **proc** (T) **returns** (bool)

Here are some instantiations of set:

```
set[int]            % supplies the int equal method for T's equal
set[array[int]]     % supplies the array[int] equal method for T's equal
set[int,bool]       % not legal – compile–time error
set[employee]       % legal only if employee has an equal method
```

The third instantiation is not legal because it does not supply the right number of parameters. The last instantiation will be legal only if type employee has an equal method; otherwise there will be a compile-time error.

Some methods of a parameterized type may place additional constraints on a parameter by having **where** clauses of their own. Such a method is *optional*: if an instantiation satisfies its requirements, the resulting type will have the method, otherwise it will not. When such a parameterized type is instantiated, methods are selected to satisfy the constraints of the optional methods if possible. The result is a type with all the non-optional methods, plus any of the optional methods whose constraints are satisfied. For example, array *(B.8)* has an optional copy method that requires that the actual parameter have a copy method. Here are some instantiations of array:

```
array[int]                % has a copy method
array[employee]           % may not have a copy method
```

If type employee does not have a copy method, the second instantiation results in an array type that does not have a copy method.

3.2.3 Routine Types

There are two kinds of routines, procedures and iterators (see Section 6.1). Routine type designators have the following special form:

routine_type_desig	→	**proc** *nonparam_proc_sig* | **iter** *nonparam_iter_sig*
nonparam_proc_sig	→	(**[** *type_list* **]**) **[** *returns* **]** **[** *signals* **]**
nonparam_iter_sig	→	(**[** *type_list* **]**) *yields* **[** *signals* **]**
type_list	→	*type_designator* **[** , *type_designator* **]***

These type designators indicate the kind of routine, and the types and numbers of the arguments, results, and exceptions. For example:

proc(int, int) **returns** (bool) **signals** (negative)
iter(stree[int]) **yields** (int)

3.2.4 Tagged Types

Tagged types include the record, struct, and oneof types. They are special parameterized types that possess named fields and are designated by the following special form:

tagged_type_desig	→	*tagged_type* [*field* **[** , *field* **]***]
tagged_type	→	record | struct | oneof
field	→	*idn_list* : *type_designator*

These type designators provide a name for each field and give its type. For example

```
record[x, y: int, s: string]
```

defines a record type. Records of this type have three fields: two int fields named x and y, and a string field named s.

3.3 Type Equality

Type checking in Theta occurs at compile time and is based on analysis of type designators to determine whether they designate the same type. Two type designators are equal if the designate the same type and otherwise they are unequal.

Type equality is defined as follows:

1. Each simple type is equal only to itself.

2. Two types obtained by instantiation are equal if they are instantiations of the same (user-defined or built-in) parameterized type and their parameters are pairwise equal.

3. Two routine types are equal if: both are **proc** or both are **iter**; both have the same number of arguments, and types of matching arguments are equal; both have the same number of results (yielded results for iterators), and the types of matching results are equal; both have the same exceptions, and matching exceptions have the same number and types of results.

4. Two tagged types are equal if each has the same *tagged_type*, the same field names in the same order, and the type of matching field names are equal. E.g., these two types are not equal:

   ```
   record[a: int, b: int]
   record[b: int, a: int]
   ```

3.4 Type Hierarchy

Types in Theta are grouped into a type hierarchy. Each type can have several supertypes. At the top of the hierarchy is type any, which has no methods and is the supertype of all types.

The type hierarchy is based on the notion of type equality. A type is always a subtype and also a supertype of itself. The subtype (supertype) relation is transitive: if T is a subtype of S and S is a subtype of R, then T is a subtype of R.

Any is the only supertype of the built-in types except that a rich hierarchy is defined for routine types. In particular, no subtype relation is provided for record and struct types, e.g.,

```
record[a: int, b: int, c: int]
```

is not a subtype of

```
record[a: int, b: int]
```

Specifications for user-defined types and user-defined parameterized types indicate the immediate supertypes explicitly *(9.2)*. All supertypes must be user-defined types, except that every user-defined type is automatically a subtype of any. All types generated by instantiating a user-defined parameterized type are similarly subtypes of any. The Theta compiler guarantees that subtypes have all the methods of their supertypes, with compatible signatures *(9.2)*.

3.4.1 Routine Type Hierarchy

Routine type R_1 is a subtype of routine type R_2 if all the following conditions are satisfied:

1. The two routine types must either both be procedure types or both be iterator types.
2. *Contravariance of arguments*: They must have the same number of arguments, and in each argument position, the type of R_1's argument must be a supertype of the type of R_2's argument.
3. *Covariance of results*: If the two routine types are procedure types, they must have the same number of return results, and the types of R_1's results must be subtypes of the types of the corresponding R_2 results.
4. *Covariance of yields*: If the two routine types are iterator types, they must have the same number of yielded results, and the types of R_1's yielded results must be subtypes of the types of the corresponding R_2 yielded results.
5. R_1 must not have any exceptions that are not also exceptions of R_2.
6. *Covariance of exception results*: Corresponding exceptions must have the same numbers of results, and in the corresponding result positions, the type of R_1's result must be a subtype of the type of R_2's result.

These rules ensure that calling a routine of type R_1 is always legal wherever a routine of type R_2 is expected. In particular, the routine will not raise any unexpected exceptions.

4 Scopes, Declarations, and Equates

This chapter gives the scoping rules for Theta. It also describes variable declarations and equates, two important constructs that introduce scoped identifiers.

When we say that an identifier is scoped, or that it has a scope, we mean it is *defined* within a particular scope. The scoping rules given below *(4.4)* ensure that a given identifier is either not defined within a scope or is defined exactly once, and that external names *(4.3)* are never confused with scoped identifiers within a program unit *(1.7)*.

4.1 Scoped Identifiers

There are two kinds of scoped identifiers in Theta: equated identifiers and variables. An equated identifier is immutable and denotes the same object for its entire lifetime. A variable is mutable; it can be modified so that it denotes a different object. When a variable is created, it may not denote any object.

Equated identifiers are introduced by equates *(4.6)*, type specifications *(9.2)*, routine implementations *(10.2)*, method implementations *(10.4)*, formal parameter declarations *(9.3)*, classes *(10.4)*, and superclass declarations *(10.5)*. Variables are introduced by declarations *(4.5)*, instance variable declarations *(10.4)*, formal argument declarations *(10.2)*, and special declaration forms in the **tagcase** *(8.11)* and **typecase** statements *(8.12)*.

An equated identifier has a scope that is the entire scoping unit containing the construct that introduces it, while a variable has a scope from its declaration to the end of the containing scoping unit.

4.2 Scoping Units

The scope of variable declarations and equates is defined in terms of scoping units. This section contains a complete list of the scoping units of Theta.

1. From the start of a routine or type specification *(9)* to its end.
2. From the start of a module *(10.1)*, routine implementation *(10.2)*, or class *(10.4)* to its end.
3. From a **for** *(8.8)*, **do** *(8.8, 8.7)*, **begin** *(8.13)*, or the **then** in a **make** statement *(8.17)* to the matching **end**.
4. From a **then**, **elseif**, or **else** in an **if** statement *(8.6)* to the end of the corresponding body.
5. From a **when** or **others** in a **tagcase** statement *(8.11)*, **typecase** statement *(8.12)*, or **except** statement *(8.14)* to the end of the corresponding body.

If one scoping unit overlaps another (textually), then one is fully contained in the other. The contained scope is called a *nested* scope, and the containing scope is called a *surrounding* scope.

4.3 External Names

An identifier that is used in a scope where it is not defined is an *external name*. External names are used to denote specifications and equates from other program units *(1.7)*.

4.4 Scope Rules

The scope rules are:

1. An identifier may not be defined twice in a scope. Note that this rule implies that an identifier defined in a scope may not be redefined in a nested scope.

2. Within a single program unit *(1.7)*, an identifier may not appear as an external name in one scope and as a scoped identifier in another scope.

4.5 Variables and Declarations

Objects are the fundamental runtime entities in Theta. Variables are a way of denoting or naming objects.

Declarations introduce new variables. The scope of a variable is from its declaration to the end of the scoping unit containing the declaration. Hence, variables must be declared before use.

A variable has two properties: its type (which determines what can be done with it) and the object it denotes, if any. A variable is said to be *uninitialized* if it does not refer to any object. An attempt to use an uninitialized variable at runtime causes the exception *(8.14)* failure("uninitialized variable") to be raised.

A declaration without initialization just introduces some new, uninitialized variables:

$$decl \quad \rightarrow \quad idn\_list \,:\, type\_designator$$

This statement introduces one or more new variables of the specified type. The following are examples of legal declarations:

```
x, y: int          % declare two integer variables
parts: set[part]   % declare a set[part] variable
```

As discussed in the next chapter, new variables can be declared and initialized at the same time, in an assignment statement *(5.2.1)*.

4.6 Equates

An equate allows a single identifier to be used as an abbreviation for a constant that may have a lengthy textual representation; it also allows a mnemonic identifier to be used in place of a constant such as a numerical value.

The syntax of equates is:

$$equate \quad \rightarrow \quad idn = expr \;\big|\; idn = type\_designator$$

For the first form, the *expr* must be a constant expression *(7.15)* or a compile-time error will occur.

An identifier equated to an expression may be used as an expression *(7)*; the value of such an expression is the constant to which the identifier is equated. An identifier equated to a type designator is itself a type designator and denotes the same type as the type designator it is equated to. An equated identifier may not be used on the left-hand side of an assignment *(5)*.

Some examples of legal equates:

```
as = array[string]    % a type equate
pi = 3.1416           % a constant expression equate
```

An equated identifier is defined in the smallest scoping unit surrounding its equate; here we mean the entire scoping unit, not just the portion after the equate. Equates that occur within the body of a statement must appear prior to any statements in that body. Equates not in such a scope (e.g., at the top level of a module) can appear anywhere within the scope.

All equates in a scope are processed by the compiler as a unit, and forward references are allowed. The compiler reports an error if a cyclic dependency is detected without any intervening user-defined type. For example, the following set of equates is illegal:

```
tree      = maybe[tree_node]
tree_node = record[left: tree, right: tree, val: int]
```

However, within a class implementing a tree type we might have

```
tree = class ...
   tree_node = record[left: tree, right: tree, val: int]
   t: tree_node
   ...
   end tree
```

These equates are legal because a user-defined type, tree, breaks the cycle.

5 Assignment

Assignment causes a variable to refer to an object. Assignment is a fundamental action in Theta: all other actions, including invocation *(6.1)*, depend on the rules for assignment.

5.1 Type Inclusion

Based on the declared types of variables and routine headers, the compiler can compute a type for any expression; we refer to this type as the *apparent* type, as opposed to the *actual* type of the object that results when the expression is evaluated at run time. The actual type is always a subtype of the apparent type.

An assignment

```
v := e
```

is legal if and only if the apparent type of expression e is a subtype of the type of variable v. Thus, the compiler guarantees that for any initialized variable v, the type of the object referred to by v is a subtype of the type of v.

5.2 Assignment

The simplest form of assignment assigns the value of a single expression to a single variable. In addition, there are two forms of multiple assignments, one for assigning a set of expressions to a set of variables, and one for assigning a set of return results (from a single invocation) to a set of variables. (Theta supports multi-valued routines.)

An assignment statement has one of two forms:

$$\begin{array}{lcl} \textit{statement} & \rightarrow & \textit{lhs} := \textit{expr} \ \textbf{[} \ , \ \textit{expr} \ \textbf{]*} \\ & | & \textit{lhs} := \textit{invoc} \end{array}$$

where

$$\begin{array}{lcl} \textit{lhs} & \rightarrow & \textit{var} \ \textbf{[} \ , \ \textit{var} \ \textbf{]*} \\ \textit{var} & \rightarrow & \textit{idn} \ | \ \textit{primary} \ . \ \textit{idn} \end{array}$$

A variable *var* is either an *idn*, a field selector (for a record or struct), or an instance variable; the form *primary.idn* is used to select a field of a record or struct *(B.11, B.12)*, or an instance variable of an object *(7.5, 10.4)*. (Instance variables can be accessed only within the module containing the object's class *(10.1)*.) A *primary* is a limited kind of expression *(7.16)*.

If the right hand side consists of one or more expressions, the number of expressions on the right hand must match the number of variables on the left hand side, and their types must be subtypes of the types of the corresponding variables. The primaries and expressions are evaluated in arbitrary order; if no exceptions are raised (see 8.14), the result of the first expression is assigned to the first variable and so on; as a result the variables (including fields of records, structs, and instance variables) refer to the objects obtained from evaluating the expressions. This form allows a permutation of variables, e.g.,

```
x, y := y, x
```

causes x to refer to the object previously referred to by y, and y to refer to the object previously referred to by x.

If the right hand side consists of an invocation, the number of return results must match the number of variables and the result types must be subtypes of those of the corresponding variables. The primaries on the left hand side and the invocation are evaluated in an arbitrary order and if no exceptions occur, the results of the invocation are assigned to the corresponding variables. An example of the use of this form is:

```
quotient, remainder := intdiv(a, b)
```

It is illegal to invoke a multi-valued routine in a context where a single result is expected *(6.1)*. Such a routine can only be invoked in a multiple assignment statement, or in an invocation statement *(8.1)* (where the return results are discarded).

5.2.1 Initialization Assignment

The assignment examples shown above use already-declared variables on the left hand side. Assignment can also be combined with declarations of new variables, so that new variables can be declared and initialized in one statement. In this case, the left hand side has the form

lhs → *decl* **[** , *decl* **]***

Declarations can be used with either expressions or an invocation on the right hand side, as above. If there are multiple expressions on the right hand side, they are evaluated in arbitrary order. If no exceptions result from evaluating the right hand side, the new variables are created and the result objects assigned to them. The statement is legal if the number of objects resulting from evaluation of the right hand side matches the number of variables declared and the types of these objects are subtypes of the corresponding variable types.

Note that either an assignment affects already-declared variables, or it affects newly-declared variables; the two forms are never mixed. Note also that the newly-declared variables cannot be used in the right hand side of the initialization assignment that creates them.

The following are legal initialization assignments:

```
x: int := foo + 5
c: char, i: int := 'a', 42
quotient, remainder: int  := intdiv(a, b)
val: int, in_range: bool  := search(myset, low, high)
```

provided foo, intdiv, and search have the following types:

```
foo:     int
intdiv:  proc(int,int) returns(int,int)
search:  proc(set,int,int) returns(int,bool)
```

6 Invocation

There are two kinds of routines in Theta. A *procedure* produces a group of one or more objects when it returns. An *iterator* produces a sequence of *items* (where an item is a group of one or more objects) one item at a time; it is invoked (only) in a **for** statement *(8.8)* and the body of the for statement is executed for each item in the sequence.

Invocation of a routine causes the routine to be executed on the argument objects. This section discusses the part of the invocation mechanism that is common to calls of procedures and iterators (and also makers *(10.5.2)*).

Stand-alone routines are defined by specifications *(9.1)*; such a specification may be parameterized, in which case it can be instantiated to obtain a routine. Routines are also obtained by evaluating expressions. For example, a routine can be obtained by *selecting* a method from an object *(7.7)* or by *binding* a routine to some actual arguments *(7.9)*.

6.1 Form of Invocation

Invocations have the form:

invoc → *expr0* (**[** *args* **]**)

where

args → *expr* **[** , *expr* **]*** **[** , *varying_args* **]** | *varying_args*

varying_args → .. | .. *expr* **[** , *expr* **]***

The *varying_args* form allows a variable number of arguments (including none) to be supplied; these arguments together comprise the elements of a sequence that is the last actual argument of the call, and the form is legal only when calling a routine whose last argument is a sequence.

