MIT/LCS/TR-201

A DENOTATIONAL SEMANTICS OF CLU

Robert W. Scheifler

This blank page was inserted to preserve pagination.

A Denotational Semantics of CLU

Robert William Scheifler

May 1978

This research was supported in part by the National Science Foundation under grant MCS74-21892 A01.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

A Denotational Semantics of CLU

by

Robert William Scheifler

Submitted to the
Department of Electrical Engineering and Computer Science
on 15 May 1978 in partial fulfillment of the requirements
for the degree of Master of Science.

Abstract

A denotational semantics of CLU, an object-oriented language supporting data abstractions, is presented. The definition is based on Scott's lattice-theoretic approach to the theory of computation. Modules, the basic unit of compilation, are represented in terms of a set of recursively defined domains called the abstract syntax. As part of checking the legality of a module, a transformation is made from the abstract syntax to a modified syntax; the transformation reflects the results of compile-time computations that need not be repeated at run-time. An execution environment is defined to be a set of objects, each with a particular state, together with a mapping from variable names to objects. The meaning of an expression or statement is a function that takes an environment and produces a result consisting of a termination condition, a list of zero or more objects, and a new environment; the termination condition is used to govern control flow. This uniform treatment of expressions and statements allows a simple definition of the run-time exception handling mechanism provided in CLU. The meaning of a procedure generator or iterator generator is a function that takes a list of actual parameters, a list of arguments, and an environment, and produces a result as for statement and expression evaluation. The meaning of parameters is given in terms of textual substitution. A non-parameterized routine is viewed as being a generator with an empty parameter list. The meaning of a cluster is a function that takes a list of actual cluster parameters and an operation name, and produces the meaning of that operation.

Keywords: denotational semantics, programming language semantics, CLU, Scott-Strachey method

Thesis Supervisor: Barbara H. Liskov
Associate Professor of Computer Science and Engineering

Acknowledgements

I particularly wish to thank Eliot Moss and Craig Schaffert for helpful discussions in the early stages of this thesis, and for plowing through rough drafts. I also wish to thank my thesis supervisor, Barbara Liskov, for supporting this research and for reading drafts. Finally, I wish to thank Alan Snyder for his text formatting program R, and numerous kind souls for keeping the Xerox Graphics Printer running; without R and the XGP this thesis would have been doomed.

Table of Contents

1. Introduction

With the design of the programming language CLU [Liskov77a, Liskov77b, Liskov78] essentially complete, it is important to have a precise semantic definition of the language. Such a definition can be useful to:

1) the designers, in verifying that CLU is well-defined,
2) implementors, in proving the correctness of system implementations,
3) programmers, in writing programs and proving them correct, and
4) other language designers, in gaining a precise understanding of CLU.

There are four basic styles of semantic definition. Grammar-oriented methods, such as attribute grammars [Knuth68], W-grammars [vanWijngaarden76], and production systems [Ledgard77], consist of a set of rules for generating legal programs together with a set of rules for defining the meaning of a program; the meaning is defined either by generating all possible execution sequences or by translating the program into a predefined programming or assertion language. Operational schemes, such as the Vienna Definition Language [Wegner72] and SEMANOL [Blum73], specify meaning by defining a metaprogram, executable on an abstract machine, to serve as an interpreter; the meaning of a program for particular inputs is given by a trace of the computation performed by the interpreter. Denotational methods, such as the Oxford definition method [Scott71a, Milne76, Stoy77], define the meaning of a program as a mathematical function from initial to final states, with intermediate state changes ignored. Axiomatic techniques, such as that proposed by Hoare [Hoare69], specify meaning with axiom schemas and rules of inference; the meaning of a program is a mathematical relation between predicates true of all states before and after execution.

Unfortunately, no single style of definition is well-suited to the needs of all four classes of people listed above [Hoare74]. Grammar-oriented and operational definitions are the most appropriate for implementors, whereas designers often need a denotational semantics, with "irrelevant" implementation details abstracted away. On the other hand, an axiomatic definition is the most useful for programmers. Thus several definitions may be needed. However, there should be one "standard" definition, against which all others must

ultimately (if indirectly) be compared.

This thesis presents a denotational semantics of CLU, based on the lattice-theoretic approach to the theory of computation as initiated by Scott [Scott70, Scott71b, Scott72]. We have chosen a denotational semantics for the following reasons:

1) Compile-time type-checking, a very important aspect of CLU, does not fit within an axiomatic framework but is easily described with denotational techniques. Hence a denotational semantics can be used to verify a compiler's implementation of type-checking.
2) Current axiomatic methods do not handle the object-oriented semantics of CLU. Specifically, the treatment of side-effects on objects is not well developed. We believe a denotational semantics will be a useful tool in extending axiomatic techniques to encompass object-oriented semantics. In particular, proving axiomatic and denotational definitions equivalent is generally much simpler than proving axiomatic and operational definitions equivalent.
3) Although a denotational semantics is not particularly suited to the task of verifying a code generator or a run-time system, a denotational semantics is generally simpler and more concise than an operational semantics, and thus is a better candidate for the standard definition.
4) A new formalism, based on a new model of computation, has been developed by Schaffert [Schaffert78b] and used to define CLU [Schaffert78a]. Designed explicitly for object-oriented languages supporting data abstractions, Schaffert's method combines various elements of operational, denotational, and axiomatic techniques. We wish to present an alternative definition of CLU using an established method, so that the usefulness of Schaffert's work can be evaluated for a non-trivial language.

We do not provide an informal description of the syntax and semantics of CLU. Rather, we assume the reader is familiar with [Liskov78]. We do describe the most basic elements of lattice theory necessary to understand the definition, but we assume familiarity with the λ-calculus [Church51]. We do not deal with the question of whether the definition is well-founded; readers interested in the details of fixed points and reflexive

domain constructions are referred to [Scott71b, Scott76] for proof.

Chapter 2 provides an overview of the definition, describing the most important and novel aspects. Chapter 3 presents the theory and notation used throughout the remainder of the thesis. An abstract syntax for CLU is given in Chapter 4, as well as a modified syntax used for defining meaning. Chapter 5 defines legal programs and the transformation from the abstract to the modified syntax, and Chapter 6 defines the meaning of legal programs. Indexes to the numerous domains and functions defined in these chapters are provided at the end of the thesis, and should prove useful aids to the reader. Chapter 7 concludes with some general comments about the definition and gives a comparison of our definition to the one done by Schaffert.

2. Overview

The basic unit of compilation in CLU is a *module*, an implementation of an abstraction. Our definition consists of two major parts. The first part, defining what constitutes a legal module, captures the notion of *compilation*, while the second part, defining the meaning of legal modules, captures the notion of *execution* (or *evaluation*). This chapter gives an overview of these two parts.

2.1 Legality

For a module to be legal, it must satisfy certain syntactic and semantic constraints. Checking the context-free syntactic constraints is part of parsing, and will not be dealt with. Instead we define an *abstract syntax*, in terms of a set of recursively defined domains, to represent all possible parse trees. The abstract syntax retains the basic syntactic structure of the program text but various details, such as formatting and the presence of comments, are not represented. The precise relationship between the abstract syntax and actual text is defined in an appendix. We group the checking of context-sensitive syntactic constraints along with type-checking under the term *legality-checking*.

2.1.1 Syntactic Transformations

Among other things, legality-checking involves the evaluation of constants, the substitution of actual values for equated identifiers, the expansion of abbreviated forms of invocation, and the resolution of external references. These same computations are necessary to define the meaning of a module. If we are to separate the legality of a module from its meaning, we must retain the relevant information from the checking phase and not recompute it when defining the meaning. This implies that, as part of legality-checking, we must transform the original parse tree into a new (and simpler) form.

Some of the declarative information in a module is not needed when defining the meaning of the module. For example, much of the module heading exists solely for the purpose of type-checking. Further, the declarative information that *is* necessary for

defining the meaning is usually scattered throughout the module, and must be collected and put in a more useful form. For example, variable declarations do not cause any immediate action during evaluation, but variables local to a given scoping unit must be removed when that unit is exited. Thus at some point it is necessary to collect all of the variables declared in each scoping unit.

Since all of the declarative information is static and need only be processed once, we prefer to include all of the processing as part of legality-checking. Hence "legality-checking" involves not only checking the legality of a module, but also removing information irrelevant to evaluation, and putting information necessary to evaluation in its most usable form. However, not all of the transformations we wish to perform are mappings into a subset of the abstract syntax. Therefore we define a new set of domains, called the *transformed syntax*, for representing the modified parse trees. For each domain of the abstract syntax, we define a function that both checks an element of the domain and constructs a new element in the transformed syntax.

2.1.2 Compilation Environments

The legality of a program fragment generally depends on the specific context in which the fragment appears. Therefore, legality-checking is always done with respect to a particular *compilation environment*, or CE. Part of the CE is a user-supplied mapping from identifiers to abstractions and constants, used to resolve external references; although this mapping is called the CE in [Liskov78], we will use the term CE for the complete environment. In addition, the CE contains the current declarations of local identifiers, interface information derived from the module heading, a flag indicating if the body of a loop statement is being checked (to determine the legality of **break** and **continue** statements), and information about all exception handlers surrounding the current statement.

The CE also contains part of the CLU *library*, the repository of all information about abstractions. For each abstraction there is a *description unit* (DU) containing all system-maintained information about that abstraction. For our purposes there are only two

pieces of information a DU contains: an *interface specification* representing all information needed to type-check uses of the abstraction, and a list of *implementations*. Since this information can expand over time (DUs are initially empty) and we have only static mathematical objects to work with, a DU is represented simply by a unique id. The library is then represented by a pair of mappings, one from DUs to interface specifications, and one from DUs to implementations. The CE contains only the first mapping.

2.1.3 Interface Specifications

In general, before a module can be compiled, the library must contain an interface specification for every abstraction the module uses. For our definition to be complete then, we need to define what legal interface specifications are and how interfaces are entered into the library. Although a specific textual form for interfaces is not given in [Liskov78], the interface for an abstraction can be derived from any legal module implementing that abstraction. Therefore, rather than introducing a textual form for interfaces, we will view a request to compile a module as if it were a request to first derive the interface and (if necessary) enter the interface into the library, and then actually compile the module.

Thus we allow a module to be compiled even though the library does not contain the interface of the particular abstraction being implemented; the interface is installed as a side-effect of compilation. More generally, a group of modules can be compiled simultaneously even though interfaces do not exist for any of the corresponding abstractions.

With this view of compilation, legality-checking takes on the following form. First, interfaces are derived from the headings of the modules to be compiled. These interfaces must be identical (up to renaming of formal parameters) to the interfaces already in the library, if any. The derived interfaces are then installed temporarily in the library and the modules are checked in their entirety. The derived interfaces remain in the library if all of the modules are legal; otherwise they are removed.

Although it is fairly easy to derive and check the interface of almost any reasonable module, it is possible to construct nonsensical yet legal modules for which the process is not so easy. Unfortunately, it is extremely difficult to devise simple rules that make nonsensical modules illegal without also making some sensible modules illegal, and no such rules exist in CLU. Hence we must deal with a few problems.

In particular, certain kinds of cyclic and self references complicate the way interfaces must be derived. For example, to derive the interface of F in

```
P = proc () returns (A[false]); ... end P;
F = proc () returns (A[F = P]); ... end F;
```

one must evaluate the expression "F = P". This expression is an abbreviation for the invocation "T$equal(F, P)", where T is the type of F. If we take the stand that illegal expressions should never be evaluated, then to type-check "F = B" we must know the exact type of F. However, the type of F depends on the value of the very expression we are attempting to evaluate.

Although this is a bizarre example and would never occur in a real program, it is in fact legal. We can reason as follows. P and F are distinct procedures, and distinct procedures are never considered equal. Therefore the invocation evaluates to **false** if it is legal. But then P and F are of the same type so the invocation is actually legal. (The invocation would be illegal if P were written with "A[**true**]".)