The sequence of activities in performing an invocation is as follows:

1. The expressions *expr* (including *expr0*) are evaluated in an unspecified order.

2. The expression *expr0* must evaluate to a procedure or iterator.

3. New variables are introduced corresponding to the formal arguments of the routine being invoked (that is, a new environment is created for the invoked routine to execute in).

4. The objects resulting from evaluating the non-varying-argument *expr*s are assigned to the corresponding new variables (the formal arguments). The first formal is assigned the first actual, the second formal the second actual, and so on. The type of each expression must be a subtype of the type of the corresponding formal argument.

5. If the *varying_arg* form is being used, the last argument of the routine must be a sequence[T], and the type of each *varying_arg* must be a subtype of T. The objects resulting from evaluating the *varying_args* are used to construct a sequence[T], with the first *varying_arg* being the first element and so on, and the sequence is assigned to the last formal.

6. Control is transferred to the routine at the start of its body.

The invocation is legal in exactly those situations where there are the same number of actual arguments as formal arguments (after constructing the sequence from the *varying_args*), and the (implicit) assignments of actuals to formals are legal.

For example, if procedure p has signature

proc (int, sequence[int]) **returns** (sequence[int])

then here are some legal calls of p:

```
s: sequence[int]
s := p(0,..3,5,7)  % second argument is a sequence containing elements 3, 5, and 7
s := p(6,..)       % second argument is an empty sequence
s := p(5, s)       % second argument is the sequence s
```

6.2 Call by Sharing

The caller and called routine communicate only through the argument and result objects; routines do not have access to any variables of the caller.

After the assignments of actual arguments to formal arguments, the caller and the called routine *share* objects. If the called routine modifies a shared object, the modification is visible to the caller on return. The names used to denote the shared objects are distinct in the caller and called routine; if a routine assigns an object to a formal argument variable, there is no effect on the caller. From the point of view of the invoked routine, the only difference between its formal argument variables and its other local variables is that the formals are initialized by its caller.

6.3 Run-Time Dispatch

Although the compiler can determine whether or not an invocation is type-safe, it cannot necessarily determine exactly what code will execute. In particular, for a method call the compiler usually knows only the apparent type of the object from which the method is being selected, and not its actual type; the code to be executed is determined by the object's actual type, and the particular implementation used for that type. Therefore a method invocation may involve a runtime dispatch.

6.4 Termination

Routines can terminate in two ways: *normally*, or *exceptionally*, by signaling an exception. When a routine terminates normally, any result objects become available to the caller and may be assigned to variables or passed as arguments to other routines. When a routine terminates exceptionally, the flow of control passes to an exception handler in the caller *(8.14)*.

7 Expressions

An expression evaluates to an object in the Theta universe. This object is said to be the *result* or *value* of the expression. The simplest expressions are literals and identifiers that name their result object directly. More complex expressions are generally built up out of nested invocations of procedures. The result of such an expression is the value returned by the outermost invocation. A *primary (7.16)* is a limited kind of expression used in left hand sides of assignments *(5.2)*, and also in invocation statements *(8.1)* and store statements *(8.2)*.

An expression has a *apparent* type known at compile time. This type is derived from the types of the entities of which it is composed, e.g., the types of the variables used in it, and the types of the procedures it invokes. Compile-time type checking guarantees that the apparent type of an expression is a supertype of the object obtained by evaluating the expression.

Theta has prefix and infix operators for the common arithmetic and comparison operations, and uses the familiar syntax for array indexing (for example, a[i]). However, in Theta these notations are abbreviations for method invocations *(7.11)*. This allows familiar notation to be used for user-defined types when appropriate.

7.1 Literals

Literals denote objects of the built-in types int, real, char, string, bool, and null. The type of a literal expression is the type of the object named by the literal. Some examples of literals being assigned to variables are:

```
t:      bool   := true
f:      bool   := false
s:      string := "A string"
c:      char   := 'c'
nl:     char   := '\n'        % newline
oct47:  int    := 8_47        % octal integer
hex7e:  int    := 16_7e       % hexadecimal integer
p:      int    := −5
pi:     real   := 3.141592
avog:   real   := 6.02e23
empty:  null   := nil
```

The full syntax for literals is given in Appendix B.

7.2 Identifiers that denote objects

Identifiers that denote objects can be used as expressions. When such an identifier is used as an expression, its value is the object it denotes, and the type of the expression is the type of the identifier.

The following kinds of identifiers can be used as expressions:

- Variables are introduced by declarations *(4.5)*, which specify their type, and are caused to denote objects by means of assignments *(5)*.

- Identifiers defined by equates *(4.6)* can be used as expressions and so can identifiers that denote built-in and user-defined routine definitions *(9.1, 10.2)*. Such identifiers denote a particular (constant) object. The type of an equated identifier is the type of the object it denotes.

- The reserved word **self** is used within a method implementation to denote the method's object and its type is the class type *(10.4)* of the enclosing class *(10.4)*. It is also used within the then clause of a make statement to refer to the object being initialized and its type is the class type of the object being initialized.

7.3 Constructors

There are special forms called *constructors* that enable users to create and initialize record, struct, oneof, and maybe objects. A constructor has the form

tagged_type_desig { *field_inits* }

where

field_inits → *field_init* **[** , *field_init* **]***

field_init → *idn* := *expr*

The *tagged_type_desig* is the type of the constructed object. If a struct or record is being constructed, the component names in the field list must be exactly the field names in the *tagged_type_desig*, although the names may appear in any order in the constructor; the type of the initialization expression for a field must be a subtype of the declared type of that field. The expressions are evaluated in an unspecified order; the results form the components of the newly constructed object, which is the value of the constructor expression. For example,

```
rt = record[x: int, c: char]
x: rt := rt{c := 'A', x := 7}     % legal
x := rt{x := 7}                   % compile−time error −− not enough fields
x := rt{x := 7, d: 'A'}           % compile−time error −− misnamed field
```

If a oneof or maybe is being constructed, just one field name of the type can be present, and the type of the expression must be a subtype of the declared type of that field; the result is a new oneof or maybe object with the given tag whose value is the object resulting from evaluating the expression. For example,

```
ot = oneof[none: null, some: int]
x: ot := ot{none := nil}          % x's object has tag none and value nil
x := ot{some: 7}                  % now x's object has tag some and value 7
x := ot{some: 3.1}                % compile−time error −− expression has wrong type
```

Decomposition of oneof objects is usually done via the **tagcase** statement *(8.11)*.

7.4 Class Constructors

Within a class *(10.4)* and its module *(10.1)*, new objects belonging to the class can be created and initialized using a special constructor for the class. This constructor is similar to the one for records and structs, and also to the specialized constructor in the make statement *(8.17)*. It has the form:

type_designator { **[** *ivar_inits* **]** }

The *type_designator* must name a class type *(10.4)*); this gives the type of the constructed object. All class constructor expressions have a (possibly empty) *ivar_inits* section surrounded by curly braces. This section consists of two parts:

ivar_inits	→	*field_inits* ; *maker_invoc* | *field_inits* | *maker_invoc*
field_inits	→	*field_init* **[** , *field_init* **]***
field_init	→	*idn* := *expr*
maker_invoc	→	*idn* **[** *actual_parms* **]** (**[** *args* **]**)
actual_parms	→	[*type_list*]
type_list	→	*type_designator* **[** , *type_designator* **]***

The *field_inits* part initializes the instance variables of the class; there must be one *field_init* for each instance variable. (If the class has no instance variables, *field_inits* is not used.) The *maker_invoc* part is used to initialize inherited instance variables. This invocation is present *iff* the class named by the *type_designator* has a superclass *(10.5.3)*; *idn* must name a maker *(10.5.2)* provided by *(10.5.1)* the superclass.

The constructor creates a new object of the class type, evaluates the *field_init* expressions (in an arbitrary order), assigns the results to the associated instance variables of the new object, and calls the superclass maker if the *maker_invoc* part is present. If all of these steps terminate normally (i.e., they do not raise an exception), the class construction expression terminates normally with the newly-created object as the result.

For example if a class C has two instance variables, x of type int and y of type char, and does not have a superclass, then

```
n: C := C {x := 3, y := 'A'}
```

creates a new C object with the indicated values in its instance variables and assigns it to n.

7.5 Instance Variable Selection

Within a class *(10.4)* and its module *(10.1)*, the instance variables of objects of the class can be accessed directly using the form

[*expr* . **]** *idn*

where the type of the *expr* is the class type *(10.4)*, and *idn* is the name of an instance variable of that class. If the *expr* is omitted, it defaults to **self** *(10.4)*; i.e., within a method of the class, the instance variables of **self** can be named directly.

Outside a class's module, the instance variables of an object of the class cannot be accessed directly; they can only be accessed indirectly, by invoking methods on the object.

7.6 Field Selection

The form

expr . *idn*

allows fields to be selected from records and structs. The *expr* must evaluate to a record or struct object, and the *idn* must name one of the fields of the object; the result is the object stored in that field.

7.7 Routine Instantiation

Instantiations of parameterized routines can be used as expressions. The form is

idn actual_parms

The *idn* must denote a routine definition *(9.1, 10.2)*. The actual parameters *actual_parms* are the parameters being supplied and legality checking is the same as for type instantiation *(3.2.2)*. The value of such an expression is a routine object.

The type of the resulting routine is derived from the parameterized routine interface *(9.3)* by replacing each occurrence of a formal parameter with the corresponding actual. Thus for

```
p[T] (x: T) returns (T) signals (E)
    where T has lt(T ) returns (bool)
```

the instantiation

```
p[int]
```

is legal and has type

```
proc (int) returns (int) signals (E)
```

Instantiation of parameterized methods is discussed in Section 7.10.

7.8 Procedure Invocation

An invocation *(6.1)* of a procedure with exactly one result can be used as an expression. The type of the expression is the result type of the called procedure.

7.9 Binding

A routine can have some of its arguments *bound*, producing another routine that expects fewer arguments. Binding is essentially an incomplete invocation *(6.1)*: the binding arguments are treated like actuals, and matched up with formals just as in an invocation. An asterisk (*) is used as a placeholder; it indicates an argument position for which no binding is made.

The form for binding is:

bind (*expr bind_args*)

where

bind_args → [, *bind_arg*]* [, *varying_args*]

bind_arg → *expr* | *

The expressions are evaluated in an unspecified order. The result of the first expression must be a routine. The results of evaluating the *bind_arg* expressions are treated just as in an invocation *(6.1)*, except that when the *bind_arg* is an asterisk, no assignment is made to the corresponding formal. The binding is legal if the right number of arguments is provided and the assignments are legal. The result of the **bind** expression is a new routine. The type of that routine has arguments only for the formals where an asterisk appeared. The new routine is the same kind (procedure or iterator) as the original routine, and has the same number and types of results (yielded results) and exceptions as the original.

For example, consider a procedure

```
p (x: int, y: char) returns (bool)
```

Then

```
q: proc (int) returns (bool) := bind(p, *, 'c')
```

is legal and results in a new procedure q with 'c' bound to the formal y; q is invoked with a single int for the formal x, which was not bound. Thus, invocation q(5) has identical behavior to invocation p(5, 'c'). A bound routine can be bound again, e.g.,

```
r: proc ( ) returns (bool) := bind(q, 5)
```

produces a new procedure r, where invocation r( ) has identical behavior to invocation p(5, 'c').

Note that if *varying_arg*s are provided, the corresponding formal is bound to the newly-constructed sequence *(6.1)*. It is not possible to extend the sequence (so that it would contain additional elements) in a subsequent **bind** expression or invocation.

7.10 Method Selection

A method selection selects a method from an object, and produces a routine. The form is:

[*expr* .] *method_idn*

where

method_idn → [^] *idn* [*actual_parms*]

The *expr* denotes the object from which the method is to be selected; it can be omitted within a class to select a method of **self**. The selection is legal if and only if the (apparent) type of the expression has a method named *idn*. The method might be parameterized; in this case the *actual_parms* must allow a successful instantiation *(7.7)*. The optional ^ is used only within a subclass definition *(10.5)* to name an overridden method that comes from the superclass.

A method selection binds *(7.9)* the object computed by the *expr* into the method, producing a routine object whose type is the same as that of the method. When the routine runs, it will be able to refer to the bound object as **self**. Thus, given the following code fragment:

```
counter = type
   inc(x: int)
end counter

c: counter := ... % some initialization
```

the method selection

```
c.inc
```

binds c into the method, producing a routine object of type **proc** (x: int).

Every method invocation is conceptually a method selection followed by a routine invocation. Implementations are expected to optimize the common case in which the call happens immediately; programmers should expect that the code fragment

```
c.inc(1)
```

will run faster than the code fragment

```
m: proc(x:int) := c.inc
m(1)
```

7.11 Prefix and Infix Operators

Theta allows prefix and infix notation to be used as a shorthand for calls of certain methods. The notation is legal if the corresponding method is a procedure and its call is legal. The following table shows the shorthand form and the equivalent expanded form for each operator.

a + b	a.add(b)	~ a	a.not()
a − b	a.sub(b)	a < b	a.lt(b)
a * b	a.mul(b)	a <= b	a.le(b)
a / b	a.div(b)	a = b	a.equal(b)
a // b	a.mod(b)	a ~= b	~(a.equal(b))
a ** b	a.power(b)	a >= b	a.ge(b)
− a	a.minus()	a > b	a.gt(b)
a || b	a.concat(b)		

This notation is used extensively for the built-in types, and may be used for user-defined types as well.

USAGE NOTE

When these methods are provided for user-defined types, they ought to be side-effect-free, and they should mean roughly the same thing as they do for the built-in types. For example, the comparison methods should only be used for types that have a natural partial or total order.

7.12 Fetch

A special form is provided for fetching the element of an array, vector, or sequence, or an abstract object with a method named "fetch":

expr0 `[` *expr1* `]`

This form is just a shorthand for an invocation of a *fetch* method and is equivalent to

expr0 `. fetch (` *expr1* `)`

The expression is legal whenever the corresponding invocation is legal. In other words, the type of *expr0* must define a procedure method named fetch with a single argument whose type is a supertype of *expr1*. For example, if a is an array of integers, a[27] is equivalent to the invocation a.fetch(27).

USAGE NOTE

The use of fetch *for user-defined types should be restricted to types with array-like behavior. Objects of such types will contain an indexed collection of objects. For example, it might make sense for an associative map type to provide a* fetch *method to access the value associated with a string key. A* fetch *method ought not to have side effects.*

Array-like types may also provide a store *operation (8.2).*

7.13 & and |

Two special "short-circuit" binary Boolean operators are provided, & and |.

$expr_1$ & $expr_2$

is the boolean *and* of $expr_1$ and $expr_2$ except that $expr_1$ is guaranteed to be evaluated before any part of $expr_2$. If $expr_1$ is false, $expr_2$ is not evaluated. Similarly, | is the same as a boolean *or* except that $expr_2$ is not evaluated if $expr_1$ evaluates to true. For both & and |, the two expressions must have type bool and the result is a bool.

Because of the conditional expression evaluation, uses of & and | are not equivalent to any normal invocation.

7.14 Precedence and Associativity

When an expression is not fully parenthesized, the proper nesting of subexpressions may in principle be ambiguous. The following precedence and associativity rules are used to resolve such ambiguity; the table lists the higher-precedence operators before lower-precedence ones:

```
.    [ ]  ( )
~    − (unary minus)
**
//   *    /
||   +    − (binary minus)
<    <=   =    ~=   >=   >
&
|
```

The binary operators are all left associative, except for the exponentiation operator **; the exponentiation operator is right associative. I.e., a+b+c is parsed as (a+b)+c; a**b**c is parsed as a**(b**c); a[10].b is parsed as (a[10]).b; a.b[10] is parsed as (a.b)[10].