As a slightly more reasonable example, consider a two-person game where players operate by different rules. One might try to represent alternating moves in the game as follows:

```
offense = cluster [t: type] is best_replies, ...     where t has ... ;
            ...
          best_replies = iter (x: cvt) yields (defense[t]); ... end best_replies;
          end offense;

defense = cluster [u: type] is best_replies, ...     where u has ... ;
            ...
          best_replies = iter (x: cvt) yields (offense[u]); ... end best_replies;
          end defense;
```

The interface of defense is needed to check the interface of offense (specifically to check

"defense[t]"), and vice versa (to check "offense[u]"). Initially we can assume that "defense[t]" and "offense[u]" are legal type specifications; then after the derived interfaces are installed we can actually check the legality of these constructs.

As these examples suggest, we can derive an interface and check its legality, but two passes are required to guarantee that no errors are missed. In the first pass certain constructs are simply assumed to be legal, and must be evaluated with incomplete type information. Then these constructs are rechecked when checking the entire module, this time with complete information, to ensure there are no inconsistencies.

In more detail, we begin legality-checking by installing one of two special interfaces for each abstraction being implemented. The special interface **type** is used when the implementing module is a cluster; every use of the abstraction is considered a legal type specification, regardless of the number and types of actual parameters supplied. In addition, every operation name qualified by such a type specification is considered legal. The special interface **routine** is used when the implementing module is a procedure or iterator; every use of the abstraction is considered a legal routine name, regardless of the number and types of actual parameters supplied.

We also define a special type, **routine**, to be used as the syntactic type of every routine name (and every operation name) that is assumed legal. (This does not mean that "routine" is a reserved word; a use of "routine" in a program will not refer to this special type.) The type **routine** is a "wild card" in that it is considered equal to (and hence includes and is included in) all procedure and iterator types. **Routine** is defined to have the same operations that procedure and iterator types have. Returning to the invocation "F = P" in the first example above, both F and P would be assumed to be legal names of syntactic type **routine**, so the invocation would expand to "**routine$equal**(F, P)" and then evaluate to **false**.

The special interfaces and the type **routine** are used only when deriving interfaces. The type **routine** is used only when an abstraction has a special interface, and the special interfaces disappear when the derived interfaces are installed.

2.2 Meaning

Our definition of the meaning of a legal program can be summarized as follows. The CLU *universe* is defined to be the set of all potentially existing *objects*. A particular set of objects, each object having a particular *individual state*, is termed a *universal state*. An *execution environment* is defined to be a universal state together with a set of *variable bindings* represented as a mapping from identifiers to objects. The meaning of an expression or statement is a function that takes an environment and produces a *result*, consisting of a *termination condition*, a list of zero or more objects, and a new environment. The meaning of a procedure generator or iterator generator is a function that takes a list of actual parameters, a list of arguments, and an environment, and produces a result. The meaning of parameters is given in terms of "textual" substitution. A non-parameterized routine is viewed as being a generator with an empty parameter list. The meaning of a cluster is a function that takes a list of actual cluster parameters and an operation name, and produces the meaning of that operation.

We now discuss these points in more detail.

2.2.1 Objects

Every object has an *identity*, distinct from the identity of all other objects, and a set of *properties*. Objects with only time-invariant properties are called *constant*; the properties associated with a constant object comprise that object's *value*. Objects with time-varying properties are called *mutable*; the current "values" of the properties of a mutable object are collectively termed the *state* of that object.

One time-invariant property associated with every object is membership in a type. Were it not for the necessity of defining the parameterized procedure **force**, which requires testing the type of an object, there would be no need to include any type information in our representation of objects. However, since this information must be kept for at least some objects, we choose to keep it for all objects. One component of every object is thus a *type descriptor*, a canonical name for the type of which the object is a

member.

The presence of a type descriptor in every object simplifies the task of making objects distinguishable. In particular, since every constant object has a unique value, the value can serve to represent the object without introducing explicit identities. For example, the constant objects obtained from the invocations

```
oneof[a, b: int]$make_a(3)
oneof[a, c: int]$make_a(3)
```

can (only) be distinguished by their types:

```
(tag: "a", val: (val: 3, type: int), type: oneof[a: int, b: int])
(tag: "a", val: (val: 3, type: int), type: oneof[a: int, c: int])
```

However, it is not sufficient to represent a mutable object by its state, as different mutable objects can have identical state. Rather, the identity of a mutable object is explicitly represented by a unique id.

Although we would like to represent a mutable object directly by its identity and state, this cannot be done. The problem is that we want to be able to change the state of an object without having to alter the representation of any other object. To illustrate, suppose we have two arrays, represented (approximately) as

```
(id: A, low: 1, elts: (), type: array[int])
(id: B, low: 1, elts: ( (id: A, low: 1, elts: (), type: array[int]) ),
                        type: array[array[int]])
```

The array B contains the array A as an element. Now suppose we change the low bound of A:

```
(id: A, low: 2, elts: (), type: array[int])
(id: B, low: 1, elts: ( (id: A, low: 1, elts: (), type: array[int]) ),
                        type: array[array[int]])
```

Since we are dealing with static mathematical values there is no sharing, and the change to A is not reflected in the "copy" of A contained in B.

Our solution to this problem is to represent a mutable object with just a unique id and a type descriptor. The above example would then look something like

(id: A, type: **array[int]**)
(id: B, type: **array[array[int]]**)

The association between mutable objects and the rest of their state is given by a mapping (or mappings) as part of the universal state. For example,

state(A) = (low: 2, elts: ())
state(B) = (low: 1, elts: ((id: A, type: **array[int]**)))

In fact, by considering constant objects as always in existence, the universal state can be represented simply as a list of mutable objects (i.e., unique id-type descriptor pairs) together with one or more state mappings.

In dealing with parameters, it will be convenient to treat actual parameters uniformly as objects. The only actual parameters not normally thought of as objects are types, but there is no difficulty in treating them as such. A type object is represented by a pair of type descriptors, one for the type itself and one for the type of the object as a whole. For example,

(val: **array[int]**, type: **type**)

2.2.2 Variables

We would like to model variables with a mapping from identifiers to objects. (In particular, we wish to avoid using the standard "store" semantics [Strachey73], where variables name memory locations that contain pointers to locations containing objects.) Because of the representation we have chosen for objects, there are no difficulties with this model; having several variables denote the same object is essentially the same as having several objects contain the same object.

Uninitialized variables are treated in the following manner. Initially all variables denote a unique "bad" object, which otherwise cannot be produced in a legal computation. Whenever a variable is referenced, the denoted object is checked to ensure that it is not the "bad" object. At the end of each scoping unit, all variables declared local to that unit are again made uninitialized by mapping them back to the "bad" object.

2.2.3 Termination and Exception Handling

A *continuation* is a function that defines the meaning of the "rest" of the program, starting from a particular point in the text. Continuations are normally used when non-sequential control flow is possible [Wadsworth73]. For example, continuations can be defined at every labelled point in a program; the meaning of "go to L" is then defined as the application of the continuation for L to the current environment. However, since only well-structured, forward transfers of control occur in CLU, we have chosen not to use continuations. Instead, we "tag" the outcome of every evaluation with a *termination condition.*

The basic idea is that one condition, **normal**, implies that control should proceed sequentially, while all other conditions imply a transfer of control. The equations of the definition are written so that results with non-**normal** termination conditions propagate through all intermediate evaluations until the appropriate point of control is reached. For example, the result of a **return** statement propagates to the procedure or iterator boundary and then changes to a result with a **normal** termination condition. Similarly, the result of a **signal** statement propagates to the procedure or iterator boundary and then changes to look like the result of an **exit** statement; the result then propagates to the appropriate handler in the caller.

For this technique to work well, it is necessary that expressions and statements evaluate to elements of the same domain, so that results can propagate freely. The result of such an evaluation therefore is defined to be a triple consisting of a termination condition, a list of zero or more objects, and an environment. Normally, expression evaluation produces a result with just one object and statement evaluation produces a result with no objects.

Since expression evaluation cannot alter any variable bindings (but statement evaluation can), one might wish to make the result of expression evaluation include just a universal state instead of a complete environment. Although this would obviate the fact that expression evaluation does not alter the variable bindings for subsequent evaluations, it would not obviate the fact that variable bindings for subexpression evaluations are not

affected. Since the proofs of these facts are trivial, but having two result forms would complicate the definition, we prefer to use a single form of result.

2.2.4 Iterators

There are basically two ways to view an iterator. The first way is that an iterator actually yields values to a **for** statement. In order to define iterators in this way, an iterator must actually yield an extra value, namely a continuation, so that the iterator can be resumed. In the second view a *closure*, consisting of the loop body and the variable bindings active for that body, is passed to the iterator. At each **yield** statement the loop body is evaluated with its active variable bindings to produce a new universal state and new variable bindings. The new bindings replace the previous bindings in the closure, and the iterator continues with its own active variable bindings and the new universal state. When the iterator terminates it returns the variable bindings produced by the last loop cycle, and the caller continues with those bindings.

We are not using continuations to define any other constructs of CLU, so it seems best to take the second view of iterators. Further, defining iterators in this fashion involves much less interaction with other clauses of the definition than would be the case with the first view, and works out much simpler overall.

The closure for a **for** statement is kept as a component of the environment. However, iterators can invoke other iterators, so the environment actually contains a stack of closures. The caller pushes on a closure before invoking the iterator. At each **yield** statement the iterator pops off the top closure, evaluates the loop body, and then pushes a new closure back on the stack. When the iterator terminates, the caller pops off the top closure.

Since the iterator controls evaluation of the loop body, the **for** statement automatically terminates when the iterator terminates. On the other hand, if the loop body executes a **return** or **signal** statement or terminates in an exceptional condition, this information must be propagated through the iterator and back to the caller. The information must be treated specially; for example, the result of a **return** statement executed in the loop body

should not be changed at the iterator boundary to look like a normal result, as would happen for a **return** statement executed in the iterator. Thus, special termination conditions are used in these cases.

2.2.5 Modules

Rather than distinguishing between parameterized and non-parameterized modules, we prefer to view every module as having a parameter list, though the list can be empty. The meaning of a procedure is then a function that takes a list of actual parameter objects (treating types as objects as previously discussed), a list of argument objects, and an execution environment, and produces a result (a termination condition, a list of objects, and a new environment). The meaning of parameters is given by a rewriting rule: the actual parameters are substituted for the formal parameters in the parse tree before the procedure body is evaluated. Hence there is no need for parameter bindings in the execution environment.

Because of the way we have defined iterators, they have the same functionality as procedures. The iterator and its caller modify the environment rather than explicitly passing extra values back and forth.

The meaning of a cluster is a function that takes a list of actual cluster parameters and an operation name, and produces the meaning of that operation. The actual parameters are substituted for the formal cluster parameters before the meaning of the operation is derived; the meaning of the operation is then defined as for any other procedure or iterator. The function produces the meanings of all routines in the cluster, including "hidden" operations; legality-checking ensures that only the cluster can directly name the hidden operations.

3. Theory and Notation

This chapter describes the most basic elements of the lattice-theoretic approach to the theory of computation, and introduces the notation used in the remainder of the thesis. Our goal here is to say just enough about the underlying theory for the language definition to make sense; many aspects will be glossed over or ignored. Motivation for and detailed discussion of the lattice-theoretic approach can be found in [Scott72].

3.1 Domains

A *domain* is a non-empty set of elements. (A domain must also have a partial ordering defined on its elements, but the ordering will not concern us here.) Every domain D has a distinguished element, $\perp_D$, called *bottom*. Bottom is used to represent the value of meaningless, illegal, nonterminating, or otherwise undefined computations. Legality-checking never involves such a computation, and legality-checking guarantees that bottom is produced only by nonterminating computations during program execution.

Domains are constructed in the following ways:

1) Given a countable set of elements, a *primitive* domain is formed by adjoining to the set a distinguished element to serve as bottom.

2) Given domains $D_1, D_2, \ldots, D_n$, the *product* domain

$$\times_{i=1}^{n} D_i = D_1 \times D_2 \times \ldots \times D_n$$

is defined to be the set of all n-tuples of the form

$$\langle d_1; d_2; \ldots; d_n \rangle$$

where each d_i is an element of D_i. A product domain is essentially a form of constant record. The element

$$\langle \perp_{D_1}; \perp_{D_2}; \ldots; \perp_{D_n} \rangle$$

is distinguished as bottom for the product domain. When all of the D_i are the same domain D, we write $\times_{i=1}^{n} D_i$ as D^n, the set of all lists of length n of elements of D. We treat D and D^1 as the same domain. We define D^0 to be the set $\{\perp_D, \langle\rangle\}$ for all domains D. The 0-tuple $\langle\rangle$ is written as "nil".

3) Given domains $D_1, D_2, ..., D_n$, the *sum* domain

$$\overset{n}{\underset{i=1}{+}} D_i = D_1 + D_2 + ... + D_n$$

is defined to be the set of all pairs of the form

$$\langle i; d_i \rangle$$

where d_i is a proper element (i.e., not bottom) of D_i. In addition, the sum domain contains a distinguished element as bottom. A sum domain is similar to a oneof type.

4) Given a domain D, the domain D^* of all lists of elements of D is defined by

$$D^* = \overset{\infty}{\underset{n=0}{+}} D^n = D^0 + D^1 + ... + D^n + ...$$

Similarly, the domain D^+ of all non-empty lists is defined by

$$D^+ = \overset{\infty}{\underset{n=1}{+}} D^n = D^1 + D^2 + ... + D^n + ...$$

5) Given domains D_1 and D_2, the *function* domain

$$D_1 \rightarrow D_2$$

is defined to be the set of all continuous functions from D_1 to D_2. We will not define continuity here (see [Scott72]), but simply note that all of the functions used in this thesis are continuous. The function

$$\lambda d_1.\, \perp_{D_2}$$

is distinguished as bottom for the function domain.

We define '×' to have precedence over '+' and '→', and '+' to have precedence over '→'. For example, we write

Expression × CE → Expr × CE

instead of

[Expression × CE] → [Expr × CE]

3.2 Notation

In general each domain is given a name, consisting of letters and underscores. As an exception, the domains $\overset{\infty}{\underset{n=0}{+}} D^n$ and $\overset{\infty}{\underset{n=1}{+}} D^n$ are named D^* and D^+, respectively, for any domain D. Domain names are written with at least the first letter capitalized.

An arbitrary element of a domain is written as the domain name all in lower case, with a digit added if necessary to distinguish it from other names. Thus,

d, d1, d2	are names for arbitrary elements of **D**
d^*, $d1^*$, $d2^*$	are names for arbitrary elements of D^*
d^+, $d1^+$, $d2^+$	are names for arbitrary elements of D^+

Names for specific functions appear in italics.

For any name D, the boldface name **D** stands for a two-element domain whose (only) proper element is written as the boldface name **d**. For example, the domain **True** consists of two elements, $\perp_{True}$ and **true**. In addition, a boldface operator (+, −, etc.) or punctuation symbol (⦇, ⟦, etc.) names, according to context, either a two-element domain or the proper element of that domain.

A typical element of a product domain $\overset{n}{\underset{i=1}{\times}} D_i$ is usually written as

$\langle d_1; d_2; \ldots; d_n \rangle$

However, when all of the components come from syntactic domains, boldface elements and spaces serve to separate the various components, and elements are written as

$⟦d_1\ d_2 \ldots d_n⟧$

For example, a typical element of the domain

Begin × Body × End

is written as

⟦**begin body end**⟧

The syntactic domains are those used in the abstract syntax, the transformed syntax, and interface specifications. They are defined in the next chapter.

We will often want to extract specific components from elements of product domains. Therefore, each component is given a name. Occasionally we will list these names explicitly in the domain definition, as in

vars: Idn^* × body: Unit × env: Env

When names are not explicitly given, they are taken to be the component domain names in lower case, except that * and + are changed to 's', and a digit is added if necessary to distinguish components of the same domain. For example, the component names of

$Obj^* \times Obj^* \times Env$

are

objs1 objs2 env

Given an expression representing an element of a domain, a component is selected by appending a period, followed by the component name. For example,

d.objs2

To "replace" a component named *name* of an element *elt* with a new value *val*, we use the notation

elt [val • name]

Note that this does not change *elt*, it merely stands for some other element:

$\langle d_1; \ldots; d_n \rangle [val \bullet d_i] = \langle d_1; \ldots; d_{i-1}; val; d_{i+1}; \ldots; d_n \rangle$

We define

let name = val *in* exp

to be

(λname. exp) val

That is, substitute *val* for every free occurrence of *name* in *exp*. Often only names for the components of *val* are needed, so we define both

let $\langle d_1; \ldots; d_n \rangle$ = val *in* exp

and

let $[\![d_1 \ldots d_n]\!]$ = val *in* exp

to be

let d_1 = val.d_1 *in* ... *let* d_n = val.d_n *in* exp

We build the naming of components into λ-expressions by defining

$\lambda(d_1, \ldots, d_n).$ exp

to be

$\lambda t.$ (*let* $\langle d_1; \ldots; d_n \rangle$ = t *in* exp)

and when applying a function to an element of a product domain we write $g(x_1, \ldots, x_n)$ instead of $g(\langle x_1; \ldots; x_n \rangle)$.

The form

$$g(x_1, ..., x_n) \equiv exp$$

means that g is defined to be

$$\lambda(x_1, ..., x_n).\ exp$$

When defining functions recursively, we use the form

$$g(x_1, ..., x_n) \equiv \underline{rec}\ h(g, x_1, ..., x_n)$$

This means that one should compute g as the least fixed point of

$$\lambda f.\ [\lambda(x_1, ..., x_n).\ h(f, x_1, ..., x_n)]$$

Least fixed points are defined in [Scott72].

3.3 Functions

Some of the functions defined below return boolean values. The domain Bool is defined to be the sum domain **True + False.**

A domain D is said to be *function-free* if no element of D is, or contains, a function. For every function-free domain D we define the function

Equal: $D \times D \rightarrow Bool$ written $d1 = d2$

as follows:

$d1 = d2$	$\equiv$	**true**	if d1 and d2 are the same proper element
$d1 = d2$	$\equiv$	**false**	if d1 and d2 are distinct proper elements
$d1 = d2$	$\equiv$	$\perp_{Bool}$	if d1 or d2 is $\perp_D$

For every sum domain $D = \sum_{i=1}^{n} A_i$ we define the functions

A_i-*injection*:	$A_i \rightarrow D$	written	a_i *in* D
A_i-*projection*:	$D \rightarrow A_i$	written	d *to* A_i
A_i-*inspection*:	$D \rightarrow Bool$	written	d *is* A_i

as follows:

a_i *in* D	$\equiv$	$<i; a_i>$	when a_i is a proper element
$\perp_{A_i}$ *in* D	$\equiv$	$\perp_D$	
$<i; a_i>$ *to* A_i	$\equiv$	a_i	
$<j; a_j>$ *to* A_i	$\equiv$	$\perp_{A_i}$	when $i \neq j$
$\perp_D$ *to* A_i	$\equiv$	$\perp_{A_i}$	
$<j; a_j>$ *is* A_i	$\equiv$	$i = j$	

$\perp_D$ *is* A_i = $\perp_{Bool}$

These functions are roughly the same as the oneof *make_*, *value_*, and *is_* operations in CLU.

We leave off the explicit injection when an element is in boldface, since the injection is obvious. In addition, we write **true** and **false** instead of (**true** *in* Bool) and (**false** *in* Bool).

For every domain D, we define the function

Cond: Bool × D × D → D written *if* bool *then* d1 *else* d2

as follows:

if **true** *then* d1 *else* d2 = d1
if **false** *then* d1 *else* d2 = d2
if $\perp_{Bool}$ *then* d1 *else* d2 = $\perp_D$

For every domain D^* we define the functions

Head:	$D^* \rightarrow D$	get first element
Tail:	$D^* \rightarrow D^*$	get all but first element
Cons:	$D \times D^* \rightarrow D^*$	add element at front
Append:	$D^* \times D \rightarrow D^*$	add element at end
Concat:	$D^* \times D^* \rightarrow D^*$	concatenate
Empty:	$D^* \rightarrow$ Bool	test for empty list
Same_size:	$D^* \times D^* \rightarrow$ Bool	test for equal length
Delist:	$[D^*]^* \rightarrow D^*$	remove one level of nesting

The definitions of all but the last of these should be obvious. If we let $X = D^*$, then *Delist* is defined as

Delist(x^*) = *rec if Empty*(x^*)
then nil *in* D^*
else Concat(*Head*(x^*), *Delist*(*Tail*(x^*)))

Thus, for example (leaving out a number of injections),

Delist(< <<1; 2>; <3; 4>>; <<5; 6; 7>; <8; 9>> >) = < <1; 2>; <3; 4>; <5; 6; 7>; <8; 9> >

Occasionaly it will be useful to treat elements of a domain D^* as if they represent unordered sets. For every function-free domain D^*, we define the functions

Element: $D \times D^* \rightarrow Bool$ written $d \in d^*$

Subset: $D^* \times D^* \rightarrow Bool$ written $d1^* \subset d2^*$

as follows:

$d \in d^* \equiv$ *rec if* $Empty(d^*)$
then **false**
else $d = Head(d^*) \vee d \in Tail(d^*)$

$d1^* \subset d2^* \equiv$ *rec if* $Empty(d1^*)$
then **true**
else $Head(d1^*) \in d2^* \wedge Tail(d1^*) \subset d2^*$

Note that $d1^* \subset d2^*$ simply checks that every element of $d1^*$ is an element of $d2^*$; duplicate elements are allowed in both lists.

Often when dealing with lists of elements from a product domain $D = \overset{n}{\underset{i=1}{\times}} A_i$ we will want to collect into a list the *ith* component of each element. For every domain D^*, where $D = \overset{n}{\underset{i=1}{\times}} A_i$, we define the functions

A_i*-collection*: $D^* \rightarrow A_i^*$ written $d^* \downarrow a_i$

as follows:

$d^* \downarrow a_i \equiv$ *rec if* $Empty(d^*)$
then nil *in* A_i^*
else $Cons(Head(d^*).a_i, Tail(d^*) \downarrow a_i)$

Sometimes we will want to "change" the value produced by a function for some particular argument. For example, we will use a function from identifiers to objects to represent variable bindings, and assignment to a variable will correspond to changing the object produced by the function for that variable. However, since we cannot change functions, we construct new ones instead. For every function domain $F = A \rightarrow B$, where A is a function-free domain, we define the function

Rebind: $F \times A \times B \rightarrow F$ written $f[a \leftarrow b]$

as follows:

```
f[a ← b] = λal. if al = a
                then b
                else f(al)
```

Occasionaly we will want to take an element of one sum domain and treat it as an element of some other sum domain. For example, we might want to treat an element of a domain D^+ as an element of D^*. For any two sum domains

$$D = [\overset{m}{\underset{i=1}{+}} B_i] + [\overset{n}{\underset{i=1}{+}} A_i] + [\overset{s}{\underset{i=1}{+}} C_i] \qquad T = [\overset{r}{\underset{i=1}{+}} E_i] + [\overset{n}{\underset{i=1}{+}} A_i] + [\overset{v}{\underset{i=1}{+}} F_i]$$

we define the function

Coercion: $D \to T$ written d *to_in* T

as follows:

```
d to_in T = if d is A1
              then d to A1 in T
            else if d is A2
              then d to A2 in T
            ...
            else if d is An
              then d to An in T
            else ⊥T
```

We leave off explicit coercions from D^+ to D^* for every domain D.