7.15 Constant Expressions

Constant expressions are a limited kind of expression that can be used in equates. They are evaluated at compile time to produce objects of built-in, immutable types. They can contain method calls but only to methods belonging to compile-time known, built-in immutable objects; the only other calls are to certain built-in, side-effect-free routines. The following forms are allowed:

- literals *(7.1)*
- constructors for structs, oneofs, and maybe's, provided all fields are assigned constant expressions *(7.3)*;
- calls of the built-in routines that create new sequences *(B.9)*, provided all the arguments are constant expressions.
- equated identifiers *(4.6)*;
- identifiers that name built-in or user-defined stand-alone routines;
- instantiations of built-in or user-defined parameterized, stand-alone routines;
- selection of methods of objects denoted by constant expressions;
- invocation of methods of objects denoted by constant expressions provided the actual arguments are also constant expressions.

Evaluation of a constant expression must terminate normally or there will be a compile-time error.

7.16 Primaries

A primary is a limited kind of expression that can be used in the left hand side of an assignment *(5.2)*, or in an invocation statement *(8.1)* or a store statement *(8.2)*. The syntax of primaries rules out expressions that would be ambiguous in these situations, namely, expressions that begin with a left parenthesis and expressions that use infix and prefix operators at the top level.

Primaries are defined as follows:

```
primary      →  simple_expr
             |  primary . idn
             |  primary . method_idn
             |  simple_invoc
             |  primary [ expr ]

simple_expr  →  literal
             |  idn [ actual_parms ]
```

	|	**self**
	|	*method_idn*
	|	*tagged_type_desig* { *field_inits* }
	|	*type_designator* { **[** *ivar_inits* **]** }
	|	**bind** (*expr bind_args*)
simple_invoc	→	*primary* (**[** *args* **]**)

Here are some examples:

```
(a[x])    % not legal -- starts with (
x + y     % not legal -- infix notation at top level
a[x+y]    % legal (for a: array[int] and x, y: int)
p(x)      % legal (for p: proc(int) and x: int)
```

8 Statements

Theta is a statement-oriented language. There are two kinds of statements: simple statements and control statements. Some of the control statements have one or more code bodies as components. A *body* consists of equates followed by statements:

body → **[** *equate* **]*** **[** *statement* **]***

This leads to a nested statement structure.

8.1 Simple Statements

Simple statements do the actual computing. They consist of declarations *(4.5)*, assignments *(5)*, and invocations *(6.1)*.

An invocation statement invokes a procedure. Its form is

primary (**[** *args* **]**)

(A *primary* is a limited kind of expression *(7.16)*.) The semantics of an invocation statement is the same as an invocation expression *(6.1)* except that the procedure invoked may return any number of results, and any such results are discarded.

8.2 Store Statement

A special statement is provided for updating components of array-like types. The statement resembles assignment *(5)* syntactically, but is really an invocation. It has the form

primary [*expr1*] := *expr2*

(A *primary* is a limited kind of expression *(7.16)*.) This form is merely a shorthand for an invocation of a *store* method and is equivalent to the invocation statement

primary . `store` (*expr1* , *expr2*)

The evaluation of the *primary* and the other expressions takes place in an unspecified order.

The form is legal if the corresponding invocation statement is legal, and therefore it is not restricted to arrays but can be used with user-defined types as well. The object resulting from *primary* must have a procedure method named store that takes two arguments whose types are supertypes of the types of *expr1* and *expr2*.

USAGE NOTE

The use of store *for user-defined types should be restricted to types with array-like behavior, that is, types whose objects contain mutable collections of indexable elements. For example, it might make sense for an associative map type to provide a* store *operation for changing the value associated with a key.*

8.3 Return Statement

The form of the **return** statement is

> **return** [(*expr* [, *expr*]*)]

The **return** statement terminates execution of the containing procedure or iterator. There must be the same number of expressions as there are return result types listed in the routine's header, and their types must be subtypes of the corresponding listed types. (If **return** is used in an iterator, no results can be given; iterators do not have return result types.) The expressions (if any) are evaluated in an unspecified order, and the objects obtained become the results of the procedure.

8.4 Yield Statement

A **yield** statement may occur only in the body of an iterator *(6)*. Its form is

> **yield** (*expr* [, *expr*]*)

It has the effect of suspending operation of the iterator and returning control to the invoking **for** statement *(8.8)*. There must be the same number of expressions as there are yield types listed in the iterator's header, and their types must be subtypes of the corresponding listed types. The values obtained by evaluating the expressions (in an unspecified order) are passed to the **for** statement to be assigned to the corresponding loop identifiers. After the body of the **for** loop has been executed, execution of the iterator is resumed at the statement following the **yield** statement.

8.5 Signal Statement

An exception is raised with a **signal** statement, which has the form

> **signal** *name* [(*expr* [, *expr*]*)]

where

> *name* → *idn*

The execution of a **signal** statement begins with evaluation of the expressions (if any), in an unspecified order, to produce a list of *exception results*. The activation of the routine is then terminated and execution continues in the caller *(8.14)*.

The exception name must be either one of the exception names listed in the routine heading or failure. If the name is failure, there must be exactly one expression present, of type string. If the name is listed in the routine header, there must be the same number of expressions as are listed for that exception in the header, and their types must be subtypes of the corresponding listed types.

8.6 If Statement

The form of the **if** statement is

> **if** *expr* **then** *body* **[** **elseif** *expr* **then** *body* **]*** **[** **else** *body* **]** **end**

The expressions must be of type bool. They are evaluated successively until one is found to be true. The body corresponding to the first true expression is executed, and the execution of the **if** statement then terminates. If none of the expressions is true, the body in the **else** clause is executed (if the **else** clause is present). The **elseif** form provides a convenient way to write a multiway branch.

8.7 While Statement

The **while** statement has the form

> **while** *expr* **do** *body* **end**

Its effect is to execute the body repeatedly as long as the expression remains true. The expression must be of type bool. If the value of the expression is true, the body is executed, and then the entire **while** statement is executed again. When the expression evaluates to false, execution of the **while** statement terminates.

8.8 For Statement

A **for** statement is used to invoke an iterator *(6)*, and is the only way an iterator can be invoked. The iterator produces a sequence of *items* (where an item is a group of one or more objects) one item at a time; the body of the **for** statement is executed for each item in the sequence.

The **for** statement has the form

> **for** *for_idns* **in** *invoc* **do** *body* **end**

where

> *for_idns* → *idn_list* | *decl* **[** , *decl* **]***

The loop variables are given first. Either a list of already-declared variables is given, or new loop variables local to the **for** statement are introduced using declarations. Previously declared variables cannot be mixed with new declarations. The invocation, given next, must be an invocation of an iterator.

The first loop variable is assigned the first object yielded in an item, etc.; the type of each yielded object (according to the specification of the iterator) must be a subtype of the type of the corresponding loop variable.

Execution of the **for** statement proceeds as follows. First the iterator is invoked, and it either yields an item or terminates. If the iterator yields an item, its execution is temporarily suspended, the objects in the item are assigned to the loop variables, and the body of the **for** statement is executed. The next cycle of the loop is begun by resuming execution of the iterator from its point of suspension. Whenever the iterator terminates, the entire **for** statement terminates. If the **for** statement terminates, this also terminates the iterator.

The following example creates an array[int] and appends the numbers 1 through 10 to it:

```
a: array[int] := array_new[int]()
for i: int in 1.to(10) do
   a.append(i)
end
```

This example uses the to iterator method of int object 1 *(B.4)*, which yields successively larger values starting with its object and ending with the argument.

8.9 Break Statement

The **break** statement has the form

> **break**

Its effect is to terminate execution of the nearest **for** or **while** statement that contains the **break**. It is a compile-time error to use **break** outside the body of a **for** or **while** statement.

8.10 Continue Statement

The **continue** statement has the form

> **continue**

Its effect is to terminate execution of the body of the nearest **for** or **while** statement that contains the **continue**, and to start the next cycle of that loop (if any). It is a compile-time error to use **continue** outside the body of a **for** or **while** statement.

8.11 Tagcase Statement

The **tagcase** statement is a special statement provided for decomposing oneof *(B.13)* and maybe *(B.14)* objects; it permits the selection of a body to be performed based on the tag of the object. Its form is

> **tagcase** *expr tag_arm tag_arm** **[** **others** : *body* **]** **end**

where

> *tag_arm* → **when** *name* **[** , *name* **]*** **[** (*idn* : *type_designator*) **]** : *body*

The expression must evaluate to a oneof or maybe object. The tag of this object is then matched against the names on the *tag_arm*s. When a match is found, if a declaration exists in the arm, the value component of the object is assigned to the local variable *idn*. The matching *body* is then executed; the *idn* is defined only in that body. If no match is found, the *body* in the **others** arm is executed. When execution of the body completes, control continues at the statement after the **tagcase**.

In a syntactically correct **tagcase** statement, the following constraints are satisfied:

1. The type of the expression is some oneof or maybe type T.

2. The tags named in the *tag_arm*s are a subset of the tags of T, and no tag occurs more than once.

3. If all tags of T are present, there is no **others** arm; otherwise an **others** arm must be present.

4. On any *tag_arm* containing a declaration, the *type_designator* must denote a supertype of the type that corresponds to each tag named in the *tag_arm*.

Here is an example:

```
x: oneof[none: null, some: int]
...
tagcase x
   when none: ...
   when some(y: int): ... y+7 ...
   end
```

8.12 Typecase Statement

A variable or expression in Theta has an apparent type known to the compiler, and the compiler guarantees that the actual type of the object denoted by the variable or computed by the expression is a subtype of that type. Sometimes it is useful to determine what the actual type of the object is, or to narrow its apparent type to some subtype. This is accomplished by the **typecase** statement. The form of this statement is

typecase *expr type_arm* [*type_arm*]* [**others** : *body*] **end**

where

type_arm → **when** *type_designator* [(*idn*)] : *body*

The expression *expr* is evaluated, and the actual type A of the resulting object is then used to select a *type_arm*. The *type_arm*s are considered in order, and the first one whose type designator denotes a supertype of type A is selected: the object is assigned to the *idn* of that *type_arm* (if present), and the corresponding *body* is executed. Within this body the *idn* can be used to refer to the object with the type specified in the arm; the *idn* is defined only in this body. When execution of the body completes, control continues at the statement after the **typecase**. An **others** arm always matches, but provides no useful additional information about the object's type.

A legal **typecase** statement satisfies the following constraints:

1. No type occurs in more than one arm.

2. The type of each arm must be a *proper subtype* of the *apparent type* of expression *expr*. A proper subtype of a type includes all subtypes except for the type itself, so that it is always narrower than the type itself.

3. If S is a subtype of T and both S and T are used in type arms, the type arm for S must precede the type arm for T. (The more specific type must precede the more general type since otherwise the arm with the more specific type is useless.)

The following example assumes set and bag are both subtypes of type collection, and stack is a subtype of bag:

```
x: collection
...
typecase x
   when stack(y):  ...% y is a stack in this arm
   when bag(z):    ...% z is a bag in this arm
   others:         ...% x must be used as a collection in this arm
   end
```

The **others** arm will be selected if x's actual type is set (or some other collection type that is not a subtype of stack or bag).

8.13 Begin Statement

The **begin** statement permits a sequence of statements to be grouped together into a single statement. Its form is

begin *body* **end**

Since control statements already have bodies that group statements, the main use of the **begin** statement is to group statements together for use with the **except** statement *(8.14)*.

8.14 Except Statement

By attaching handlers to a statement, the caller can specify the action to be taken when an exception is signaled by an invocation contained within that statement. (**Except** statements also handle signals raised by contained **exit** statements *(8.16)*.) A statement with handlers attached is called an **except** statement and has the form

statement **except** [*handler*]* [**others** [(*idn* : **string**)] : *body*] **end**

where

handler → **when** *name* [, *name*]* [(*decl* [, *decl*]*)] : *body*

Let S be the statement to which the handlers are attached, and let X be the entire **except** statement. Each *handler* arm specifies one or more exception names and a body. The body is executed if an exception with one of those names is signaled by an invocation in S. All the names listed in the arms must be distinct. The optional **others** arm is used to handle all exceptions not explicitly named in a *handler* arm. S can be any form of statement, even another **except** statement.

If, during the execution of S, some invocation in S signals an exception E, control immediately transfers to the closest applicable handler: that is, the closest handler for E that is attached to a statement containing the invocation. When execution of the handler is complete, control passes to the statement following the one to which the handler is attached. Thus if the closest handler is attached to S, the statement following S is executed next. If execution of S completes without signaling an exception, the attached handlers are not executed.

An exception raised by an invocation inside a handler is treated the same as any other exception: Control passes to the closest handler attached to a statement containing the invocation. Note that if handlers H1 and H2 are attached to the same statement S, and H1 makes an invocation that raises an exception, H2 cannot handle this exception since H2 is not attached to a statement that encloses H1. Either H1 must handle the exception itself, or a handler attached to a statement that contains H1 must handle the exception.

8.14.1 Handlers without Declarations

If a handler for exception E has no declarations, then if it is selected to handle E, any results raised with E are simply ignored. Such a handler can be used to handle exceptions that are raised without results, or in cases where the results are not of interest.

8.14.2 Handlers with Declarations

A handler with declarations is used to handle exceptions with the given names when the exception results are of interest. The declared variables, which are local to the handler, are assigned the exception results before the body is executed.

When matching exceptions to handlers, only the exception names are used. If a handler for exception E includes declarations, it must be able to handle all possible exceptional results that can be raised with E. For example, suppose in the same statement there are two routine invocations, where one has **signals**(foo(int)) in its signature, and the other has **signals** (foo(string,real)). In this case it is impossible to use a handler with declarations to handle exception foo; no single handler with declarations could cover both cases. (The solution is to use a handler without declarations, or to split the statement up into two different **except** statements, so that different handlers can be used.)

Stated more formally: If a handler H for exception E has K variable declarations, and is attached to statement S, then for each invocation in S that can raise E so that it is handled by H, E must be raised with K results and the ith result type must be a subtype of the ith variable declared by H; otherwise there is a compile-time error.

8.14.3 Others Handler

The **others** arm is optional and must appear last in a handler list. This form handles any exception not handled by other handlers in the list. If a variable is declared, it must be of type string. The variable, which is local to the handler, is assigned a lower-case string representing the actual exception name; any results of the exception are discarded.

8.14.4 Example

The following example assumes that procedure p raises exceptions e1, e2, and e3, and that e1 and e2 have no results, while e3 has an int result. It also assumes that procedure q raises e1 (with no results) and e4 (also with no results).

```
begin
   p(q( ))  except  when e1: % handle e1
                    when e3(x:int): % handle e3
                    end
end except  when others: % handle e2 and e4 here
            end
```

The first arm handles exception e1, which might have been raised by either the call of q or the call of p. Exceptions e2 and e4 aren't handled by the inner except statement, but are handled by the outer one.

8.14.5 The Failure Exception

The exception failure(string) is implicitly included in every routine interface; it is illegal to list a failure exception explicitly in an interface.