3.4 Further Notation

We define

res <term; obj*; env> = t *in* e

to be an abbreviation for

```
if t.term is Normal
  then let <term; obj*; env> = t in e
  else t
```

when *t* and *e* are (or produce) elements of the domain Result (defined in Chapter 6). It is this form that allows results with non-normal termination conditions to propagate through intermediate evaluations. Use of this form also has the desirable effect of allowing the result of a nonterminating computation (i.e., $\perp_{Result}$) to propagate through.

We abbreviate

```
let a1 = t1 in
  ...
let ai = ti in
res ai+1 = ti+1 in
  ...
res aj = tj in
let aj+1 = tj+1 in
  ...
let an = tn in e
```

to

```
let a1 = t1
  ...
  ai = ti
res ai+1 = ti+1
  ...
  aj = tj
let aj+1 = tj+1
  ...
  an = tn
in
e
```

To decompose elements of a sum domain $D = \sum_{i=1}^{n} A_i$ we define

```
case d
  elem aj of Aj
    then ej
    ...
  elem ak of Ak
    then ek
  else e
```

to be

```
if (d is Aj)
  then let aj = (d to Aj) in ej
  ...
else if (d is Ak)
  then let ak = (d to Ak) in ek
else e
```

and

```
case d
  elem aj of Aj
    then ej
    ...
  elem ak of Ak
    then ek
  end
```

to be

```
if (d is Aj)
  then let aj = (d to Aj) in ej
  ...
else if (d is Ak)
  then let ak = (d to Ak) in ek
else ⊥
```

There need not be an *elem* arm for every domain A_i. As for *let* expressions, the a_i can be $\langle c_1; \ldots; c_n \rangle$ and $[\![c_1 \ldots c_n]\!]$ forms as well as simple identifiers.

Many of the functions we will define take an element of a sum domain as their first argument, and perform different computations based on the particular form of that argument. In defining a function whose first argument belongs to a sum domain $D = \overset{n}{\underset{i=1}{+}} A_i$ we usually abbreviate

$g(d, \ldots) \equiv$ *case* d
elem a_1 *of* A_1
then e_1
...
elem a_n *of* A_n
then e_n
end

to

$g[\![a_1]\!](\ldots) \equiv e_1$
...
$g[\![a_n]\!](\ldots) \equiv e_n$

Similarly, we will sometimes treat a product domain $D = \overset{n}{\underset{i=1}{\times}} A_i$ as if it were a "unary sum domain" +D, and abbreviate

$g(d, \ldots) \equiv$ *let* $[\![a_1 \ldots a_n]\!] = d$
in
e

to

$g[\![a_1 \ldots a_n]\!](\ldots) \equiv e$

4. Syntactic Domains

This chapter defines all of the *syntactic* domains used in the thesis, domains whose elements either directly represent pieces of program text, or have a natural textual representation. First we present our abstract syntax for CLU, and then the transformed syntax. The relationship between these two sets of domains should be fairly obvious and will not be elaborated on here; the precise relationship is defined in the next chapter. Finally, we give the forms of interface specifications used in the library.

To improve the readability of these domain definitions, we introduce a BNF-like notation. All domain names appear in lower case, "x" symbols are omitted when forming product domains, and "|" symbols are used in place of "+" symbols when forming sum domains. This allows us to write

```
when_arm    ::= when name+ ( decl* ) : body
              | when name+ ( * ) : body
```

instead of

```
When_arm:     When x Name+ x ( x Decl* x ) x : x Body
            + When x Name+ x ( x * x ) x : x Body
```

This notation will be used only in this chapter.

4.1 Abstract Syntax

Two of the domains below merit special comment: Constant and Equate. These two domains contain extra alternatives that would not be found in a normal syntax. The extra alternatives represent the overlap that exists between the normal alternatives. In particular, the forms "idn" and "idn[constant+]" occur as both expressions and type_specs, and an idn can additionally occur as a type_set. For example, in the equate $x = y$ it is possible for y to name a type, a type_set, or an object of some built-in type. Deciding which domain such a form actually belongs in requires semantic information; therefore the decision is made as part of legality-checking, not parsing.

```
full_module ::= equate* module

module      ::= procedure
              | iterator
              | cluster

procedure   ::= idn = proc [ decl* ] ( decl* ) returns ( type_spec* )
                                                 signals ( cond_spec* ) where restriction*
                    equate*
                    statement*
                    end idn

iterator    ::= idn = iter [ decl* ] ( decl* ) yields ( type_spec* )
                                                 signals ( cond_spec* ) where restriction*
                    equate*
                    statement*
                    end idn

cluster     ::= idn = cluster [ decl* ] is idn+ where restriction*
                    equate+
                    routine+
                    end idn

decl        ::= idn+ : type_spec

cond_spec   ::= name ( type_spec* )

restriction ::= idn has oper_decl+
              | idn in type_set

type_set    ::= { idn | idn has oper_decl* equate* }
              | idn

oper_decl   ::= oper_name+ : type_spec

oper_name   ::= name [ constant+ ]
```

routine ::= procedure
| iterator

constant ::= expression
| idn
| idn [constant^{+}]
| type_spec

body ::= equate* statement*

equate ::= idn = constant
| idn = idn
| idn = type_set
| **rep** = type_spec

type_spec ::= **null**
| **bool**
| **int**
| **real**
| **char**
| **string**
| **any**
| **rep**
| **cvt**
| **type**
| **array** [type_spec]
| **record** [field_spec^{+}]
| **oneof** [field_spec^{+}]
| **proctype** (type_spec*) **returns** (type_spec*) **signals** (cond_spec*)
| **itertype** (type_spec*) **yields** (type_spec*) **signals** (cond_spec*)
| idn [constant^{+}]
| idn

field_spec ::= name^{+} : type_spec

```
statement   ::= decl
              | idn : type_spec := expression
              | decl+ := invocation
              | idn+ := invocation
              | idn+ := expression+
              | invocation
              | expression . name := expression
              | expression [ expression ] := expression
              | if expression then body
                  elseif_arm*
                  end
              | if expression then body
                  elseif_arm*
                  else body
                  end
              | while expression do body end
              | return ( expression* )
              | yield ( expression* )
              | signal name ( expression* )
              | exit name ( expression* )
              | break
              | continue
              | begin body end
              | tagcase expression
                  tag_arm+
                  end
              | tagcase expression
                  tag_arm+
                  others : body
                  end
              | for decl* in invocation do body end
              | for idn* in invocation do body end
              | statement except when_arm+ end
              | statement except when_arm*
                                others_arm
                                end
```

```
elseif_arm   ::= elseif expression then body

tag_arm      ::= tag name+ : body
               | tag name+ ( idn : type_spec ) : body

when_arm     ::= when name+ ( decl* ) : body
               | when name+ ( * ) : body

others_arm   ::= others : body
               | others ( idn : type_spec ) : body

expression   ::= nil
               | bool
               | int
               | real
               | char
               | string
               | type_spec $ name [ constant* ]
               | idn [ constant+ ]
               | idn
               | invocation
               | type_spec $ { field+ }
               | type_spec $ [ expression* ]
               | type_spec $ [ expression : expression* ]
               | force [ type_spec ]
               | up ( expression )
               | down ( expression )
               | expression . name
               | expression [ expression ]
               | ~ expression
               | - expression
               | expression bin_op expression
               | expression cand expression
               | expression cor expression

invocation   ::= expression ( expression* )

field        ::= name+ : expression
```

```
bool      ::= true
            | false

real      ::= int e int

char      ::= char int

string    ::= char*

bin_op    ::= **
            | //
            | /
            | *
            | ||
            | +
            | -
            | &
            | |
            | <
            | <=
            | =
            | >=
            | >
            | ~<
            | ~<=
            | ~=
            | ~>=
            | ~>
```

The domain Int is the primitive domain obtained from the mathematical integers. An element [[int1 e int2]] of Real represents the real number $int1*10^{int2}$. The parser produces exact values for literals and, in the case of reals, normalizes the values according to some scheme. For simplicity, the domains Name and Idn are both considered isomorphic to the domain String, although the parser guarantees that the actual elements used consist of lowercase letters, digits, and underscores (no leading digit).

4.2 Transformed Syntax

A domain name of the form "d_xxx" should be read "xxx descriptor".

```
mod         ::= du = mod_form

mod_form    ::= op_form
              | type_form

op_form     ::= routine [ idn* ] ( idn* ) unit end

unit        ::= stmt* local idn*

type_form   ::= cluster [ idn* ] oper* end

oper        ::= idn = op_form

d_type      ::= any
              | type
              | d_record
              | d_oneof
              | d_proc
              | d_iter
              | routine
              | d_mod
              | idn
              | bad

d_record    ::= record [ d_comp* ]

d_oneof     ::= oneof [ d_comp* ]

d_comp      ::= name : d_type

d_proc      ::= proctype ( d_type* ) returns ( d_type* ) signals ( d_cond* )

d_iter      ::= itertype ( d_type* ) yields ( d_type* ) signals ( d_cond* )
```

```
d_cond      ::= name ( d_type* )

d_mod       ::= du [ obj* ]

stmt        ::= none
              | idn* := expr*
              | idn* := invoke
              | invoke
              | if expr then unit
                  elseif*
                  else unit
                  end
              | while expr do unit end
              | return ( expr* )
              | yield ( expr* )
              | signal name ( expr* )
              | exit name ( expr* )
              | break
              | continue
              | begin unit end
              | tagcase expr
                  tag*
                  end
              | for idn* in invoke do unit end
              | stmt except catch* end

elseif      ::= elseif expr then unit

tag         ::= tag name* ( idn ) : unit
              | tag name* : unit
              | others : unit

catch       ::= when name* ( idn* ) : unit
              | when name* ( * ) : unit
              | others : unit
              | others ( idn ) : unit
```

```
expr        ::= obj
              | idn
              | invoke
              | d_type $ ( comp* )
              | d_type $ [ expr : expr* ]
              | up [ d_type ] ( expr )
              | down ( expr )
              | expr cand expr
              | expr cor expr

invoke      ::= expr ( expr* )

comp        ::= name : expr

obj         ::= val : d_type

val         ::= nil
              | bool
              | int
              | real
              | char
              | string
              | array
              | record
              | oneof
              | d_mod
              | d_oper
              | d_type
              | idn
              | obj
              | bad

oneof       ::= name : obj

d_oper      ::= d_type $ name [ obj* ]

tset        ::= idn has op_decl*
```

op_decl ::= name [obj*]: d_type

The domains Bool, Int, Real, Char, and String are defined in the previous section. Array and Record are unique id domains, and are isomorphic to the natural numbers. The domain DU is isomorphic to the domain Idn; a particular element of DU is written simply as du_{idn}, without an explicit injection into DU.

4.3 Interface Specifications

The domain DU_spec, representing interface specifications, is defined as follows:

```
du_spec    ::= r_spec
             | routine
             | t_spec
             | type
             | none

r_spec     ::= [ d_parm* ] : d_type

t_spec     ::= [ d_parm* ] : op_spec*

op_spec    ::= name [ d_parm* ] : d_type

d_parm     ::= idn : constraint

constraint ::= d_type
             | op_decl*
```

The domains D_type and Op_decl are defined in the previous section.

The domain R_spec is used for procedural and control abstractions; the d_type in an r_spec names a procedure or iterator type. T_spec is used for data abstractions; the op_specs represent the primitive operations. **Type** and **Routine** are used only when deriving interface specifications, as described in Chapter 2.

Formal parameters are described by an idn and a constraint. The idn is needed because it can appear elsewhere in the interface specification, and such instances have to be replaced by an actual parameter to type-check instantiations of the abstraction. The constraint associated with a type parameter is a list of op_decls representing the operations any actual parameter must have; the constraint for any other parameter is simply the declared type of that parameter.

To simplify our definition, the d_parm list in an op_spec contains both the cluster parameters and the operation parameters, in that order; the constraint associated with a cluster type parameter in this list contains the union of the restrictions imposed by the cluster and the operation.

Not all t_specs and r_specs represent legal interface specifications; only those produced by legality-checking are considered legal. Even the interface specifications of the built-in types and the **array** type generator can be produced this way. However, interface specifications for the **record**, **oneof**, **proctype**, and **itertype** type generators cannot be represented as elements of T_spec because of the unusual form of their parameters. These generators are handled specially, as described in section 5.3.

5. Legality

To minimize the number of forward references, we define legality-checking from the bottom up. We begin by defining the domain CE of compilation environments. Then, in order, we define the legality of type specifications, constants, expressions, equates, statements, interface specifications, and modules. In general, for each domain Xxx of the abstract syntax, we define a function:

C_xxx: Xxx × CE → Xxx' × CE

where Xxx' is the corresponding domain in the transformed syntax.

5.1 Environments

Compilation environments are defined as follows:

CE:	info: Info_map × specs: Spec_map × up: D_type × parms: Obj* × down: D_type × locals: Idn* × used: Idn* × results: D_type* × cvts: Bool* × sigs: Sig* × iter: Bool × cluster: Bool × loop: Bool × handles: Handle* × err: Bool
Info_map:	Idn → Info
Info:	D_type + Constraint + Op_decl* + DU + R_spec + Routine + Obj + Tset + None
Spec_map:	DU → DU_spec
Sig:	d_cond: D_cond × cvts: Bool*
Handle:	Name* + D_hand + Others
D_hand:	names: Name* × d_types: D_type*

The meanings of the various components of a CE are:

info – a mapping that holds information about the current bindings of all idns. The initial mapping is provided by the user; any idns bound in the initial mapping can (but need not) be used as external references. The meanings of the various alternatives of the domain Info are:

- d_type – the declared type of a variable.
- constraint – the constraint associated with a parameter. A non-type parameter always has this form of info; a type parameter only has this form of info after interface specifications are derived.
- op_decl* – a list for accumulating the restrictions on a type parameter, used only when deriving interface specifications. A type parameter with this form of info is allowed to have any and all operations.
- du – the DU bound to an external reference.
- r_spec – the interface specification of a cluster routine. When a cluster routine is referred to with an idn rather than a name qualified by the abstract type, the use of the idn is checked with respect to this r_spec, not with respect to the library. Although both forms of reference transform to the same value when legal, they are checked differently to allow unqualified references to hidden operations. (Hidden operations are not specified in the interface of the abstraction.)
- routine – used for cluster routines when deriving interface specifications. An idn with this form of info is considered a legal routine name, regardless of the number and types of parameters supplied.
- obj – the evaluated constant bound to an equated identifier or external reference. Note that types are included here as objects.
- tset – the evaluated type_set bound to an equated identifier or external reference.

none - the info for all undefined idns.

specs - the part of the library mapping DUs to their interface specifications.

up - the abstract type when checking a cluster, otherwise the bad type.

parms - the abstract type has the form "du[obj*]"; this component contains the list obj*.

down - the representation type when checking a cluster, otherwise the bad type.

locals - a list of all currently defined local idns. At the end of each scoping unit, all idns local to that unit are made undefined (i.e., their info is set to none), and thus cannot be used subsequently as external references.

used - a list of all idns seen so far in the full_module. Every time an idn is encountered, the idn is added to this list. An idn cannot be defined locally if it has been used as an external reference or if it is already defined. In both of these cases (but no others), the idn will have info other than none and already will be in the list of used idns.

results - the return or yield types of the routine being checked. If cvt was used as the type specification for a result, the representation type is used in this list.

cvts - a list of flags indicating which positions in the results list were declared with cvt.

sigs - for each exceptional condition listed in the signals clause of the routine being checked, this list contains an element describing the exception name, the types of results (with cvt replaced by the representation type), and flags indicating the positions declared with cvt. The list also contains an element for the *failure* exception.

iter - true when checking an iterator.

cluster - true when checking a cluster.

loop - true when checking the body of a loop statement.

handles - a list describing all exception handlers surrounding the current statement, in order from innermost to outtermost handler. The meanings of the three kinds of handles are:

name* - a list of exception names, used for when arms that discard results.

d_hand - a list of exception names and a (possibly empty) list of variable types, used for all other when arms.

others - used for both forms of others arms. For any given except statement, the handles for the when arms appear in the handles list before the others handle.

err - true if any errors have been detected.

Only certain info_maps can be provided legally by the user for use in the CE. The legal info_maps are those which can be generated with the function *New_info_map* using legal info_maps and legal spec_maps, but arbitrary equates, as arguments. As a basis, any info_map mapping idns solely to DUs (and none) is considered legal. Legal spec_maps are discussed in Section 5.8.

```
New_info_map(info_map, spec_map, equate*) =
    let  ce1 = Create_ce(info_map, spec_map)
         ce2 = C_equates(equate*, ce1)
     in
    if ce2.err
      then info_map
      else ce2.info
```

C_equates is defined in section 5.5.

```
Create_ce(info_map, spec_map) =
    <info_map; spec_map; bad in D_type; ⊥Obj*; bad in D_type; nil in Idn*;
     nil in Idn*; ⊥D_type*; ⊥Bool*; ⊥Sig*; ⊥Bool; false; false;
     nil in Handle*; false>
```

The bottoms indicate components that will be filled in before being used.

5.2 Type Specifications

The major functions defined in this section are:

C_type_spec: Type_spec × CE → D_type × CE
C_type_specs: Type_spec* × CE → D_type* × CE

Bad_type(ce) ≡ <bad *in* D_type; ce[true ⊕ err]>

Has_bad(d_type*) ≡ *rec if Empty*(d_type*)
then false
else *Head*(d_type*) *is* Bad ∨ *Has_bad*(*Tail*(d_type*))

C_type_specs(type_spec*, ce) ≡ *rec*
let <d_type; ce1> ≡ *C_type_spec*(*Head*(type_spec*), ce)
<d_type*; ce2> ≡ *C_type_specs*(*Tail*(type_spec*), ce1)
in
if Empty(type_spec*)
then <nil *in* D_type*; ce>
else <*Cons*(d_type, d_type*); ce2>

C_type_spec[[null]](ce) ≡ <[[du_{null} []]] *in* D_type; ce>

We abbreviate "[nil *in* Obj*]" to "[]".

C_type_spec[[bool]](ce) ≡ <[[du_{bool} []]] *in* D_type; ce>

C_type_spec[[int]](ce) ≡ <[[du_{int} []]] *in* D_type; ce>

C_type_spec[[real]](ce) ≡ <[[du_{real} []]] *in* D_type; ce>

C_type_spec[[char]](ce) ≡ <[[du_{char} []]] *in* D_type; ce>

C_type_spec[[string]](ce) ≡ <[[du_{string} []]] *in* D_type; ce>

C_type_spec[[any]](ce) ≡ <any *in* D_type; ce>

C_type_spec[[**rep**]](ce) = *if* ce.down *is* **Bad**
then *Bad_type*(ce)
else <ce.down; ce>

Rep cannot be used unless it has been defined in an equate.

C_type_spec[[**cvt**]](ce) = *Bad_type*(ce)

C_type_spec[[**type**]](ce) = *Bad_type*(ce)

Cvt and type can appear only as top-level type specifications in module headings, so we will check specially for them there and otherwise consider them illegal.

C_type_spec[[**array**[type_spec]]](ce) =
let <d_type; cel> = *C_type_spec*(type_spec, ce)
obj* = [[(d_type *in* Val): (type *in* D_type)]] *in* Obj*
in
if d_type *is* **Bad**
then *Bad_type*(cel)
else <[[du_{array} [obj*]]] *in* D_type; cel>

C_type_spec[[**record**[field_spec+]]](ce) =
let <d_comp*; cel> = *C_field_specs*(field_spec*, ce)
d_compl* = *Order*(d_comp*↓name, d_comp*)
in
if *Duplicates*(d_comp*↓name) ∨ *Has_bad*(d_comp*↓d_type)
then *Bad_type*(cel)
else <[[**record**[d_compl*]]] *in* D_type; cel>

The order and grouping of selectors in the text does not matter, but the selectors must be distinct.

```
C_type_spec[[oneof[field_spec+]]](ce) =
    let  <d_comp*; cel> = C_field_specs(field_spec+, ce)
         d_compl*       = Order(d_comp*↓name, d_comp*)
     in
    if  Duplicates(d_comp*↓name) ∨ Has_bad(d_comp*↓d_type)
      then  Bad_type(cel)
      else  <[[oneof[d_compl*]]] in D_type; cel>
```

The order and grouping of tags in the text does not matter, but the tags must be distinct.

Order: $\text{Name}^* \times D^* \to D^*$

Order is defined for all domains D. The names are permuted to be in increasing lexicographic order, the D-list is permuted in the same way, and the permuted D-list is returned. The two argument lists must be the same length.

```
C_field_specs(field_spec*, ce) = rec
    let  [[name+: type_spec]] = Head(field_spec*)
         <d_type; cel>        = C_type_spec(type_spec, ce)
         d_compl*             = Get_d_comps(name+, d_type)
         <d_comp2*; ce2>      = C_field_specs(Tail(field_specs*), cel)
     in
    if  Empty(field_spec*)
      then  <nil in D_comp*; ce>
      else  <Append(d_compl*, d_comp2*); ce2>

Get_d_comps(name*, d_type) = rec
    let  d_comp  = [[Head(name*): d_type]]
         d_comp* = Get_d_comps(Tail(name*), d_type)
     in
    if  Empty(name*)
      then  nil in D_comp*
      else  Cons(d_comp, d_comp*)
```

```
Duplicates(d*) = rec if Empty(d*)
                     then false
                     else Head(d*) ∈ Tail(d*) ∨ Duplicates(Tail(d*))
```

Duplicates is defined for all function-free domains D.

```
C_type_spec[[proctype (type_spec1*) returns (type_spec2*) signals (cond_spec*)]](ce) =
    let <d_type1*; ce1> = C_type_specs(type_spec1*, ce)
        <d_type2*; ce2> = C_type_specs(type_spec2*, ce1)
        <d_cond*; ce3>  = C_cond_specs(cond_spec*, ce2)
     in
    C_proc_type(d_type1*, d_type2*, d_cond*, ce3)

C_proc_type(d_type1*, d_type2*, d_cond*, ce) =
    let name*    = d_cond*↓name
        d_cond1* = Order(name*, d_cond*)
     in
    if Has_bad(d_type1*) ∨ Has_bad(d_type2*) ∨ Duplicates(name*) ∨
       ("failure" in Name) ∈ name* ∨ Has_bad(Delist(d_cond*↓d_types))
       then Bad_type(ce)
       else <[[proctype (d_type1*) returns (d_type2*) signals (d_cond1*)]] in D_type;
             ce>
```

The order in which signals are listed in the text is unimportant. A given name can only be used once as a signal name, and "failure" cannot be specified explicitly.

```
C_type_spec[[itertype (type_spec1*) yields (type_spec2*) signals (cond_spec*)]](ce) =
    let <d_type1*; ce1> = C_type_specs(type_spec1*, ce)
        <d_type2*; ce2> = C_type_specs(type_spec2*, ce1)
        <d_cond*; ce3>  = C_cond_specs(cond_spec*, ce2)
     in
    C_iter_type(d_type1*, d_type2*, d_cond*, ce3)
```

```
C_iter_type(d_type1*, d_type2*, d_cond*, ce) =
    let  name*    = d_cond*↓name
         d_cond1* = Order(name*, d_cond*)
     in
    if  Has_bad(d_type1*) ∨ Has_bad(d_type2*) ∨ Duplicates(name*) ∨
        ("failure" in Name) ∈ name* ∨ Has_bad(Delist(d_cond*↓d_types))
      then  Bad_type(ce)
      else  <[[itertype (d_type1*) yields (d_type2*) signals (d_cond1*)]] in D_type;
             ce>
```

The order in which signals are listed in the text is unimportant. A given name can only be used once as a signal name, and "failure" cannot be specified explicitly.

```
C_cond_specs(cond_spec*, ce) = rec
    let  [[name(type_spec*)]]  = Head(cond_spec*)
         <d_type*; ce1>        = C_type_specs(type_spec*, ce)
         d_cond                = [[name(d_type*)]]
         <d_cond*; ce2>        = C_cond_specs(Tail(cond_spec*), ce1)
     in
    if  Empty(cond_spec*)
      then  <nil in D_cond*; ce>
      else  <Cons(d_cond, d_cond*); ce2>
```

```
C_type_spec[[idn[constant+]]](ce) =
    let  <obj; ce1> = C_constant([[idn[constant+]]] in Constant, ce)
     in
    if  obj.d_type is Type
      then  <obj.val to D_type; ce1>
      else  Bad_type(ce1)
```

Since this construct can denote various things, including types, when used as a constant, we simply check the construct as an arbitrary constant and then make sure the resulting object is a type. In this way the basic checking is centralized in one place.

```
C_type_spec[[idn]](ce) ≡ let  <obj; cel> = C_constant(idn in Constant, ce)
                          in
                         if  obj.d_type is Type
                            then  <obj.val to D_type;  cel>
                            else  Bad_type(cel)
```

Again, we check the construct as a constant and then make sure the resulting object is a type.