If a routine performs an invocation that raises an exception not handled by any **except** statements in the routine body, the routine will terminate automatically with the failure exception. If the unhandled exception is not failure, the failure exception will have as a result a string naming the unhandled exception; otherwise its result will be the string argument of the unhandled failure exception. The semantics is the same as if every routine body were placed in an **except** statement of this form:

```
begin
   % routine_body
end
 except  when failure (s: string): signal failure(s)
         others (s: string): signal failure ("unhandled exception: " || s)
 end
```

8.15 Resignal Statement

A **resignal** statement is a syntactically abbreviated form of exception handling:

statement **resignal** *name* **[** , *name* **]***

Each name listed must be distinct, and each must be either one of the condition names listed in the routine heading or failure. The **resignal** statement acts like an except statement containing a handler for each condition named, where each handler simply signals that exception with exactly the same results. Thus if the resignal clause names an exception with a specification in the routine heading of the form name(T1, ..., Tn), effectively there is a handler of the form

```
when name (x1: T1, ..., xn: Tn): signal name(x1, ..., xn)
```

The compiler checks all exceptions that can be raised by the statement that the **resignal** is attached to against these implicit handlers. As discussed above, an error is reported if an exception can be raised with the wrong number of arguments, or if the ith result type is not a subtype of the type of the ith variable declared in the handler.

8.16 Exit Statement

The **exit** statement provides a **signal**-like transfer of control to a local handler, without terminating the current invocation.

The **exit** statement has the form

exit *name* **[** (*expr* **[** , *expr* **]***) **]**

An **exit** statement raises a local exception that must be handled explicitly by a **when** arm of a containing **except** statement; the compiler reports an error if an **exit** is not handled, or is handled by a **resignal** or **others** handler. Furthermore, the handler must handle any results explicitly: if there are K results, the *decl*s in the matching arm must declare K variables, with types that are supertypes of the associated expressions in the **exit** statement.

8.17 Make Statement

The **make** statement is used to initialize a newly created object. It may be used only within a maker *(10.5.2)*; the maker's routine interface specifies the class of the object to be initialized. Makers use **make** and not **return** for termination: if the **make** statement terminates normally, this causes the maker to terminate normally.

The **make** statement has the form

make { **[** *ivar_inits* **]** } **[** **then** *body* **end** **]**

All **make** statements have a (possibly empty) *ivar_inits* section surrounded by curly braces. This section (which is identical to the class constructor expression *(7.4)*) consists of two parts:

$$
\begin{array}{rcl}
\textit{ivar\_inits} & \rightarrow & \textit{field\_inits} \ ; \ \textit{maker\_invoc} \mid \textit{field\_inits} \mid \textit{maker\_invoc} \\
\textit{field\_inits} & \rightarrow & \textit{field\_init} \ \mathbf{[} \ , \ \textit{field\_init} \ \mathbf{]}* \\
\textit{field\_init} & \rightarrow & \textit{idn} := \textit{expr} \\
\textit{maker\_invoc} & \rightarrow & \textit{idn} \ \mathbf{[} \ \textit{actual\_parms} \ \mathbf{]} \ (\ \mathbf{[} \ \textit{args} \ \mathbf{]} \ )
\end{array}
$$

The *field_inits* part initializes the instance variables of the maker's class *(10.4)*; there must be one *field_init* for each instance variable. (If the class has no instance variables, *field_inits* is not used.) The *maker_invoc* part is used to initialize inherited instance variables. This invocation is present *iff* the maker's class has a superclass *(10.5.3)*; *idn* must name a maker *(10.5.2)* provided by the superclass *(10.5.1)*.

The **make** statement has an optional **then** *body* **end** clause that can be used to do additional work after the instance variables have been assigned initial values. Within this clause, the newly created object can be referred to using the variable self, and the instance variables of the newly-created object can be used directly as variables, as in a method body *(10.4)*.

A **make** statement is evaluated in three steps; later steps are performed only if earlier steps complete normally (i.e., they do not raise an exception). The steps are:

1. All the *expr*s in the *field_init*s are performed in an unspecified order and the resulting objects are assigned to the associated instance variables of the new object.

2. If present, the invocation *maker_invoc* is performed.

3. If present, the *body* is evaluated.

If all three steps complete normally, the **make** statement terminates and causes the containing maker to terminate normally.

A **make** statement cannot appear in the body of another **make** statement.

9 Specifications

New types and routines are introduced by giving specifications. Specifications can be parameterized; we describe non-parameterized specifications first, and then discuss parameterization in Section 9.3.

9.1 Stand-Alone Routine Specifications

A stand-alone routine specification defines the interface of a stand-alone procedure or iterator. Both procedures and iterators can have zero or more arguments, and can terminate either normally or in some named exception condition; different numbers and types of results can be returned in the different cases. An iterator can yield one or more intermediate results and must have no results in the case of normal termination. Here are examples:

```
% procedures:
search (a: array[int], x: int) returns (int) signals (not_found)
combine (x: sequence[int]) returns (int)
% an iterator:
elements (a: array[int]) yields (int)
```

Routine specifications have the following form:

routine_interface → *proc_interface* | *iter_interface*

proc_interface → *idn* [*parms*] *formal_args* [*returns*] [*signals*] [*where*]

iter_interface → *idn* [*parms*] *formal_args* *yields* [*signals*] [*where*]

The *parms* and *where* appear only if the routine is parameterized; we defer discussion of these forms to Section 9.3. The *formal_args* defines the formal arguments of the routine:

formal_args → ([*decl* [, *decl*]*])

The returns clause lists the types of the results of a procedure:

returns → **returns** (*type_designator* [, *type_designator*]*)

There can be zero or more results, and the returns clause is omitted if there are no results. The yields clause lists the types of yielded items for an iterator:

yields → **yields** (*type_designator* [, *type_designator*]*)

An iterator must always have a yields clause, and the yielded items must always contain at least one result. The signals clause lists the names and result types for the exceptions of the routine:

signals → **signals** (*exception* [, *exception*]*)

where

exception → *name* [(*type_designator* [, *type_designator*]*)]

name → *idn*

Each exception name must be distinct and none can be failure; in addition to the explicitly listed exceptions, every routine can raise the failure exception, with a single string result.

The type of a routine is derived from its *routine_interface* in a straightforward way: all the information in the interface is significant except for the routine name and the names of the formal arguments. For example, the type of the combine procedure shown above is proc (sequence[int]) **returns** (int), the type of the search procedure is **proc** (array[int], int) **returns** (int) **signals** (not_found), and the type of the elements iterator is **iter** (array[int]) **yields** (int). The failure exception does not appear in these types; it is suppressed because every routine has this exception.

The various routine types form a type hierarchy *(3.4.1)*.

A routine whose last argument is a sequence can be called using a varying number of arguments in that position *(6.1)*.

9.2 Type Specifications

A type specification defines the interface of a type. It has the form

type_interface → *idn* = **type** [*parms*] [*supertypes*] [*where*]
  [*interface_or_equate*]*
  end *idn*

where

interface_or_equate → *routine_interface* | *equate*

The initial *idn* is the identifier of the new type; this identifier will be used throughout the specification to refer to the new type. The final *idn* must be the same as the initial *idn*. The *parms* and *where* clauses are present only if the type is parameterized *(9.3)*. In this section we consider only non-parameterized types.

The *supertypes* clause lists all supertypes of the new type except for any, which is the supertype of all types and is never listed. The form is

supertypes → < *super_info* [, *super_info*]*

super_info → *type_designator* [{ *renames* [, *renames*]* }]

renames → *idn* **for** *idn*

The *type_designator* in a *super_info* clause identifies a type that is an immediate supertype of the new type. The optional *renames* clauses allow methods of that supertype to be renamed; the supertype method with the second name will be renamed with the first name in the subtype. For example,

stack = **type** < bag {push **for** put, pop **for** get}

states that stack is a subtype of bag, except that the bag method named put will be renamed push within stack and the bag method named get will be renamed pop within stack.

The remainder of the definition consists of *equates* and interfaces of the methods of objects of the type. Each method is defined by a *routine_interface (9.1)*, and the type of the method is derived from this interface in the usual way. Each method must have a distinct name. Methods can be procedures or iterators; they can have parameters and can contain constraints *(9.3)*. A *routine_interface* must be provided for a method when (1) there is no corresponding supertype method or (2) the subtype method has a different signature from that of the corresponding supertype method. The new names introduced by the renamings (if any) are used in these interfaces.

USAGE NOTE

An explicit routine_interface *should be provided for a method whose behavior differs from that of the corresponding supertype method, even if the types of the two methods are the same, since this will allow the specification of the subtype method to be given.*

Names corresponding to operators (7.11) should be used when this makes sense. For example, if the new type has an addition method, naming the method add *will allow the + operator to be used when it is called.*

The methods and their signatures determine the legality of the subtype declarations. A subtype must have all the methods of its supertype and the type of the subtype method must be a subtype of the type of the corresponding supertype method *(3.4.1)*. Compilation of a type specification will fail if these conditions are not satisfied for every declared supertype.

USAGE NOTE

The legality constraint ensures that when a call is made to a method based on information about the apparent type of its object, the call will be legal even if the actual type of the object is a subtype of this type. For the call to make sense, the definer of a subtype should ensure that objects of the subtype behave similarly to those of the supertype. This means roughly that corresponding methods do the same thing, except that the subtype methods might do something extra to those parts of the subtype object state that aren't visible through the supertype methods. Also, subtype methods that do not correspond to supertype methods should not do anything to the part of the subtype object state that is visible via the supertype methods that cannot be accomplished using supertype methods. A discussion of the meaning of the subtype relation can be found in [3].

Renamings should be avoided whenever possible. They are needed, however, when a type has multiple supertypes, and these supertypes have methods of the same name but different behavior, or methods of different names and the same behavior.

A type specification provides no way to create objects of the type "from scratch." Instead, new objects are created by calls on stand-alone routines.

USAGE NOTE

Names for creation routines ought to indicate that they create objects of a given type T, *e.g., by using names of the form* make_T, create_T, *or* new_T.

Some examples of type specifications are given in Section 9.4.

9.3 Parameterized Specifications

Routine and type specifications can be parameterized by types, and so can specifications of methods, even those contained within parameterized type specifications. In all cases, legal instantiations can be constrained by **where** clauses.

parms	→	[*idn_list*]
where	→	**where** *restriction* **[** , *restriction* **]***
restriction	→	*idn* **has** *nonparam_interface* **[** , *nonparam_interface* **]***
nonparam_interface	→	*idn nonparam_proc_sig* | *idn nonparam_iter_sig*
nonparam_proc_sig	→	(**[** *type_list* **]**) **[** *returns* **]** **[** *signals* **]**
nonparam_iter_sig	→	(**[** *type_list* **]**) *yields* **[** *signals* **]**
type_list	→	*type_designator* **[** , *type_designator* **]***

All the parameters are types. The **where** clause lists any restrictions on a parameter by stating methods that objects of the parameter type must have. For example,

```
find [T] (a: collection[T], v: T) returns (int) signals (not_found)
    where T has equal (T) returns (bool)
```

defines a parameterized routine with one parameter T. Within the parameterized definition, objects of a parameter type can be assumed to have the methods listed in the **where** clause; any instantiation of the parameterized definition will provide a binding for each of these methods *(3.2.2, 7.7)*.

Within the scope corresponding to a parameterized specification, parameter names can be used to denote types. If parameters are used in instantiations, any requirements of the parameterized types being instantiated must be met in the usual way. For example, consider the instantiation collection[T] used in the specification of find; if collection required an lt method, the Theta compiler would reject the specification of find because the instantiation of collection is illegal.

A parameterized type may have declared supertypes. For example

```
stack = type[T] < item, bag[T], stack_pair[T, int]
```

states that every instantiation of stack is a subtype of item and of the corresponding instantiations of bag and stack_pair (e.g., stack[foo] is a subtype of item, bag[foo], and stack_pair[foo, int]). There is no subtype/supertype relationship on parameters: for example, stack[foo] is not a subtype of stack[bar] even if foo is a declared subtype of bar. Subtyping operates only "outside the square brackets" of a type definition.

9.4 Type Specification Examples

First we specify a simple bag abstraction and two creation routines for bags:

```
bag = type [T]

    put (x: T)
        % effects  adds x to the bag
```

```
        get ( ) returns (T) signals (empty)
            % effects  removes and returns an arbitrary element of the bag
            %     signals empty if bag is empty

        size ( ) returns (int)
            % effects  returns the number of elements in the bag

        copy ( ) returns (bag[T])
          where T has copy( ) returns (T)
            % effects  returns a new bag containing copies of the elements of self

    end bag

    create_bag [T] ( ) returns (bag[T])
        % effects  returns a new, empty bag

    singleton_bag [T] (x: T) returns (bag[T])
        % effects   returns a new bag containing x as its only element
```

Note in the specification of copy the use of self to refer to the method's object. Note also that the copy method imposes a constraint on T whereas the bag type has no constraint on T; copy is an optional method *(3.2.2)*.

Next we specify type stack, a subtype of bag:

```
    stack = type [T] < bag[T] {push for put, pop for get}

        push (x: T)
            % effects  adds x to the top of the stack

        pop ( ) returns (T) signals (empty)
            % effects  removes and returns the top element of the stack
            %     signals empty if stack is empty

        top ( ) returns (T) signals (empty)
            % effects  returns top element of the stack
            %     signals empty if stack is empty

        copy ( ) returns (stack[T])
          where T has copy ( ) returns (T)
            % effects  returns a new stack containing copies of the elements of self
            %     in the same order as in self.

    end stack

    create_stack [T] ( ) returns (stack[T])
        % effects  returns a new, empty stack
```

The specification of stack introduces a new method, top. It contains specifications for push and pop, since they constrain the behavior of the corresponding bag methods, and for copy, since it has a different signature and specification than its counterpart, but it omits the specification of size, since it would be identical to its counterpart. Note that there are two creation routines for bag but only one for stack.

10 Implementations

Implementations are provided by modules. A module contains a set of classes *(10.4)*, routine implementations, maker definitions, and equates. It can export its routine implementations so that they can be used in other modules. Its classes may also be available for use in other classes (as superclasses) but this is indicated directly by the class definition *(10.5.1)*.

10.1 Modules

The form of a module is:

module → **module** [*implements*] [*impl_elt*]* **end**

impl_elt → *routine_def* | *class_def* | *maker_def* | *equate*

Every module must contain at least one class or routine implementation. The **implements** clause identifies the routine specifications that are being implemented by what module and defines which routine implementations in the module implement each specification:

implements → **implements** *exported_item* [, *exported_item*]*

exported_item → *idn* [{ *idn_list* }] | *idn* `[` *parms* `]` { *idn_list* }

The first kind of *exported_item* identifies a *routine_interface (9.1)* that is being implemented within the module. The optional *idn_list* names the *routine_def*s that implement that specification; the list can be omitted if there is a single implementation with the same name as that of the routine interface that is implemented. The second kind of *exported_item* indicates that the module implements a specific instantiation of a parameterized *routine_interface (9.3)*; in this case an *idn_list* naming one or more *routine_def*s must be provided. The module must contain all the *routine_def*s indicated in the **implements** clause, and the type of a *routine_def* must be a subtype of the type of the *routine_interface* or instantiation that it implements. The module can contain other *routine_def*s, but these can be used only within the module (unless a class *provides* one of these routines to its subclasses 10.5.1).

For example, a module that implements the bag type defined in Section 9.4 might say:

module implements create_bag

to indicate that it is exporting an implementation of the create_bag routine, and furthermore the *routine_def* within the module is also named create_bag. Users can call this routine (after instantiating it) to obtain bags that are implemented with whatever class is indicated in that create_bag implementation. Alternatively, a module might provide a specialized implementation for bag[char], e.g.,

module implements create_bag[char]{cbag}

indicating that internal routine cbag will implement the instantiation create_bag[char]. Users can make use of cbag to create a new bag[char] with the specialized implementation.

Using code does not use the routine names exported by modules. Instead the code uses names introduced in routine specifications. Before the code runs, a *linker* binds every use of a routine specification to a routine that implements that specification. For example, in the following code:

```
b: bag[char] := create_bag[char]( )
```

create_bag[char] could be linked to cbag.

A module defines a scope *(4)*. Modules do not nest: no module is ever contained within another module.

Modules provide encapsulation. Details of a class are visible throughout its module so that, for example, a routine in the module can access the instance variables of objects of the class. The internal details of a class are never visible to code outside its module.

USAGE NOTE

Usually a class will be part of a module that also implements routines that create new objects of the class. If the class's module does not export such routines, there will be no way for users to create its objects "from scratch." Such a class may still be useful as a superclass (10.5), making definitions of other classes easier to write.

10.2 Stand-Alone Routine Implementations

The implementation of a stand-alone routine has the form:

routine_def → *routine_interface body* **end** *idn*

routine_interface → *proc_interface* | *iter_interface*

The final *idn* in the *routine_def* must match the routine name introduced by the name of the routine.

A routine implementation is a scope in which the *idn*s that name the formal parameters and formal arguments are defined. These names can be used in the routine body to refer to the respective parameters and arguments. Theta uses call-by-sharing *(6.2)* to pass arguments to a routine. Accordingly, making an assignment to a formal argument does not affect the caller; it only changes what object is denoted by that formal. The only way a procedure or iterator can communicate with its caller is by returning results (yielding results for an iterator), signaling exceptions, or modifying mutable argument objects.

If control reaches the end of the body of a procedure that has results, the procedure will terminate with the exception failure("no return results"). Iterators, and procedures without return results, can terminate normally by reaching the end of their bodies.

10.3 Parameterized Implementations

Routine and method implementations, as well as classes, can be parameterized. Within the scope of a parameterized implementation, formal parameters can be used as types whose objects have only the methods and signatures indicated in the **where** clause. For example, in

```
find [T] (x: collection[T], v: T) returns (int) signals (not_found)
    where T has equal(T) returns (bool)
  w: T
  ...
  if v = w ...  % short for v.equal(w)
  ...
  end find
```

the call v.equal(w) (made using the associated short form) is legal. Such method calls make sense because instantiations are only permitted for types whose objects have the required methods *(3.2.2)*.

10.4 Classes

Classes implement user-defined types and user-defined parameterized types. A class has the following form:

class_def → *idn* = **class** [*parms*] [*for_type*] [*inherits*]
[*where*] [*provides*] [*hides*]
[*equate_or_ivar_decl*]*
[*equate_or_routine_def*]*
end *idn*

where

equate_or_ivar_decl → *equate* | *ivar_decl*

equate_or_routine_def → *equate* | *routine_def*

The initial *idn* names the class, and the final *idn* must match it. The *inherits*, *provides*, and *hides* clauses are part of the inheritance mechanism *(10.5)*.

The *for_type* clause names the type implemented by the class:

for_type → **for** *type_designator*

If this clause is missing, the class does not implement a type. The primary use of such a class is as a superclass to classes defined in other modules *(10.5.1)*; the class could also be used privately, within its own module.

A class is a scope in which the *idn*s that name the formal parameters are defined. Therefore these names can be used within the class to refer to the respective parameters. The **where** clause of the class lists the constraints on the parameters.

The body of a class has two main parts: a set of declarations and a set of method implementations. All the names in these two sets must be distinct.

The declarations define the *instance variables*, which are used to represent the state of objects of the class. Every object of the class has its own instance variables; the values of its instance variables define the state of the object. An instance variable is declared using a special form:

ivar_decl → *decl* | *decl_with_impls*

The *decl_with_impls* form is used to provide abbreviated implementations *(10.4.1)* of some methods.

Following the instance variable declarations are implementations of the methods. A class must implement all methods of its type (although some of these implementations can be inherited *(10.5)* or abbreviated *(10.4.1)*); these are its "public" methods. A class that does not implement a type has no public methods. The signatures of *routine_def*s that implement public methods must have types that are subtypes of those given in the type specification. In addition a class can implement some "private" methods that can be used only within the class and its module and possibly within subclasses of the class *(10.5)*.

A class name (or the name plus the actual parameters if the class is parameterized *(10.3)*) can be used as a type within the class or its module. We refer to this type as the *class type*. The class type describes the set of objects implemented by the class. Objects of this type have a field corresponding to each of the instance variables and all the public and private methods defined by the class. The "dot" notation is used to access the object's instance variables and select it methods, e.g.,

```
% assume class C has instance variable v and method m
x: C           % x is an object of class type C
x.m(...)       % selects m from x and calls it
x.v := ...     % assigns to x's instance variable v
y := ...x.v... % gets value of x's instance variable v
```

In addition to its regular arguments, a method has an implicit argument that it can refer to by using the keyword **self**. This argument refers to the method's object. The method can use **self** to access the instance variables and select the methods belonging to its object, e.g., **self**.v, or it can access instance variables and select methods of its object without using the "dot" notation, just by using their names, e.g., v means the same thing as **self**.v.

If the class implements a type, its class type is a subtype of this type. For example, within a class C or its module, the statement

```
typecase x          % assume x: T
   when C (y):      % in here y refers to x as a C
end
```

allows the use the form y.v to access y's instance variable v within the **when** arm. The arm will be selected if the actual object is a C or some subclass of C *(10.5)*.

Within a class and its module, new objects of the class type can be obtained by calling the class constructor *(7.4)*.

10.4.1 Abbreviated Implementations

Sometimes methods merely provide access to instance variables, either to *get* the value of the instance variable, or (more rarely) to *set* the value. Methods like these can be implemented using a short form by annotating the instance variable:

decl_with_impls → *idn1* : *type_designator* **implements** *idn2* **[** , *idn3* **]**

Here, *idn1* is the name of a variable being declared, *idn2* is the name of a method to get the value of the variable, and *idn3* (if present) is the name of a method to set the value of the variable. A public get method must be a procedure that takes no arguments and returns a T where T is a supertype of the type of the associated instance variable; a public set method must be a procedure that has no return values and takes one argument of type S, where S is a subtype of the type of the associated instance variable. No restrictions are placed on the exceptions listed in the type of the get or the set method. For example, it is okay to provide abbreviated implementations for the following methods:

```
size () returns (int) signals (unknown)
set_name(name: string) signals (permission_denied)
```

An abbreviated implementation for a method is permitted only if the class provides no other implementation for the method. Here is an example of the use of abbreviated implementations:

```
point = type
   x( ) returns (int)   % returns value of x coordinate
   set_x(v: int)        % changes value of x coordinate to v
   y( ) returns (int)   % returns value of y coordinate
   set_y(v: int)        % changes value of y coordinate to v
end point

c = class for point
   x_coord: int implements x, set_x
   y_coord: int implements y, set_y
end c
```

USAGE NOTE

Related names should be used for the get and set methods, and the meaning of the methods should be getting and setting abstract fields of the object.

10.4.2 Same_object

Within a class a special procedure is available for determining whether two objects of that class are actually the very same object. This procedure has the *routine_interface*:

```
same_object (x: C, y: any) returns(bool)
```

where C is the name of the class. The procedure returns true if x and y denote the same object and false otherwise. For example, the equal method for a mutable type might call **same_object(self**, z) to determine whether object z is the same object as **self**.

The **same_object** procedure is available only within the class. It cannot be used by other code in the class's module, and it cannot be exported by the class.

10.4.3 Example

Here is an implementation of the bag type whose specification was given in Section 9.4.

```
module implements create_bag

brep = class[T] for bag[T]

    sz: int implements size        % implementation of size method
    els: array[T]

    % the rep invariant is: sz = els.size( )

    put (x: T)
        els.append(x)
        sz := sz + 1
        end put
```

```
        get ( ) returns (T) signals (empty)
            x: T := els.remove( ) except when limits: signal empty end
            sz := sz − 1
            return (x)
            end get

        copy ( ) returns (bag[T])
            where T has copy ( ) returns (T)
            return (brep[T]{sz := sz, els := els.copy( )})
            end copy

        end brep

    create_bag [T] ( ) returns (bag[T])
        return (brep[T]{sz := 0, els := array_create[T]( )})
        end create_bag

    end
```

This module exports an implementation of the create_bag routine, which enables users to obtain **bag** objects implemented by the brep class. Note the use of the class constructor *(7.4)* in the copy method and create_bag to obtain a new bag object.

10.5 Inheritance

Inheritance allows a class, called the *subclass*, to be implemented as an extension of some other class, called the *superclass*.

10.5.1 Defining Superclasses

A class definition must indicate explicitly that it is available for subclassing by explicitly *providing* an interface to its subclasses:

provides → **provides** *idn_list*

hides → **hides** *idn_list*

The **provides** clause lists private methods, routines, and makers *(10.5.2)* implemented in the class's module that are available to subclasses; it also lists any public methods where the signature given in the class differs from that given in the type *and* the class definer wishes this information to be visible to definers of subclasses. Some makers must be provided, since otherwise subclasses will have no way to create new objects of their own; if no maker for the class is listed, there will be a compile-time error. The **hides** clause lists public methods that are not available to subclasses.

USAGE NOTE

A superclass must permit subclasses to be implemented efficiently. It must export enough makers that a subclass can conveniently initialize the superclass fields of its new objects. Also, it must export enough methods so that a subclass can access needed information in the superclass fields of its objects, and modify those fields if that makes sense.

The class must also ensure that subclasses cannot interfere with it; this requirement is discussed further in Section 10.5.4.

10.5.2 Makers

A maker is a special operator that fills in the fields of a newly created object:

maker_def → *maker_interface body* **end** *idn*

maker_interface → *idn* [*parms*] *formal_args makes* [*signals*] [*where*]

makes → **makes** (*type_designator*)

The final *idn* in the *maker_def* must match that given in the *maker_interface*. The *type_designator* in the **makes** clause must denote a class type implemented in the maker's module; we say this is the maker's class.

The object being initialized by a maker is created by the Theta runtime before the maker is called; its class is some subclass of the maker's class, and the new object is an implicit argument of the maker. The maker fills in its fields using a make statement; the new object can be referred to explicitly by the name self, but only within the optional *body* of the make statement *(8.17)*. The usual scoping rules for self apply: the instance variables of self can also be directly named as variables within the *body* of the make *(7.5)*.

A maker cannot contain a **return** statement. Normal termination of a **make** statement causes its execution to terminate normally. If the maker reaches the end of its body, it will terminate with the exception failure("no return results").

A maker can be called only within a **make** statement *(8.17)* or a class constructor *(7.4)* within some subclass of the maker's class; the maker cannot be called within its own class. It is used in the subclass to initialize the instance variables inherited from a superclass, and thus its use is limited to modules implementing subclasses of its class. It must be listed in the **provides** list of its class.

USAGE NOTE

The purpose of a maker is to initialize the new object properly so that its fields that belong to the maker's class satisfy the rep invariant for that class. This condition should be satisfied both when the maker calls a maker of its superclass, and also when the maker returns, e.g.,

```
m ( ) makes (C)
   ...   % do some preprocessing
   make { ...;  % initialize class fields so that class rep I holds
          sm ( ) } % call superclass maker; at return all rep I's hold
     then ...        % update class fields so that class rep I holds
     end  % all rep I's hold now
   end m
```

10.5.3 Subclasses

To inherit code from a superclass, a class includes the **inherits** clause in its *routine_interface*:

inherits → **inherits** *type_designator* [{ *renames* [, *renames*]* }]

renames → *idn* **for** *idn*

The *type_designator* names the superclass. The **inherits** clause makes the superclass name and the names of provided methods and routines visible to code in the subclass's module.

The inheritance hierarchy is independent of the type hierarchy. Therefore the type implemented by the superclass might not be a supertype of the type implemented by the subclass. For example, it might be convenient to implement stacks by inheriting from a class that implements lists even though stack is not a subtype of list. Note also that either class might not implement a type.

The instance variables declared in the subclass are in addition to those of the superclass: objects of the subclass have all the *inherited* instance variables of the superclass as well as those of the subclass. However, the inherited variables cannot be accessed directly in the subclass; it can access them only by calling the superclass methods.

Objects of the subclass have all the methods of the superclass, although the hidden methods aren't visible within the subclass. Visible superclass methods can be renamed: the second *idn* in a *renames* clause gives the name of the method in the superclass; the first gives the new name. For example,

```
C = class for T inherits D {foo for bar}
```

renames D's bar method to foo. All superclass methods not mentioned in a renaming clause retain their original names. The effect of the renamings is that the superclass appears to have been rewritten with the names needed in the subclass.

Visible superclass methods can be *inherited* by the subclass: this is accomplished by simply not giving an implementation of a method of that name. Subclass methods can also be implemented explicitly. If such a method has the same name as a visible superclass method, the new implementation *overrides* the associated superclass method. In such a case, the subclass object has both the overridden method and the new method; the overridden method is a private method and it can be named using the the special form ˆ *idn*. For example, if the subclass overrides visible superclass method m, the overriding definition is named m, and the overridden method is named ˆ m. Thus code in the subclass and its module can continue to call the overridden method using the ˆ form.

Methods overridden by a subclass can affect the behavior of superclass methods. If a superclass method calls a method n that has been overridden by the subclass, the implementation of n provided by the subclass will run, not the implementation provided by the superclass. For example, consider superclass method

```
m ( ) returns (int)
   return (self.n())
end m
```

and suppose that n is visible and has been overridden in the subclass. When m is called on an object of the subclass, its call of n goes to the overriding definition. Therefore, we require that the overriding definition have a signature that is a subtype of the signature of the method it overrides.

USAGE NOTE

This restriction means that the call of an overridden method in the inherited superclass method will be legal. For the call to be sensible, the overridden method ought to behave like that of the superclass as well.

Within a class-constructor for the subclass, or within a make statement of a maker for the subclass, a maker of the superclass must be called to initialize the superclass fields of the new object. For example, inside a parameterized maker make_stack[T], we might have a **make** statement:

make { ... ; make_list[T](...) }

where stack is being implemented as a subclass of list and make_list is a maker provided for list.

A subclass type is *not* a subtype of the superclass type, nor is it a subtype of the type implemented by the superclass, unless the type implemented by the subclass is a subtype of the type implemented by the superclass. With one exception, ordinary type checking restrictions apply to subclass objects, e.g., a subclass object cannot be assigned to a variable whose type is the superclass type. The exception is that the code of a subclass and its module can call the methods and routines provided by its superclass passing in subclass objects as arguments in positions where an object of the superclass type, or of the type implemented by the superclass, is required.

A subclass can use its **hides** clause to avoid exporting inherited methods to its subclasses and can use its **provides** clause to export methods, and also routines and makers implemented in its module. However, it cannot use the ^ notation to name methods in the **provides** clause, and it cannot provide any methods or routines that have the superclass type in their signatures (since its superclass is not visible to its subclasses).

10.5.4 Rules for Superclasses

A superclass should guarantee that subclasses cannot interfere with the correct functioning of its code and the code in its module. This can be accomplished by care in implementing the module and by using the **provides** and **hides** clauses appropriately. Below we discuss two problems that must be avoided: masquerading, and propagation of bad information.

None of the methods or routines that the superclass provides to its subclasses should create an *alias* for one of the superclass objects. If such a method or routine were provided, it could be used by the code in the subclass and its module to cause subclass objects to *masquerade* as superclass objects. Masquerading is bad because subclass objects may behave differently than superclass objects. For example, if the bag copy method were implemented:

```
copy ( ) returns (bag[T])
    return (self)
    end copy
```

it would create an alias for **self**. If copy were provided to subclasses of bag, return(x.copy()) within the subclass, where x is an object of the subclass, will cause a subclass object to appear to be a bag object, even though it might not behave like one.