5.3 Constants

The major functions defined in this section are:

C_constant:	Constant × CE → Obj × CE
C_constants:	Constant* × CE → Obj* × CE
Get_op_type:	D_oper × CE → D_type
Subst:	Obj* × Idn* × D → D

Get_op_type is used to check if an operation name is legal, and returns the type of the operation. *Subst* is used to substitute actual parameters for formal parameters, and is defined for every domain D.

```
Bad_obj(ce) = <[[(bad in Val): (bad in D_type)]]; ce[true ← err]>

C_constants(constant*, ce) = rec let <obj; ce1> = C_constant(Head(constant*), ce)
                                 <obj*; ce2> = C_constants(Tail(constant*), ce1)
                              in
                             if Empty(constant*)
                               then <nil in Obj*; ce>
                               else <Cons(obj, obj*); ce2>

C_constant[[expression]](ce) =
    let <expr; ce1> = C_expression(expression, ce)
     in
    if Const_expr(expr)
      then let <term; obj*; env> = E_expr(expr, Primitive_env())
            in
           if term is Normal
             then <obj* to Obj; ce1>
             else Bad_obj(ce1)
      else Bad_obj(ce1)
```

For an expression to be legal as a constant, it must be a legal expression and a legal form of constant expression, and it must evaluate normally. Note that a legal expression only produces one object in the normal case, so "obj* *to* Obj" cannot fail. *Primitive_env* returns an execution environment in which (only) the built-in types and type generators, the type routine, and the procedure generator

force have implementations. *Primitive_env*, Term, and Env are defined in Chapter 6.

```
C_constant[[type_spec]](ce) = let <d_type; cel> = C_type_spec(type_spec, ce)
                               in
                              <[[(d_type in Val): (type in D_type)]]; cel>

C_constant[[idn]](ce) =
    let cel = ce[Cons(idn, ce.used) ● used]
     in
    case ce.info(idn)
      elem constraint of Constraint
        then case constraint
               elem d_type of D_type
                 then <[[(idn in Val): d_type]]; cel>
               else <[[(idn in D_type in Val): (type in D_type)]]; cel>
      elem op_decl* of Op_decl*
        then <[[(idn in D_type in Val): (type in D_type)]]; cel>
      elem du of DU
        then C_du_parms(du, nil in Obj*, cel)
      elem [[[d_parm*]: d_type]] of R_spec
        then let name = Make_name(idn)
              in
             C_op_parms([[name[d_parm*]: d_type]], nil in Obj*, cel)
      elem routine of Routine
        then let name = Make_name(idn)
                  d_oper = [[ce.up$name[]]]
              in
             <[[(d_oper in Val): (routine in D_type)]]; cel>
      elem obj of Obj
        then <obj; cel>
      else Bad_obj(cel)
```

To be a legal constant, an idn must either be a formal parameter, or else name a non-parameterized abstraction, a cluster routine, or an evaluated constant.

Make_name: Idn → Name

Make_name returns the name corresponding to the given idn. (Idn and Name are isomorphic domains.)

```
C_constant[[idn[constant+]]](ce) =
    let  cel          = ce[Cons(idn, ce.used) • used]
         <obj*; ce2>  = C_constants(constant+, cel)
     in
    case ce.info(idn)
      elem du of DU
        then C_du_parms(du, obj*, ce2)
      elem [[d_parm*]: d_type] of R_spec
        then let name = Make_name(idn)
                in
             C_op_parms([[name[d_parm*]: d_type]], obj*, ce2)
      elem routine of Routine
        then let name   = Make_name(idn)
                  d_oper = [[ce.up$name[obj*]]]
             in
             <[[(d_oper in Val): (routine in D_type)]]; cel>
      elem obj of Obj
        then C_constant([[idn[constant+]]] in Expression in Constant, ce)
      else Bad_obj(ce2)
```

The idn must name a parameterized abstraction, a cluster routine, or an evaluated constant. If the idn names an object (evaluated constant), then this construct can be legal only if it is an abbreviation for an invocation of a "fetch" operation. Rather than checking the invocation explicitly here, we simply check the construct as a constant expression.

```
C_du_parms(du, obj*, ce) =
    let d_mod = [[du[obj*]]]
     in
    case ce.specs(du)
      elem [[[d_parm*]: d_type]] of R_spec
        then if C_parms(obj*, d_parm*, ce)
                then let d_type1 = Subst(obj*, d_parm*↓idn, d_type)
                      in
                     <[[(d_mod in Val): d_type1]], ce>
                else Bad_obj(ce)
      elem routine of Routine
        then <[[(d_mod in Val): (routine in D_type)]], ce>
      elem [[[d_parm*]: op_spec*]] of T_spec
        then if C_parms(obj*, d_parm*, ce)
                then <[[(d_mod in D_type in Val): (type in D_type)]], ce>
                else Bad_obj(ce)
      elem type of Type
        then <[[(d_mod in D_type in Val): (type in D_type)]], ce>
      else Bad_obj(ce)
```

If the DU is for a procedural or control abstraction and the parameters are legal (or assumed legal), a routine object is returned. If the DU is for a data abstraction and the parameters are legal (or assumed legal), a type object is returned. Otherwise, the reference is illegal.

```
C_op_parms[[name[d_parm*]: d_type]](obj*, ce) =
    let obj1* = Concat(ce.parms, obj*)
     in
    if C_parms(obj1*, d_parm*, ce)
      then let d_type1 = Subst(obj1*, d_parm*↓idn, d_type)
               d_oper = [[ce.up$name[obj*]]]
           in
          <[[(d_oper in Val): d_type1]], ce>
      else Bad_obj(ce)
```

The cluster parameters are added in for type-checking but they are not included in the d_oper's actual parameter list.

```
C_parms(obj*, d_parm*, ce) ≡
    if obj*↓d_type = Parm_types(d_parm*)
      then let constraint* = Subst(obj*, d_parm*↓idn, d_parm*↓constraint)
             in
           C_constraints(obj*, constraint*, ce)
      else false
```

The types of the actual parameters must match exactly the types of the formal parameters, and all restrictions on type parameters must be satisfied.

```
Parm_types(d_parm*) ≡ rec if Empty(d_parm*)
                           then nil in D_type*
                           else let d_type* = Parm_types(Tail(d_parm*))
                                  in
                                case Head(d_parm*).constraint
                                  elem d_type of D_type
                                    then Cons(d_type, d_type*)
                                  else Cons(type in D_type, d_type*)

C_constraints(obj*, constraint*, ce) ≡ rec
    if Empty(obj*)
      then true
    else if C_constraints(Tail(obj*), Tail(constraint*), ce)
      then case Head(constraint*)
             elem op_decl* of Op_decl*
               then let ⟦val: d_type⟧ = Head(obj*)
                      in
                    C_op_decls(val to D_type, op_decl*, ce)
             else true
    else false
```

C_parms checks that the actual parameters are of the correct type, so if the constraint is an op_decl list then the corresponding actual parameter is guaranteed to be a type object.

```
C_op_decls(d_type, op_decl*, ce) = rec
    if Empty(op_decl*)
       then true
       else let [[name[obj*]: d_type1]] = Head(op_decl*)
                d_type2                 = Get_op_type([[d_type$name[obj*]]], ce)
            in
            Includes(d_type2, d_type1) ∧ C_op_decls(d_type, Tail(op_decl*), ce)
```

We use *Includes* instead of strict equality because *Get_op_type* can return the type routine (if interface specifications are being derived).

```
Get_op_type[[d_type$name[obj*]]](ce) =
    case d_type
      elem [[record[d_comp*]]] of D_record
        then let op_spec* = Record_op_specs(d_type to D_record)
                  objl*   = Make_objs(d_comp*↓d_type)
             in
             C_op_type(name, Concat(objl*, obj*), op_spec*, ce)
      elem [[oneof[d_comp*]]] of D_oneof
        then let op_spec* = Oneof_op_specs(d_type to D_oneof)
                  objl*   = Make_objs(d_comp*↓d_type)
             in
             C_op_type(name, Concat(objl*, obj*), op_spec*, ce)
      elem d_proc of D_proc
        then C_op_type(name, obj*, Proctype_op_specs(d_proc), ce)
      elem d_iter of D_iter
        then C_op_type(name, obj*, Itertype_op_specs(d_iter), ce)
      elem routine of Routine
        then C_op_type(name, obj*, Routine_op_specs(), ce)
      elem [[du[objl*]]] of D_mod
        then case ce.specs(du)
                  elem [[d_parm*]: op_spec*]] of T_spec
                    then C_op_type(name, Concat(objl*, obj*), op_spec*, ce)
                  else routine in D_type
      elem idn of Idn
        then case ce.info(idn)
                  elem constraint of Constraint
                    then C_parm_op(name, obj*, constraint to Op_decl*)
                  else routine in D_type
      else bad in D_type

Record_op_specs:   D_record → Op_spec*
Oneof_op_specs:    D_oneof → Op_spec*
Proctype_op_specs: D_proc → Op_spec*
Itertype_op_specs: D_iter → Op_spec*
```

Although interface specifications for the **record**, **oneof**, **proctype**, and **itertype** type generators cannot be represented as elements of T_spec, interface specifications for the operations of any particular instantiation can be represented

as elements of Op_spec, as defined in Chapter 6.

Routine_op_specs: → Op_spec*

Routine_op_specs returns the op_specs for the special type routine, as defined in Chapter 6.

```
Make_objs(d_type*) = rec if Empty(d_type*)
                         then nil in Obj*
                         else let d_type = Head(d_type*)
                                  obj    = [[d_type in Val]: [type in D_type]]
                              in
                              Cons(obj, Make_objs(Tail(d_type*)))

C_op_type(name, obj*, op_spec*, ce) = rec
    let [[name][d_parm*]: d_type]] = Head(op_spec*)
     in
    if Empty(op_spec*)
       then bad in D_type
    else if name = name1
       then if C_parms(obj*, d_parm*, ce)
               then Subst(obj*, d_parm*↓idn, d_type)
               else bad in D_type
    else C_op_type(name, obj*, Tail(op_spec*), ce)

C_parm_op(name, obj*, op_decl*) = rec let [[name][obj1*]: d_type]] = Head(op_decl*)
                                      in
                                  if Empty(op_decl*)
                                     then bad in D_type
                                  else if name = name1 ∧ obj* = obj1*
                                     then d_type
                                  else C_parm_op(name, obj*, Tail(op_decl*))

Const_exprs(expr*) = rec if Empty(expr*)
                         then true
                         else Const_expr(Head(expr*)) ∧ Const_exprs(Tail(expr*))
```

Const_expr[[obj]] = ¬(obj.val *is* Idn)

Parameters are allowed only as top-level constants; they cannot be used as arguments in invocations of constant expressions.