The second problem — propagation of bad information — occurs only if a provided method, routine, or maker violates the superclass rep invariant, and is easily avoided by not providing such *violators*. However, providing violators is sometimes useful, and they are bad only in combination with *propagators*: methods, routines, and makers that perform incorrectly if the superclass rep invariant doesn't hold for some object they access. For example, suppose the bag copy method simply copied its instance variables; if the rep invariant weren't satisfied, the result would be a bag object that did not satisfy the rep invariant. So, a class that provides violators should not also provide propagator methods, routines, and makers to its subclasses. In addition its module should not export to its users any propagator routines, and its other classes should not export any propagator methods.

10.5.5 Example of Inheritance

This section illustrates the use of inheritance by means of a simple example.

Suppose we want to implement the stack abstraction specified in Section *(9.4)* as a subclass of the bag implementation given in Section *(10.4.3)*. To do so, at the least we must provide some makers for stack to use: a maker for initializing an empty stack, and also some way of initializing the stack returned by the copy method. We must also consider how to implement the top method.

Here is one solution to these problems: bag provides its subclasses with access to the array that contains its elements, e.g., by providing a get_els method, which returns the els component of a **bag**. However, this method is effectively a violator, since it allows the subclass to modify the array, thus violating the rep invariant. Therefore care must be taken to not provide any propagators.

```
module implements create_bag

brep = class[T] for bag[T]
        provides mk_brep, mk_copy, get_els
        hides copy

        sz: int implements size                 % implementation of size method
        els: array[T] implements get_els        % implementation of get_els method

        % the rep invariant is: sz = els.size( )

        put (x: T)
                els.append(x)
                sz := sz + 1
                end put

        get ( ) returns (T) signals (empty)
                x: T := els.remove( ) except when bounds: signal empty end
                sz := sz − 1
                return (x)
                end get

        copy ( ) returns (bag[T])
            where T has copy ( ) returns (T)
                return (brep[T]{sz := sz, els := els.copy( )})
                end copy

        end brep

create_bag [T] ( ) returns (bag[T])
        return (brep[T]{sz := 0, els := array_create[T]( )})
        end create_bag

mk_brep[T] ( ) makes (brep[T])
            make {sz := 0, els := array_create[T]( )}
            end mk_brep

mk_copy[T] (x: brep[T]) makes (brep[T])
    where T has copy ( ) returns (T)
```

```
        make {sz := x.els.size(), els := x.els.copy( )}
        end mk_copy

    end
```

The mk_copy maker ensures the rep invariant by setting the sz field of the new object appropriately. The copy method is hidden since it is a propagator. If the bag implementation had not exported a violator, it would not be necessary to hide the copy method.

Here is an implementation of stack that demonstrates the use of inheritance.

```
    module implements create_stack

    srep = class[T] for stack[T] inherits brep[T] {push for put, pop for get}

        top ( ) returns (T) signals (empty)
             return (self.get_els( ).top( ))
                    except when bounds: signal empty end
            end top

        copy ( ) returns (stack[T])
          where T has copy ( ) returns (T)
            return (srep[T]{mk_copy[T](self)})
            end copy

        end srep

    create_stack [T] ( ) returns (stack[T])
        return (srep[T]{mk_brep[T]( )})
        end create_stack

    end
```

For this implementation, it is not necessary to define any additional instance variables. Note that the stack implementation is dependent on the details of the bag implementation, e.g., that put adds the new element to the high end of the array and get removes the newest element.

In this example, the implemented types (stack and bag) are in a subtype relationship that mirrors the inheritance relationship of the implementing classes, as shown in Figure 10.1. This means that srep indirectly provides another implementation of bag. However, Theta does not require that such a relationship exists; another class could inherit from brep without implementing a subtype of bag.

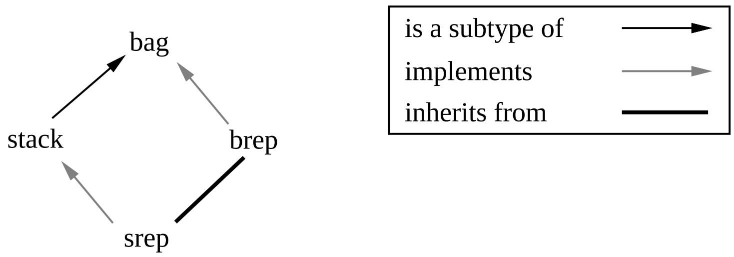

Figure 10.1: The subtype and inheritance relationships of srep

A Reference Grammar

This section presents the grammar for Theta. This grammar is authoritative wherever it conflicts with syntax productions in the previous section of the manual.

program units, routine specifications, and equates

program_unit → *routine_interface* | *equate* | *type_interface* | *module*

routine_interface → *proc_interface* | *iter_interface*

equate → *idn* = *expr* | *idn* = *type_designator*

proc_interface → *idn* **[** *parms* **]** *formal_args* **[** *returns* **]** **[** *signals* **]** **[** *where* **]**

iter_interface → *idn* **[** *parms* **]** *formal_args* *yields* **[** *signals* **]** **[** *where* **]**

parms → [*idn_list*]

idn_list → *idn* **[** , *idn* **]***

formal_args → (**[** *decl* **[** , *decl* **]*** **]**)

decl → *idn_list* : *type_designator*

returns → **returns** (*type_list*)

type_list → *type_designator* **[** , *type_designator* **]***

yields → **yields** (*type_list*)

signals → **signals** (*exception* **[** , *exception* **]***)

exception → *name* **[** (*type_list*) **]**

name → *idn*

where → **where** *restriction* **[** , *restriction* **]***

restriction → *idn* **has** *nonparam_interface* **[** , *nonparam_interface* **]***

nonparam_interface → *idn* *nonparam_proc_sig* | *idn* *nonparam_iter_sig*

nonparam_proc_sig → (**[** *type_list* **]**) **[** *returns* **]** **[** *signals* **]**

nonparam_iter_sig → (**[** *type_list* **]**) *yields* **[** *signals* **]**

type specifications

type_interface → *idn* = **type** [*parms*] [*supertypes*] [*where*]
[*interface_or_equate*]*
end *idn*

supertypes → < *superinfo* [, *superinfo*]*

superinfo → *type_designator* [{ *renames* [, *renames*]* }]

renames → *idn* **for** *idn*

interface_or_equate → *routine_interface* | *equate*

type designators

type_designator → *simple_type_desig* | *routine_type_desig* | *parm_type_desig* | *tagged_type_desig*

simple_type_desig → *idn* | null | bool | char | int | real | string | any

routine_type_desig → **proc** *nonparam_proc_sig* | **iter** *nonparam_iter_sig*

parm_type_desig → *parm_type* *actual_parms*

parm_type → *idn* | array | sequence | vector | maybe

actual_parms → [*type_list*]

tagged_type_desig → *tagged_type* [*field* [, *field*]*]

tagged_type → record | struct | oneof

field → *idn_list* : *type_designator*

modules

module → **module** [*implements*] [*impl_elt*]* **end**

implements → **implements** *exported_item* [, *exported_item*]*

exported_item → *idn* [{ *idn_list* }] | *idn actual_parms* { *idn_list* }

impl_elt → *routine_def* | *class_def* | *maker_def* | *equate*

routine_def → *routine_interface body* **end** *idn*

maker_def → *maker_interface body* **end** *idn*

maker_interface → *idn* [*parms*] *formal_args makes* [*signals*] [*where*]

makes → **makes** (*type_designator*)

body → [*equate*]* [*statement*]*

class_def → *idn* = **class** [*parms*] [*for_type*] [*inherits*]
[*where*] [*provides*] [*hides*]
[*equate_or_ivar_decl*]*
[*equate_or_routine_def*]*
end *idn*

for_type → **for** *type_designator*

inherits → **inherits** *type_designator* [{ *renames* [, *renames*]* }]

provides → **provides** *idn_list*

hides → **hides** *idn_list*

equate_or_ivar_decl → *equate* | *ivar_decl*

ivar_decl → *decl* | *decl_with_impls*

decl_with_impls → *idn* : *type_designator* **implements** *idn* [, *idn*]

equate_or_routine_def → *equate* | *routine_def*

<u>*statements*</u>

statement → *decl*
| *lhs* := *expr* [, *expr*]*
| *lhs* := *invoc*
| *simple_invoc*
| *primary* [*expr*] := *expr*
| **return** [(*expr* [, *expr*]*)]
| **yield** (*expr* [, *expr*]*)
| **signal** *name* [(*expr* [, *expr*]*)]
| **exit** *name* [(*expr* [, *expr*]*)]
| **if** *expr* **then** *body* [**elseif** *expr* **then** *body*]* [**else** *body*] **end**
| **while** *expr* **do** *body* **end**
| **for** *for_idns* **in** *invoc* **do** *body* **end**
| **break**
| **continue**
| **begin** *body* **end**
| **tagcase** *expr* *tag_arm* [*tag_arm*]* [**others** : *body*] **end**
| **typecase** *expr* *type_arm* [*type_arm*]* [**others** : *body*] **end**
| *statement* **except** [*handler*]* [**others** [(*idn* : string)] : *body*] **end**
| *statement* **resignal** *name* [, *name*]*
| **make** { [*ivar_inits*] } [**then** *body* **end**]

lhs → *var* [, *var*]* | *decl* [, *decl*]*

var → *idn* | *primary* . *idn*

invoc → *expr* ([*args*])

args → *expr* [, *expr*]* [, *varying_args*] | *varying_args*

varying_args → **..** | **..** *expr* [, *expr*]*

simple_invoc → *primary* ([*args*])

for_idns → *idn_list* | *decl* [, *decl*]*

tag_arm → **when** *name* [, *name*]* [(*idn* : *type_designator*)] : *body*

type_arm → **when** *type_designator* [(*idn*)] : *body*

handler → **when** *name* [, *name*]* [(*decl* [, *decl*]*)] : *body*

ivar_inits → *field_inits* ; *maker_invoc* | *field_inits* | *maker_invoc*

field_inits → *field_init* [, *field_init*]*

field_init → *idn* := *expr*

maker_invoc → *idn* [*actual_parms*] ([*args*])

expressions

```
expr        →  simple_expr
            |  expr . idn
            |  expr . method_idn
            |  invoc
            |  expr [ expr ]
            |  ~ expr
            |  − expr
            |  expr binary_op expr
            |  ( expr )

primary     →  simple_expr
            |  primary . idn
            |  primary . method_idn
            |  simple_invoc
            |  primary [ expr ]

simple_expr →  literal
            |  idn [ actual_parms ]
            |  self
            |  method_idn
            |  tagged_type_desig { field_inits }
            |  type_designator { [ ivar_inits ] }
            |  bind ( expr bind_args )

method_idn  →  [ ^ ] idn [ actual_parms ]

bind_args   →  [ , bind_arg ]* [ , varying_args ]

bind_arg    →  expr | *

literal     →  nil
            |  true
            |  false
            |  int_literal
            |  char_literal
            |  real_literal
            |  string_literal

binary_op   →  ** | // | / | * | || | + | − | < | <= | = | >= | > | == | ~= | & | |
```

B Built-in Types and Parameterized Types

This appendix supplements the material of Chapter 3 by providing a preliminary description of the built-in types and parameterized types. The definitions should be considered preliminary; we may very well make changes to them later.

All the built-in types except for any have equal, similar, copy, and unparse methods. These methods are optional for the parameterized types such as array[T]: an instantiation will have one of these methods only if each of the actual parameter types has the method.

None of the built-in types and parameterized types can have subtypes, except for any, which is the supertype of all types. There is no type hierarchy relating any of the other built-in types, except that there is a rich hierarchy for routine types.

The built-in types and parameterized types are implemented by classes that cannot be subclassed.

The Theta environment contains a number of equates that describe various properties and limitations of the built-in types. The corresponding type definitions in this Appendix describe these equates in more detail.

B.1 Any

The type any is the supertype of all types. It has no methods or associated routines.

B.2 Null

The type null is an immutable type with a single object, denoted by the literal nil. It is used primarily as a placeholder in a oneof to handle the "empty" case.

Methods for type null

```
equal (n: null) returns (bool)
   % effects   returns true

similar (n: null) returns (bool)
   % effects   returns true

copy ( ) returns (null)
   % effects   returns nil

unparse ( ) returns (string)
   % effects   returns the three-character string ''nil''
```

B.3 Bool

Type bool is an immutable type with two objects, denoted by the literals true and false.

Methods for type bool

```
not ( ) returns (bool)
   % effects   returns ~self

and (x: bool) returns (bool)
   % effects   returns the boolean and of self and x

or (x: bool) returns (bool)
   % effects   returns the boolean or of self and x

xor (x: bool) returns (bool)
   % effects   returns the boolean xor of self and x

equal (x: bool) returns (bool)
   % effects   returns true if self and x are either both true or
   %           both false; else returns false

similar (x: bool) returns (bool)
   % effects   returns true if self and x are either both true or both
   %           false; else returns false

copy ( ) returns (bool)
   % effects   returns self

unparse ( ) returns (string)
   % effects   if self = true returns the four-character string "true";
   %           else returns the five-character string "false"
```

B.4 Int

Type int contains a subset of the integer objects. Its objects are immutable.

Int provides a number of literal forms for its objects. Ints can be denoted in any base from 2 to 36 inclusive using the literal form:

integer_literal → **[** *base* _ **]** *integer_digits*

The base and the underscore can be omitted to use the default base 10. The base is always interpreted as a decimal number and must be between 2 and 36 inclusive. The digits of the integer are indicated using the numbers 0 to 9 and, if necessary, the appropriate letters of the alphabet. For example, for base 16, the digits are 0...9, a...f. The upper and lower case characters of the alphabet are considered equivalent in integer literals. No spaces are allowed within an integer literal. Here are some examples of literals with their corresponding base 10 form:

Literal	Base 10 form
25	25
10_25	25
16_1c	28
16_1C	28
8_72	58
3_2001	55
2_1101	13

The following equates in the Theta environment denote the range of representable integers.

int_max	an integer value indicating the smallest representable integer
int_min	an integer value indicating the largest representable integer

Integer values are representable in 32 bits and therefore under a twos-complement machine representation for integers, int_max will be $2^{31} - 1$ and int_min will be -2^{31}.

<u>Methods for type int</u>

negate ( ) **returns** (int) **signals** (overflow)
% **effects** returns −self; signals overflow if the result is not
% in the representable range.

add (x: int) **returns** (int) **signals** (overflow)
% **effects** returns self + x; signals overflow if the sum is not
% in the representable range.

subtract (x: int) **returns** (int) **signals** (overflow)
% **effects** returns self − x; signals overflow if the result is not
% in the representable range.

multiply (x: int) **returns** (int) **signals** (overflow)
% **effects** returns self ∗ x; signals overflow if the result is not
% in the representable range.

divide (x: int) **returns** (int) **signals** (zero_divide, overflow)
% **effects** if x = 0 signals zero_divide. Otherwise returns self/x.
% The result is rounded toward negative infinity. Signals

```
   %            overflow if the result is not in the representable range.

mod (x: int) returns (int) signals (zero_divide)
   % effects   if x = 0 signals zero_divide. Otherwise returns self mod x.
   %           This is the remainder when self is divided by x, and
   %           is defined such that self = (self/x) * x + self.mod(x)

power (x: int) returns (int) signals (negative_exponent, overflow)
   % effects   if x < 0, signals negative_exponent. Otherwise returns self to the x power;
   %           If self = 0, then self.power(0) is defined to be 1.
   %           Signals overflow if the result is not in the representable range.

abs ( ) returns (int) signals (overflow)
   % effects   returns |self|; signals overflow if the result is not in the representable range.

to (bound: int) yields (int)
    % effects   yields the ints from self to bound in order; if bound < self yields nothing

to_by (bound: int, step: int) yields (int)
   % effects   yields the ints self, self + step, ... up to bound inclusive.

max (x: int) returns (int)
   % effects   returns the larger of self and x

min (x: int) returns (int)
   % effects   returns the smaller of self and x

lt (x: int) returns (bool)
   % effects   returns (self < x)

le (x: int) returns (bool)
   % effects   returns (self ≤ x)

gt (x: int) returns (bool)
    % effects   returns (self > x)

ge (x: int) returns (bool)
    % effects   returns (self ≥ x)

equal (x: int) returns (bool)
   % effects   returns (self = x)

similar (x: int) returns (bool)
   % effects   returns (self = x)

copy ( ) returns (int)
   % effects   returns self

unparse ( ) returns (string)
   % effects   returns a string representing self in base 10. E.g., if self is 123, returns
   %           the three character string ''123''

to_real ( ) returns (real)
   % effects   converts self to a real and returns the result;
```

```
%            rounds toward zero;
%            assumes the range of real values covers range of integer values

to_char ( ) returns (char) signals (illegal_char)
  % effects  If self represents the ASCII code for a character, then
  %          returns that character, else signals illegal_char.
```

B.5 Real

A real is an immutable, floating-point number. It is represented by a subset of the IEEE single-precision format floating-point numbers, with the following changes:

- Positive and negative infinity are not representable.
- Positive and negative zero are not distinguishable.
- Denormalized values are not used.

The following equates from the Theta environment describe specifics of the range, granularity and format of the real numbers as represented in Theta:

real_emin	an integer value indicating the smallest binary exponent.
real_emax	an integer value indicating the largest binary exponent.
real_precision	an integer value indicating the number of binary digits of precision available in reals.

The following values describe the range of values and the granularity.

real_max	a real value indicating the maximum value of a real.
real_min	a real value indicating the minimum value of a normalized real number.
real_epsilon	a real value indicating the value of a bit **in** the least significant position in the fraction when the exponent is zero.
real_round_style	an enumerated value indicating the rounding style: real_round_toward_zero real_round_toward_minus_infinity real_round_other

Note that

$$\begin{aligned}
\text{real_max} &= (1 - 2^{-\text{real_precision}}) * 2^{\text{real_emax}} \\
\text{real_min} &= 2^{(\text{real_emin}-1)} \\
\text{real_epsilon} &= 2^{(1-\text{real_precision})}
\end{aligned}$$

Real literals can have any one of the following forms:

$$\begin{array}{rcl}
\textit{real_literal} & \rightarrow & [\ \textit{digits}\ ]\ .\ \textit{digits}\ [\ \textit{exponent}\ ] \\
 & | & \textit{digits exponent} \\
\textit{exponent} & \rightarrow & \texttt{e}\ [\ + \mid -\ ]\ \textit{digits} \\
 & | & \texttt{E}\ [\ + \mid -\ ]\ \textit{digits}
\end{array}$$

where *digits* is a non-empty sequence of digits in the range 0–9. No spaces are allowed in the middle of a *real_literal*. Here are some examples of legal *real_literals*:

```
2.6
25.0
.05
5.2e-3
5.2e+3
2e10
.02E2
.0E5
```

Note that a decimal point must be followed by one or more digits, so the following are *not* legal real literals:

```
1.
45.e6
```

Methods for type real

negate ( ) **returns** (real)
  % **effects** returns $-$self.

add (x: real) **returns** (real) **signals** (overflow, underflow)
  % **effects** returns self $+$ x. Signals overflow if result is too big to be represented.
  % Signals underflow if result it too close to zero to be represented.

subtract (x: real) **returns** (real) **signals** (overflow, underflow)
  % **effects** returns self $-$ x. Signals overflow if result is too big to be represented.
  % Signals underflow if result it too close to zero to be represented.

multiply (x: real) **returns** (real) **signals** (overflow, underflow)
  % **effects** returns self $*$ x. Signals overflow if result is too big to be represented.
  % Signals underflow if result it too close to zero to be represented.

divide (x: real) **returns** (real) **signals** (overflow, underflow, zero_divide)
  % **effects** returns self / x. Signals overflow if result is too big to be represented.
  % Signals underflow if result it too close to zero to be represented.
  % Signals zero_divide if x $=$ 0.

power (x: real) **returns** (real) **signals** (overflow, underflow, complex_result, zero_divide)
  % **effects** returns self raised to the x power.
  % Signals overflow if result is too big to be represented.
  % Signals underflow if result it too close to zero to be represented.
  % Signals zero_divide if self $=$ 0 and x $<$ 0.
  % Signals complex_result if self $<$ 0 and x has a fractional component.

abs ( ) **returns** (real)
  % **effects** returns the absolute value of self.

exponent ( ) **returns** (int) **signals** (undefined)
  % **effects** returns n such that $2^{n-1} \leq \text{self} < 2^n$
  % Signals undefined if self $=$ 0.

mantissa ( ) **returns** (real)
  % **effects** returns x such that $x \times 2^{\text{self.exponent()}} = \text{self}$

max (x: real) **returns** (real)
  % **effects** returns the larger of self and x

min (x: real) **returns** (real)
  % **effects** returns the smaller of self and x

to_int ( ) **returns** (int) **signals** (overflow)
  % **effects** returns self rounded to the nearest integer (towards zero, in the case of a tie).

% Signals overflow if the rounded number can not be represented as an integer.

floor ( ) **returns** (real)
% **effects** returns self rounded toward negative infinity.

ceiling ( ) **returns** (real)
% **effects** returns self rounded toward positive infinity.

lt (x: real) **returns** (bool)
% **effects** returns (self $<$ x)

le (x: real) **returns** (bool)
% **effects** returns (self $\leq$ x)

gt (x: real) **returns** (bool)
% **effects** returns (self $>$ x)

ge (x: real) **returns** (bool)
% **effects** returns (self $\geq$ x)

equal (x: real) **returns** (bool)
% **effects** returns (self $=$ x)

similar (x: real) **returns** (bool)
% **effects** returns (self $=$ x)

copy ( ) **returns** (real)
% **effects** returns self

unparse ( ) **returns** (string)
% **effects** returns the string representation of a literal corresponding to self.
% The general form is [−]intpart.fractpart[e+/−exp].

B.6 Char

Type char contains the ASCII characters. Its objects are immutable. Char literals are enclosed in single quotes. Printing ASCII characters (octal 40 through octal 176), other than single quote or backslash, can be written as that character enclosed in single quotes. Any character can be written by enclosing one of the following escape sequences in single quotes:

escape sequence	character
\'	single quote
\"	double quote
\\	backslash
\t	horizontal tab
\v	vertical tab
\n	newline character
\r	carriage return
\f	form feed (or new page)
\b	backspace
ddd	octal value (specified by exactly three octal digits)

If the octal value specified in the \ddd form does not correspond to a legal ASCII value, there will be a compile-time error. Examples of character literals are

'', '7', 'a', '"', '\"', '\'', '\n', '\000'

<u>Methods for type char</u>

to_int ( ) **returns** (int)
  % **effects** returns the integer ASCII code for self.

to_string ( ) **returns** (string)
  % **effects** returns a one character string containing self.

lt (c: char) **returns** (bool)
le (c: char) **returns** (bool)
ge (c: char) **returns** (bool)
gt (c: char) **returns** (bool)
  % **effects** these orderings are consistent with the ASCII numbering of chars.

equal (c: char) **returns** (bool)
  % **effects** returns true if and only if self is same as c

similar (c: char) **returns** (bool)
  % **effects** returns true if and only if self is same as c

copy ( ) **returns** (char)
  % **effects** returns self

unparse ( ) **returns** (string)
  % **effects** If self is a printable character, then returns a one character string
  % containing self. Else returns a string containing an escape sequence
  % that represents self.

B.7 String

A string is an immutable sequence of character. It is indexable; the low bound of a string is 1. The size of strings is limited to what can be indexed using ints. Thus the largest string has high bound (and size) int_max.

A string literal is written as a sequence of zero or more character representations enclosed in double quotes. Within a string literal, a printing ASCII character other than double quote or backslash is represented by itself. Any character can be represented by using the escape sequences listed for characters. Examples of string literals are

```
"", "Item\tCost", "hello\n", "\"string\"", "'c'"
```

Methods for type string

```
length ( ) returns (int)
   % effects   returns the size of self.

empty ( ) returns (bool)
   % effects   returns (self.length() = 0)

fetch (i: int) returns (char) signals (bounds)
   % effects   if i is within bounds returns the ith character of self else signals bounds.

rest (i: int) returns (string) signals (bounds)
   % effects   returns a string containing self[i],...,self[self.length()];
   %           signals bounds if i is not a legal index in self.

first (i: int) returns (string) signals (bounds)
   % effects   returns a string containing self[1], ..., self[i];
   %           signals bounds if i is not a legal index in self.

concat (s: string) returns (string)
   % effects   returns a string containing the characters of self followed by the
   %           characters of s; signals failure if the resulting string is bigger
   %           than what can be represented

append (c: char) returns (string)
   % effects   returns a string containing the characters of self followed by
   %           c;  signals failure if the resulting string is bigger than what
   %           can be represented.  This method has the same effect as
   %                   self.concat(c.to_string())

extract (at: int, count: int) returns (string) signals (bounds, negative_size)
   % effects   If count is negative, signals negative_size.
   %           If at isn't a legal index in self, signals bounds.
   %           Otherwise returns a new string containing the characters
   %           self[at], self[at+1], ...; the new string contains
   %           min(count, self.length() - at + 1) characters.  For example, if s = "abcdef", then
   %             s.substr(2, 3)        = "bcd"
   %             s.substr(2, 7)        = "bcdef"

chars ( ) yields (char)
   % effects   yields the characters of self in order from the first to last.
```

```
lt (s: string) returns (bool)
le (s: string) returns (bool)
ge (s: string) returns (bool)
gt (s: string) returns (bool)
    % effects   these are the usual lexicographic ordering operations based on the
    %           ASCII ordering of the characters contained in self and s.

equal (s: string) returns (bool)
   % effects   returns (self = s)

similar (s: string) returns (bool)
   % effects   returns (self = s)

copy ( ) returns (string)
   % effects   returns a string that contains self[1],...,self[self.size()]

unparse ( ) returns (string)
   % effects   Returns the concatenation of the strings produced by calling unparse
   %           on all of the characters contained in self.
```

Routines for type string

```
string_create (chars: sequence[char]) returns (string)
   % effects   returns a new string containing the
   %           characters in the sequence in order.
```

B.8 Array

An array is an indexable, mutable collection. It is a parameterized type; the actual value of the parameter when array is instantiated determines the type of element in the array. For example, all elements of an array[foo] will belong to subtypes of foo.

An array can grow and shrink dynamically. The initial bounds of an array are determined when it is created. As an array grows (shrinks) its length increases (decreases). The low bound is always less than or equal to the high bound, except if the array is empty; in this case, a.low() = a.high() + 1. It is always the case that a.size = a.high() - a.low() + 1. The array bounds are limited to what can be indexed using ints and to a size that can be represented as an int; thus the low bound of an array must be $\geq$ int_min, and the high bound and size must be $\leq$ int_max.

Consider an array a. The legal indexes of a are all integers i such that a.low() $\leq$ i $\leq$ a.high(). The elements in the array occur at element positions a.low(), a.low() + 1, ...,; if i is a legal index, then we refer to the element it indexes as a[i]. E.g., if a has low bound -2 and contains 1, 2 and 3, then -2, -1, and 0 are legal indexes and a[-2], a[-1], and a[0] are legal array elements.

The array routine array_create allows the new array to be created using the varying arguments form *(6.1)*. For example

```
a: array[int] := array_create[int](0, .. 6, 9, 17)
```

creates a new array with low bound 0 and containing the elements 6, 9, and 17.

Methods for type array[T]

```
empty ( ) returns (bool)
   % effects   returns true if the array is empty, else returns false.

length ( ) returns (int)
   % effects   returns the length of the array (a count of the number of elements it contains).

low ( ) returns (int)
   % effects   returns the low bound of the array.

high ( ) returns (int)
   % effects   returns the high bound of the array.

fetch (i: int) returns (T) signals (bounds)
   % effects   if i is not a legal index in self, signals bounds.  Otherwise
   %           returns the element self[i].

bottom ( ) returns (T) signals (bounds)
   % effects   if self is empty, signals bounds.  Otherwise returns the first element of self.

top ( ) returns (T) signals (bounds)
   % effects   if self is empty, signals bounds.  Otherwise, returns the last element of self.

store (i: int, v: T) signals (bounds)
   % modifies  self
   % effects   If i is not a legal index in self, signals bounds.
   %           Otherwise, sets the element at self[i] to v.

append (x: T)
   % modifies  self
```

```
    % effects    If adding x to the high end of self would cause the high bound
    %            or size of self to become too large, signals failure.
    %            otherwise, adds x to the high end of self.

remove ( ) returns (T) signals (bounds)
    % modifies  self
    % effects    if self is empty signals bounds.
    %            otherwise, removes and returns the highest element of self.

append_low (x: T)
    % modifies  self
    % effects    if adding x to the low end of self would cause the low bound
    %            or size of self to exceed the limits, signals failure.
    %            otherwise adds x to the low end of self.

remove_low ( ) returns (T) signals (bounds)
    % modifies  self
    % effects    if self is empty signals bounds.
    %            otherwise, removes and returns the lowest element of self.

predict (cnt: int)
    % effects  the method has no effect on the array state.  However, it predicts
    %          that the array will grow to have size cnt, and the append calls
    %          that cause the array to grow may operate faster as a result
    %          of the call on predict.

set_low (lb: int)
    % modifies  self
    % effects    if changing the low bound of self to lb would cause its size or
    %            high bound to become too large, signals failure.
    %            otherwise, sets the low bound of self to lb and renumbers the
    %            array elements accordingly.

trim (lb: int, count: int) signals (negative_size, bounds)
    % modifies  self
    % effects    if count < 0, signals negative_size.  if lb < low() or lb > high() + 1,
    %            signals bounds.  Otherwise, removes all elements with indices less than lb
    %            or greater than lb + count − 1; the new low bound is lb.  For example,
    %            a.trim(3, 2), where a is a 5 element array[int] with low bound 0 and
    %            containing the elements 1, 2, 3, 4, 5, changes a to have low bound 3 and
    %            contain the two elements 4, 5.  a.trim(3, 12) has the same effect.

indexes ( ) yields (int)
    % effects  yields the legal indices of self from the low bound of pre(self) to
    %          the high bound of pre(self), where pre(self) is the value of self at
    %          the time of the call.  Note that any modifications to the array done
    %          in the loop body do not affect the integers yielded by this method.

elements ( ) yields (T)
    % effects  The effect of x.elements() is equivalent to the following body:
    %            for i: int in x.indexes() do
    %                yield (x[i]) except when bounds: signal failure(...) end
    %                end
    %        Thus if the loop body does not modify the array, the array
```

% elements are yielded in order. Note, however, that changes
% made by the loop body can affect what is yielded.

equal (a: array[T]) **returns** (bool)
% **effects** returns true if self and a are the same array object.

similar (a: array[T]) **returns** (bool)
where T **has** similar (T) **returns** (bool)
% **effects** returns true if self and a have the same size and low bound and the elements at
% corresponding positions are similar (using the T similar method to do the test).

copy ( ) **returns** (array[T])
where T **has** copy ( ) **returns** (T)
% **effects** returns a new array with the same size and low bound as self and containing
% a copy of each element of self (using T copy) in the corresponding positions.

unparse ( ) **returns** (string)
where T **has** unparse ( ) **returns** (string)
% **effects** produces a string representation of the contents of self using the
% T unparse method to produce string images of the elements.
% The resulting string has the form array[L: e_L,...,e_H],
% where e_i is obtained by calling the T unparse method for that element,
% and L and H are the low and high bounds of the array.