Const_expr[[idn]] = **false**

Const_expr[[d_type$(comp*)]] = *Const_expr*[[d_type$[expr: expr*]]] = **false**

Const_expr[[up[d_type](expr)]] = *Const_expr*[[down(expr)]] = **false**

Const_expr[[expr1 **cand** expr2]] = *Const_expr*[[expr1 **cor** expr2]] = **false**

```
Const_expr[[expr(expr*)]] =
    if Const_exprs(expr*)
      then case expr
             elem [[val: d_type]] of Obj
               then case val
                      elem [[d_type1$name[obj*]]] of D_oper
                        then Const_type(d_type1) ∧
                             Const_types(Return_types(d_type))
                      else false
             else false
      else false

Return_types(d_type) =
    case d_type
      elem [[proctype (d_type1*) returns (d_type2*) signals (d_cond*)]] of D_proc
        then d_type2*
      end

Const_types(d_type*) = rec if Empty(d_type*)
                           then true
                           else Const_type(Head(d_type*)) ∧
                                Const_types(Tail(d_type*))
```

Const_type[[any]] = *Const_type*[[type]] = *Const_type*[[bad]] = **false**

Const_type[[idn]] = *Const_type*[[d_record]] = **false**

Const_type[[d_oneof]] = true

Const_type[[d_proc]] = *Const_type*[[d_iter]] = *Const_type*[[routine]] = true

Const_type[[d_mod]] = *let* [[du[obj*]]] = d_mod
in
du ∈ <du_{null} ; du_{bool} ; du_{int} ; du_{real} ; du_{char} ; du_{string} >

Subst: $Obj^* \times Idn^* \times D \rightarrow D$

Subst is used to substitute actual parameters for formal parameters, and is defined for every domain D. *Subst*(obj*, idn*, d) performs a simultaneous substitution on its third argument as follows. Let obj be an element of obj* and let idn be the corresponding element of idn* (the two lists must be of equal length). Then *Subst* replaces all occurrences of

idn *in* D_type *in* Val

and

idn *in* Val

with

obj.val

We need to substitute for two different forms because formal type parameters appear differently than other formal parameters. However, no idn will occur in both forms.

5.4 Expressions

The major functions defined in this section are:

C_expression:	Expression × CE → Expr × CE
C_expressions:	Expression* × CE → Expr* × CE
C_invocation:	Invocation × CE → Expr × CE
Type_of:	Expr × CE → D_type
Types_of:	Expr* × CE → D_type*
Includes:	D_type × D_type → Bool
Include:	D_type* × D_type* → Bool

Type_of and *Types_of* are used to obtain the syntactic types of expressions. *Includes* and *Include* define the type inclusion rule.

```
Bad_expr(ce) ≡ <[[(bad in Val): (bad in D_type)]] in Expr; ce[true ● err]>

C_expressions(expression*, ce) ≡ rec
    let <expr; ce1> = C_expression(Head(expression*), ce)
        <expr*; ce2> = C_expressions(Tail(expression*), ce1)
     in
    if Empty(expression*)
       then <nil in Expr*; ce>
       else <Cons(expr, expr*); ce2>

C_expression[[nil]](ce) ≡ let d_type = [[du_null []]] in D_type
                          in
                          <[[(nil in Val): d_type]] in Expr; ce>

C_expression[[bool]](ce) ≡ let d_type = [[du_bool []]] in D_type
                           in
                           <[[(bool in Val): d_type]] in Expr; ce>

C_expression[[int]](ce) ≡ if L_int(int)
                          then let d_type = [[du_int []]] in D_type
                               in
                               <(int in Val): d_type]] in Expr; ce>
                          else Bad_expr(ce)
```

```
C_expression[[real]](ce) = if L_real(real)
                           then let d_type = [[du_real []]] in D_type
                                    reall  = Approx(real)
                                in
                                <(reall in Val): d_type] in Expr; ce>
                           else Bad_expr(ce)

C_expression[[char]](ce) = if L_char(char)
                           then let d_type = [[du_char []]] in D_type
                                in
                                <(char in Val): d_type] in Expr; ce>
                           else Bad_expr(ce)

C_expression[[string]](ce) = if L_string(string)
                             then let d_type = [[du_string []]] in D_type
                                  in
                                  <[[(string in Val): d_type]] in Expr; ce>
                             else Bad_expr(ce)
```

The parser produces elements of the domains Int, Real, Char, and String that correspond exactly to the literals used in the text. However, an implementation can impose certain restrictions as to which elements are actually legal and, in the case of reals, need only provide approximations to the requested values. These restrictions are embodied in the functions *L_int*, *L_real*, *L_char*, *L_string*, and *Approx*, as defined in Chapter 6.

```
C_expression[[type_spec$name[constant* ]]](ce) =
    let <d_type; ce1> = C_type_spec(type_spec, ce)
        <obj*; ce2>   = C_constants(constant*, ce1)
     in
    C_d_oper([[d_type$name[obj* ]]], ce2)

C_d_oper(d_oper, ce) = let d_type = Get_op_type(d_oper, ce)
                       in
                       if d_type is Bad
                         then Bad_expr(ce)
                         else <[[(d_oper in Val): d_type]] in Expr; ce>
```

```
C_expression[[idn[constant+]]](ce) =
    let info = ce.info(idn)
     in
    if ¬(info is D_type ∨ info is Obj)
       then let <obj; cel> = C_constant([[idn[constant+]]] in Constant, ce)
             in
            if obj.d_type is Type
               then Bad_expr(cel)
               else <obj in Expr; cel>
    else if constant+ is Constant
       then let expression1 = idn in Expression
             in
            case constant+ to Constant
              elem expression2 of Expression
                then C_expression([[expression1[expression2]]] in Expression, ce)
              elem type_spec of Type_spec
                then Bad_expr(ce)
              else let expression2 = constant+ to Constant to_in Expression
                    in
                   C_expression([[expression1[expression2]]] in Expression, ce)
    else Bad_expr(ce)
```

If the idn is not a variable and does not name an object, the construct is checked as a constant expression. Otherwise the construct is only legal if it is an abbreviation for an invocation of a "fetch" operation, in which case there can be only one constant and that constant cannot be a type_spec.

```
C_expression[[idn]](ce) = if ce.info(idn) is D_type
                          then <idn in Expr; ce>
                          else let <obj; cel> = C_constant(idn in Constant, ce)
                                in
                               if obj.d_type is Type
                                  then Bad_expr(cel)
                                  else <obj in Expr; cel>
```

The idn must be either a variable, a non-type parameter, or a constant expression to be legal.

```
C_expression[[invocation]](ce) = let <expr; cel> = C_invocation(invocation, ce)
                                 in
                                 case expr
                                   elem invoke of Invoke
                                     then if Result_types(invoke, cel) is D_type
                                            then <expr; cel>
                                            else Bad_expr(cel)
                                     else Bad_expr(cel)
```

An invocation can return only one object when used as an expression.

```
C_invocation[[expression(expression*)]](ce) =
    let <expr; cel>   = C_expression(expression, ce)
        <expr*; ce2> = C_expressions(expression*, cel)
     in
    C_call(expr, expr*, ce2)

C_call(expr, expr*, ce) =
    let d_type* = Types_of(expr*, ce)
     in
    case Type_of(expr, ce)
      elem [[proctype (d_type1*) returns (d_type2*) signals (d_cond*)]] of D_proc
        then if Include(d_type1*, d_type*) ∧ C_d_conds(d_cond*, ce.handles)
               then <[[expr(expr*)]] in Expr; ce>
               else Bad_expr(ce)
      else Bad_expr(ce)
```

For an invocation to be legal the arguments must be of the correct types, and all exceptional conditions that can be raised must be legal with respect to the surrounding handlers.

```
C_d_conds(d_cond*, handle*) = rec if Empty(d_cond*)
                                  then true
                                  else C_d_cond(Head(d_cond*), handle*) ∧
                                       C_d_conds(Tail(d_cond*), handle*)
```

```
C_d_cond(d_cond, handle*) = rec
    if Empty(handle*)
      then true
      else case Head(handle*)
             elem name* of Name*
               then d_cond.name ∈ name* ∨ C_d_cond(d_cond, Tail(handle*))
             elem <name*; d_type*> of D_hand
               then if d_cond.name ∈ name*
                      then d_cond.d_types = d_type*
                      else C_d_cond(d_cond, Tail(handle*))
             elem others of Others
               then true
             end
```

A handler need not exist for every exceptional condition raised by an invocation. If a handler does not throw results away then the types of the results must equal the types of the receiving variables.

```
C_expression[[type_spec${field+}]](ce) =
    let <d_type; ce1> = C_type_spec(type_spec, ce)
        <comp*; ce2> = C_fields(field+, ce1)
        comp1*       = Order(comp*↓name, comp*)
        name*        = comp1*↓name
        d_type*      = Types_of(comp1*↓expr, ce2)
     in
    case d_type
      elem [[record[d_comp*]]] of D_record
        then if d_comp*↓name = name* ∧ Include(d_comp*↓d_type, d_type*)
               then <[[d_type${comp*}]] in Expr; ce2>
               else Bad_expr(ce2)
      else Bad_expr(ce2)
```

The order and grouping of components in the text does not matter but every component must be initialized with an object of the proper type. The components are reordered only for type-checking; they appear in the transformed expression in the order in which they were written.

```
C_fields(field*, ce) = rec
    let [[name+: expression]] = Head(field*)
        <expr; ce1>           = C_expression(expression, ce)
        comp1*                = Get_comps(name+, expr)
        <comp2*; ce2>         = C_fields(Tail(field*), ce1)
     in
    if Empty(field*)
       then <nil in Comp*; ce>
       else <Concat(comp1*, comp2*); ce2>

Get_comps(name*, expr) = rec let comp  = [[Head(name*): expr]]
                                 comp* = Get_comps(Tail(name*), expr)
                             in
                            if Empty(name*)
                              then nil in Comp*
                              else Cons(comp, comp*)

C_expression[[type_spec$[[expression*]]](ce) =
        C_expression[[type_spec$[[(1 in Int in Expression): expression*]]](ce)

C_expression[[type_spec$[[expression: expression*]]](ce) =
    let <d_type; ce1> = C_type_spec(type_spec, ce)
        <expr; ce2>   = C_expression(expression, ce1)
        <expr*; ce3>  = C_expressions(expression*, ce2)
        d_type*       = Types_of(expr*, ce3)
     in
    case d_type
      elem [[du[obj*]]] of D_mod
        then if du = du_array
                then let obj = obj* to Obj
                      in
                     if Integer(expr, ce3) ∧ Includes_all(obj.val to D_type, d_type*)
                       then <[[d_type$[[expr: expr*]]] in Expr; ce3>
                       else Bad_expr(ce3)
                else Bad_expr(ce3)
        else Bad_expr(ce3)
```

```
C_expression[[force[type_spec]]](ce) =
    let  <d_type; cel>  = C_type_spec(type_spec, ce)
         obj*           = [[(d_type in Val): (type in D_type)]] in Obj*
         d_mod          = [[du_force [obj*]]]
         d_type1*       = any in D_type in D_type*
         d_type2*       = d_type in D_type*
         d_cond*        = [["wrong_type" in Name)(nil in D_type*)]] in D_cond*
         d_proc         = [[proctype (d_type1*) returns (d_type2*) signals (d_cond*)]]
     in
    if d_type is Bad
       then  Bad_expr(cel)
       else  <[[(d_mod in Val): (d_proc in D_type)]]; cel>

C_expression[[up(expression)]](ce) = let  <expr; cel> = C_expression(expression, ce)
                                          d_type      = Type_of(expr, cel)
                                     in
                                     if ce.cluster ∧ Includes(ce.down, d_type)
                                       then  <[[up[ce.up](expr)]] in Expr; cel>
                                       else  Bad_expr(cel)
```