Routines for type array[T]

array_new[T] ( ) **returns** (array[T])
% **effects** creates new empty array with a low bound of 1

array_create[T] (lb: int, els: sequence[T]) **returns** (array[T])
% **effects** creates a new array with low bound lb containing the elements of the
% sequence in order; signals failure if the high bound of the
% resulting array would be too large.

array_generate[T] (lb: int, els: **iter** () **yields** (T)) **returns** (array[T])
% **effects** returns a new array containing the elements yielded by the
% iterator in order; signals failure if the high bound of the
% resulting array would be too large.

B.9 Sequence

A sequence is an immutable indexed collection. It is a parameterized type; the actual parameter of an instantiation determines the type of the sequence elements. The low bound of a sequence is always 1. The size of a sequence must be represented as an int and is therefore $\leq$ int_max.

The sequence routine sequence_create allows the new sequence to be created using the varying arguments form. *(6.1)*. For example

```
s: sequence[int] := sequence_create[int](.. 6, 9, 17)
```

creates a new sequence containing the elements 6, 9, and 17.

Methods for type sequence[T]

```
empty ( ) returns (bool)
   % effects   returns true if the sequence is empty, else returns false.

length ( ) returns (int)
   % effects   returns the length of the sequence (a count of the number of elements it contains).

fetch (i: int) returns (T) signals (bounds)
   % effects   if i is not a legal index in self, signals bounds.  Otherwise returns the
   %           ith element.

replace (i: int, v: T) returns (sequence[T]) signals (bounds)
   % effects   if i is not a legal index in self, signals bounds.  Otherwise, returns a new
   %           sequence containing the elements of self, except that the ith element is v.

append (x: T) returns (sequence[T])
   % effects   returns a new sequence containing the elements of self extended by x on the
   %           high end;  signals failure if the size of the new sequence would be too large.

extract (at: int, count: int ) returns (sequence[T]) signals (bounds, negative_size)
   % effects   if "at" is not a legal index, signal bounds, else if "count"
   %           is negative, signal negative_size. Otherwise,
   %           return a new sequence containing the elements
   %           self[at], ..., self[min(at + count − 1, self.length())]

concat (s: sequence[T]) returns (sequence[T])
   % effects   returns a new sequence containing the elements of self followed by the elements
   %           of s; signals failure if the size of the new sequence would be too large.

indexes ( ) yields (int)
   % effects   yields the legal indexes of self

elements ( ) yields (T)
   % effects   yields the elements of self in order from low bound to high bound.

equal (s: sequence[T]) returns (bool)
    where T has equal (T) returns (bool)
   % effects   returns true if self and s are indistinguishable, i.e., they are
   %           the same size and their corresponding elements are equal.

similar (s: sequence[T]) returns (bool)
    where T has similar (T) returns (bool)
```

```
   % effects  returns true if self and s are the same size and their corresponding
   %          elements are similar, using T similar to do the test.

copy ( ) returns (sequence[T])
     where T has copy ( ) returns (T)
   % effects  returns a new sequence with the same size as self and containing a copy
   %          of each element of self (using T copy) in the corresponding positions.

unparse ( ) returns (string)
    where T has unparse ( ) returns (string)
   % effects  produces a string representation of the contents of self using the
   %          T unparse method to produce string images of the elements.
   %          The resulting string has the form vector[e1,...en],
   %          where ei is obtained by calling the T unparse method for that element.
```

Routines for type sequence[T]

```
sequence_create[T] (els: sequence[T]) returns (sequence[T])
   % effects   returns a new sequence containing the
   %           elements of els in order.

sequence_generate[T] (n: int, els: iter () yields (T)) returns (sequence[T])
   signals (negative_size, not_enough)
   % effects  If n < 0, signals negative_size. If the iterator yields less than n elements,
   %          signals not_enough. Otherwise, returns a new sequence containing the
   %          first n elements yielded by the iterator in order.
```

B.10 Vector

A vector is a fixed-size, mutable, homogeneous collection with a low bound of 1. The elements of a vector[T] are all initialized with some element of type T. The size of a vector must be representable by an int and therefore must be less than or equal to int_max.

The vector routine vector_create allows the new vector to be created using the varying arguments form *(6.1)*. For example

```
v: vector[int] := vector_create[int](.. 6, 9, 17)
```

creates a new vector containing the elements 6, 9, and 17.

Methods for type vector[T]

```
length ( ) returns (int)
   % effects   returns the size of the vector (a count of the number of elements it contains).

fetch (i: int) returns (T) signals (bounds)
   % effects   if i is not a legal index in self, signals bounds.  Otherwise
   %           returns the element self[i].

store (i: int, v: T) signals (bounds)
   % modifies  self
   % effects     If i is not a legal index in self, signals bounds.
   %             Otherwise, sets the element at self[i] to v.

indexes ( ) yields (int)
   % effects   yields the legal indexes of self

elements ( ) yields (T)
   % effects  The effect of x.elements() is equivalent to the following body:
   %           for i: int in x.indexes() do yield(x[i]) end
   %          note that stores to the vector may affect the yielded values.

equal (v: vector[T]) returns (bool)
   % effects   returns true if self and a are the same object.

similar (v: vector[T]) returns (bool)
    where T has similar (T) returns (bool)
   % effects   returns true if self and v have the same size and the elements at
   %           corresponding positions are similar (using the T similar method to do the test).

copy ( ) returns (vector[T])
      where T has copy ( ) returns (T)
   % effects   returns a new vector with the same size as self and containing a copy
   %           of each element of self (using T copy) in the corresponding positions.

unparse ( ) returns (string)
    where T has unparse ( ) returns (string)
   % effects   produces a string representation of the contents of self using the
   %           T unparse method to produce string images of the elements.
   %           The resulting string has the form vector[e1,...en],
   %           where ei is obtained by calling the T unparse method for that element.
```

Routines for type vector[T]

```
vector_fill[T] ( count: int, elem: T) returns (vector[T]) signals (negative_size)
   % effects   creates a new vector containing count elements each of which is elem.
   %           Signals negative size if count is negative.

vector_create[T] (els: sequence[T]) returns (vector[T])
   % effects   returns a new vector containing the elements of the
   %           the sequence in order.

vector_generate[T] (n: int, els: iter () yields (T)) returns (vector[T])
   signals (negative_size, not_enough)
   % effects   If n < 0, signals negative_size.  If the iterator yields less than n elements,
   %           signals not_enough.  Otherwise, returns a new vector containing the
   %           first n elements yielded by the iterator in order.
```

B.11 Record

Records are mutable tuples consisting of a set of fields. An instantiation of the record type provides the name and type of each field of the record objects belonging to that type; there must be at least one field, and the field names must all be distinct. The case of the field names is not significant. Associated with each record type there is a record constructor that can be used to create new records of the type *(7.3)*. Each record type has a pair of methods for each field that allow users to read and modify the field; for a field named A, these methods are named A and set_A. Here is an example:

```
rt = record[a: int, b: real]  % a record type
x: rt                         % x will denote objects of this record type.
x := rt{a: 3, b: 1.1}         % construct an object
i: int := x.a( )              % read the a component
x.set_b(1.7)                  % modify the b component
```

In determining record type equality, the field names and types are significant and so is the order of the fields *(3)*.

A record type rt has the following methods. (st is the related struct type, i.e., it has the same field names and types in the same order.)

Methods for record type rt

```
a ( ) returns (T)
   % (here a is a field name and T is the corresponding type)
   % effects   returns the object stored in field a of self

set_a (x: T)
   % (here a is a field name and T is the corresponding type)
   % modifies  self
   % effects   stores x in field a of self

r_gets_r (x: rt)
   % modifies  self
   % effects   replaces the fields of self with the objects in the corresponding fields of x

r_gets_s (x: st)
   % modifies  self
   % effects   replaces the fields of self with the objects in the corresponding fields of x

to_s ( ) returns (st)
   % effects   returns an st object containing the elements of self in the corresponding fields.

equal (x: rt) returns (bool)
   % effects   returns true if self and x are the same object else returns false.

similar (x: rt) returns (bool)
     where all field types T of rt have similar(T) returns (bool)
   % effects   returns true if all corresponding fields of self and x are similar
   %           (using the T similar method for that field) else returns false

copy ( ) returns (rt)
      where all field types T of rt have copy ( ) returns (T)
   % effects   returns a new record each of whose fields contains a
```

```
%            copy of the object (obtained by calling that object's copy
%            method) in the corresponding field of self
```

unparse ( ) **returns** (string)
  where all field types T have unparse ( ) **returns** (string)
  % **effects** returns a string representing the value of self. The form is
  % record{n_1: f_1,...,n_n: f_n}, where n_i is the name of the ith
  % record field and f_i is obtained by calling the unparse method
  % for the object in the corresponding field.

B.12 Struct

Structs are immutable records. Like the record types, there is a method to read each field of a struct. Structs are created using constructors; the form of these constructors is identical to those for type record.

A struct type st has the following methods. (rt is the related record type, i.e., it has the same field names and types in the same order.)

Methods for struct type st

a ( ) **returns** (T)
  % (here a is a field name and T is the corresponding type)
  % **effects** returns the object stored in the a component of self.

replace_a (x: T) **returns** (st)
  % **effects** returns a new struct containing the objects in the
  % corresponding fields of self except that x is in field a

to_r ( ) **returns** (rt)
  % **effects** returns a new record whose fields contain the objects in the
  % corresponding fields of self

equal (x: st) **returns** (bool)
  where all field types T of st have equal (T) **returns** (bool)
  % **effects** returns true if x and self are pairwise equal (using the equal
  % methods for the fields) else returns false

similar (x: st) **returns** (bool)
  where all field types T of st have similar (T) **returns** (bool)
  % **effects** returns true if x and self are pairwise similar (using the similar
  % methods for the fields) else returns false

copy ( ) **returns** (st)
  where all field types T of st have copy ( ) **returns** (T)
  % **effects** returns a new struct containing as components copies of the objects
  % (using the object's copy method) in the corresponding fields of self.

unparse ( ) **returns** (string)
  where all field types T of st have unparse ( ) **returns** (string)
  % **effects** returns a string representing the value of self. The form of the
  % string is struct{n_1: f_1,..., n_n: f_n}, where n_i is the name of the
  % ith struct field and f_i is the unparsing of that field.

B.13 Oneof

Oneofs are immutable tagged objects. An instantiation of the oneof type provides the name and associated type of each possible tag for the oneof objects belonging to that type; there must be at least one tag, and the tag names must all be distinct. Each object of the type has one of these tags, and a value of the associated type. Methods exist to determine the tag and value of a oneof object, but usually oneofs are decomposed using the **tagcase** statement *(8.11)*.

Oneofs are created using constructors. Associated with each oneof type there is a set of constructors, one for each tag of the type. Here is an example:

```
ot = oneof[some: int, none: null]  % a oneof type
x: ot                              % a variable to denote objects of this type
x := ot{none: nil}                 % creating an object with the none tag

if x.is_none( ) then            % checking the tag
   x := ot{some: 7}             % creating an object with the some tag
   end

tagcase x                      % decomposing a oneof using the tagcase statement
   when none: ...
   when some(y: int): ...
   end
```

Methods for oneof type ot

```
is_a ( ) returns (bool)
   % (here a is a tag name and T is the corresponding type)
   % effects   returns true if the tag of self is a else returns false

value_a ( ) returns (T) signals (wrong_tag)
   % (here a is a tag name and T is the corresponding type)
   % effects   if the tag of self is a returns the associated object else signals wrong_tag

equal (x: ot) returns (bool)
     where all field types T have equal (T) returns (bool)
   % effects   returns true if self and x have the same tag and equal values (determined
   %           by calling the equal method for the value).

similar (x: ot) returns (bool)
     where all field types T have similar (T) returns (bool)
   % effects   returns true if self and x have the same tag and similar values (determined
   %           by calling the similar method for the value).

copy ( ) returns (ot)
       where all field types T have copy ( ) returns (T)
   % effects   returns a new oneof object with the same tag as self and whose value is a
   %           copy (obtained by calling the value's copy method) of that of self

unparse ( ) returns (string)
     where all field types T have unparse ( ) returns (string)
   % effects   returns a string representing the tag and value of self. The form of the string
   %           is oneof{t: v}, where t is a string corresponding to the current tag and v is
   %           produced by calling the unparse method of the value
```

B.14 Maybe

Maybes are just oneofs with a somewhat more convenient syntax and a possibly more efficient implementation; maybe[T] = oneof[empty: null, full: T]. The methods and routines for a maybe type are exactly as defined for the associated oneof type.

B.15 Routines

Routines (procedures and iterators) are immutable. They have only equal, similar, copy, and unparse methods. The equal and similar methods have *weak* definitions (see below); they are provided mainly so that structure and collection types that contain them can have these methods. E.g., records of type

```
record[i: int, p: proc(int) returns(int)]
```

will have equal and similar methods, since both field types have these methods.

Methods for routine type rt

```
equal (p: rt) returns (bool)
similar (p: rt) returns (bool)
   % effects  if a call returns true, then self and p are guaranteed to be
   %          indistinguishable: they return the same results for calls with
   %          equal arguments and have the same side effects for those calls.
   %          if the call of equal or similar returns false, there are no
   %          guarantees: self and p might or might not be indistinguishable.

copy ( ) returns (rt)
   % effects  returns self.

unparse ( ) returns (string)
   % effects  Returns a string that indicates whether this routine is a procedure or an
   %          iterator. The implementation is free to put more information about the
   %          routine (for example the routine signature) in the returned string.
```

C Additional Types and Routines

In addition to the built-in types and routines described in Appendix B, we expect most Theta implementations will provide a standard set of other useful types and routines. (They are "standard" in the sense that most implementations, if they provide them at all, will provide the full set; Theta implementations are not required to provide any of them.)

The standard types and routines can all be implemented using the built-in types and routines; they have no special status in the language. However, the Theta implementation is free to implement some of these types and routines directly rather than on top of the built-ins. Therefore, these types and routines, if available, may be more efficient than corresponding user-defined types and routines.

The full set of standard types and routines has not been determined; a future version of the reference manual will provide the definitive list. We expect to include some parsing routines (for converting strings to built-in types). Two types that we expect to include are `set` and `bag` (multiset).

Bibliography

[1] B. Liskov, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. The Language-Independent Interface of the Thor Persistent Object System. In O. Bukhres and A. Elmagarmid, editors, *Object-Oriented Multidatabase Systems*. Prentice-Hall, 1994. Also available as Programming Methodology Group Memo 80, MIT Lab. for Computer Science, Cambridge, MA, March 1994.

[2] B. Liskov, R. Gruber, P. Johnson, and L. Shrira. A Highly Available Object Repository for Use in a Heterogeneous Distributed System. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 255–266, Martha's Vineyard, MA, September 1990. Proceedings published as *Implementing Persistent Object Bases: Principles and Practice*, A. Dearle, G. Shaw, and S. Zdonik, editors, Morgan Kaufmann, 1991. Also available as Programming Methodology Group Memo 70, MIT Lab. for Computer Science, Cambridge, MA, August 1990.

[3] B. Liskov and J. Wing. Family Values: A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

Index