We explicitly include the abstract type in the transformed expression because it is needed to define evaluation.

```
C_expression[[down(expression)]](ce) = let  <expr; cel> = C_expression(expression, ce)
                                            d_type      = Type_of(expr, cel)
                                       in
                                       if ce.cluster ∧ d_type = ce.up
                                         then  <[[down(expr)]] in Expr; cel>
                                         else  Bad_expr(cel)

C_expression[[expression.name]](ce) =
        C_op_call(Add_get_(name), expression in Expression+, ce)
```

*Add_get_*: Name → Name

*Add_get_* simply adds "get_" to the beginning of the name.

```
C_expression[[expression1[[expression2]]]](ce) =
    let expression+ = <expression1; expression2> in Expression+
     in
    C_op_call("fetch" in Name, expression+, ce)

C_expression[[~ expression]](ce) =
        C_op_call("not" in Name, expression in Expression+, ce)

C_expression[[- expression]](ce) =
        C_op_call("minus" in Name, expression in Expression+, ce)

C_expression[[expression1 bin_op expression2]](ce) =
    if bin_op = Base_op(bin_op)
       then let expression+ = <expression1; expression2> in Expression+
             in
            C_op_call(Op_name(bin_op), expression+, ce)
       else let bin_op1      = Base_op(bin_op)
                expression3 = [[expression1 bin_op1 expression2]] in Expression
             in
            C_expression([[~ expression3]] in Expression, ce)
```

Op_name[[**]] = "power" in Name		Base_op[[**]] = [[**]] in Bin_op	
Op_name[[//]] = "mod" in Name		Base_op[[//]] = [[//]] in Bin_op	
Op_name[[/]] = "div" in Name		Base_op[[/]] = [[/]] in Bin_op	
Op_name[[*]] = "mul" in Name		Base_op[[*]] = [[*]] in Bin_op	
Op_name[[||]] = "concat" in Name		Base_op[[||]] = [[||]] in Bin_op	
Op_name[[+]] = "add" in Name		Base_op[[+]] = [[+]] in Bin_op	
Op_name[[-]] = "sub" in Name		Base_op[[-]] = [[-]] in Bin_op	
Op_name[[<]] = "lt" in Name		Base_op[[<]] = [[<]] in Bin_op	
		Base_op[[~<]] = [[<]] in Bin_op	
Op_name[[<=]] = "le" in Name		Base_op[[<=]] = [[<=]] in Bin_op	
		Base_op[[~<=]] = [[<=]] in Bin_op	
Op_name[[=]] = "equal" in Name		Base_op[[=]] = [[=]] in Bin_op	
		Base_op[[~=]] = [[=]] in Bin_op	
Op_name[[>=]] = "ge" in Name		Base_op[[>=]] = [[>=]] in Bin_op	
		Base_op[[~>=]] = [[>=]] in Bin_op	
Op_name[[>]] = "gt" in Name		Base_op[[>]] = [[>]] in Bin_op	
		Base_op[[~>]] = [[>]] in Bin_op	
Op_name[[&]] = "and" in Name		Base_op[[&]] = [[&]] in Bin_op	

Op_name⟦[]⟧ ≡ "or" *in* Name *Base_op*⟦[]⟧ ≡ ⟦[]⟧ *in* Bin_op

C_op_call(name, expression$^+$, ce) ≡ *let* <expr; cel> ≡ *C_op_inv*(name, expression$^+$, ce)
in
case expr
elem invoke *of* Invoke
then *if* *Result_types*(invoke, cel) *is* D_type
then <expr; cel>
else *Bad_expr*(cel)
else *Bad_expr*(cel)

C_op_inv(name, expression$^+$, ce) ≡ *let* <expr*; cel> ≡ *C_expressions*(expression$^+$, ce)
d_type ≡ *Type_of*(*Head*(expr*), cel)
<expr; ce2> ≡ *C_d_oper*(⟦d_type$name[]⟧, cel)
in
C_call(expr, *Tail*(expr*), ce2)

For abbreviated forms of invocation, the exact operation to invoke is determined by the type of the first argument.

C_expression⟦expression1 **cand** expression2⟧(ce) ≡
let <expr1; cel> ≡ *C_expression*(expression1, ce)
<expr2; ce2> ≡ *C_expression*(expression2, cel)
in
if *Boolean*(expr1, ce2) ∧ *Boolean*(expr2, ce2)
then <⟦expr1 **cand** expr2⟧ *in* Expr; ce2>
else *Bad_expr*(ce2)

C_expression⟦expression1 **cor** expression2⟧(ce) ≡
let <expr1; cel> ≡ *C_expression*(expression1, ce)
<expr2; ce2> ≡ *C_expression*(expression2, cel)
in
if *Boolean*(expr1, ce2) ∧ *Boolean*(expr2, ce2)
then <⟦expr1 **cor** expr2⟧ *in* Expr; ce2>
else *Bad_expr*(ce2)

Boolean(expr, ce) ≡ *Type_of*(expr, ce) ≡ (⟦$\mathbf{du}_{\mathbf{bool}}$ []⟧ *in* D_type)

Booleans(expr*, ce) = *rec if Empty*(expr*)
then true
else *Boolean*(*Head*(expr*), ce) ∧ *Booleans*(*Tail*(expr*), ce)

Integer(expr, ce) = *Type_of*(expr, ce) = (⟦du_{int} [ ]⟧ *in* D_type)

Type_of⟦obj⟧(ce) = obj.d_type

Type_of⟦idn⟧(ce) = ce.info(idn) *to* D_type

Type_of⟦d_type$(comp*)⟧(ce) = d_type

Type_of⟦d_type$⟦expr: expr*⟧⟧(ce) = d_type

Type_of⟦up[d_type](expr)⟧(ce) = d_type

Type_of⟦down(expr)⟧(ce) = ce.down

Type_of⟦expr1 cand expr2⟧(ce) = ⟦du_{bool} [ ]⟧ *in* D_type

Type_of⟦expr1 cor expr2⟧(ce) = ⟦du_{bool} [ ]⟧ *in* D_type

Type_of⟦invoke⟧(ce) = *Result_types*(invoke, ce) *to* D_type

Types_of(expr*, ce) = *rec if Empty*(expr*)
then nil *in* D_type*
else *Cons*(*Type_of*(*Head*(expr*), ce),
Types_of(*Tail*(expr*), ce))

Result_types⟦expr(expr*)⟧(ce) = *Return_types*(*Type_of*(expr, ce))

Includes_all(d_type, d_type*) = *if Empty*(d_type*)
then true
else *Includes*(d_type, *Head*(d_type*)) ∧
Includes_all(d_type, *Tail*(d_type*))

Include(d_type1*, d_type2*) = rec *if* *Empty*(d_type1*) ∨ *Empty*(d_type2*)
then d_type1* = d_type2*
else *Includes*(*Head*(d_type1*), *Head*(d_type2*)) ∧
Include(*Tail*(d_type1*), *Tail*(d_type2*))

The two lists must be the same length and each type in the first list must include the corresponding type in the second list.

Includes(d_type1, d_type2) = *if* d_type1 *is* **Bad** ∨ d_type2 *is* **Bad**
then **false**
else if d_type1 = d_type2 ∨ d_type1 *is* **Any**
then **true**
else if d_type1 *is* **Routine**
then d_type2 *is* D_proc ∨ d_type2 *is* D_iter
else if d_type2 *is* **Routine**
then d_type1 *is* D_proc ∨ d_type1 *is* D_iter
else **false**

The type **bad** does not include and is not included in any type. The type **routine** includes and is included in every procedure and iterator type.

5.5 Equates

In this section we define the legality and meaning of equates with the function

C_equates: Equate* × CE → CE

We also define the function

Add_info: Idn × Info × CE → CE

which is used to define local identifiers.

Equates are checked as follows. We define a function domain Perm, consisting of permutation functions for the domain Equate*. That is, elements of this domain have functionality

Equate* → Equate*

and each element of the domain satisfies

perm(equate*) ⊂ equate* ∧ equate* ⊂ perm(equate*) ∧
Same_size(equate*, perm(equate*))

for all elements of Equate*. To check a list of equates, we collect all permutations of the list that are legal when checked in linear order. If at least one such permutation exists, then the equates are legal. The meaning of the equates is then obtained by evaluating one such permuted list; since all legal permutations will have the same meaning, the particular one we choose does not matter.

```
C_equates(equate*, ce) =
    let  ce1  =  ce[false • err]
         ce*  =  {ce2 | ∃perm[ce2 = E_equates(perm(equate*), ce1)  ∧  ¬ce2.err]}
     in
    if  Empty(ce*)
       then  ce[true • err]
       else  Head(ce*)[ce.err • err]
```

The use of set notation is informal here, as is the use of existential quantification. However, it should be clear that one could define a computable function to do the same thing.

```
E_equates(equate*, ce) = rec if Empty(equate*)
                         then ce
                         else let cel = E_equate(Head(equate*), ce)
                              in
                              E_equates(Tail(equate*), cel)

E_equate[[idn = constant]](ce) = let <obj; cel> = C_constant(constant, ce)
                                 in
                                 Add_info(idn, obj in Info, cel)

E_equate[[idn1 = idn2]](ce) = case ce.info(idn2)
                              elem tset of Tset
                              then let cel = ce[Cons(idn2, ce.used) • used]
                                   in
                                   Add_info(idn1, tset in Info, cel)
                              else let <obj; cel> = C_constant(idn2 in Constant, ce)
                                   in
                                   Add_info(idn1, obj in Info, cel)

E_equate[[idn = type_set]](ce) = let <tset; cel> = C_type_set(type_set, ce)
                                 in
                                 Add_info(idn, tset in Info, cel)
```

The definition of *C_type_set* is deferred to section 5.7.

```
E_equate[[rep = type_spec]](ce) = let <d_type; cel> = C_type_spec(type_spec, ce)
                                  in
                                  if ce.down is Bad ∧ ce.cluster
                                  then cel[d_type • down]
                                  else cel[true • err]
```

The representation type can be defined only within a cluster, and only once within a cluster.

```
Add_info(idn, info, ce) = let cel = ce[Cons(idn, ce.used) / used]
                          in
                          if ce.info(idn) is None ∨ ¬(idn ∈ ce.used)
                            then let info_map = ce.info[idn ← info]
                                 in
                                 cel[info_map / info][Cons(idn, ce.locals) / locals]
                            else cel[true / err]
```

An idn cannot be defined locally if it has been used as an external reference or if it is already defined locally.

5.6 Statements

The major functions defined in this section are:

C_statement:	Statement × CE → Stmt × CE
C_body:	Body × CE → Unit × CE
C_xbody:	Decl* × Body × CE → D_type* × Unit × CE
Restore:	CE × CE → Idn* × CE

C_xbody is used to check those tag, when, and others arms in which variable declarations appear prior to the body. *Restore* is used to reset the CE at the end of a scoping unit, and to generate a list of all variables defined in that unit; the arguments are the CEs before and after checking the unit, respectively.

```
C_body⟦equate* statement*⟧(ce) =
    let cel          = C_equates(equate*, ce)
        <stmt*; ce2> = C_statements(statement*, cel)
        <idn*; ce3>  = Restore(ce, ce2)
     in
    <⟦stmt* local idn*⟧; ce3>

Restore(cel, ce2) = Fix_info(ce2.locals, ce2.info, cel[ce2.used / used][ce2.err / err])
```

```
Fix_info(idn*, info_map, ce) = rec
    if Empty(idn*)
       then <nil in Idn*; ce>
       else let idn          = Head(idn*)
                <idnl*; cel> = Fix_info(Tail(idn*), info_map, ce)
            in
            if idn ∈ ce.locals
               then <idnl*; cel>
               else let info_mapl = cel.info[idn ← none in Info]
                        ce2       = cel[info_mapl • info]
                    in
                    if info_map(idn) is D_type
                       then <Cons(idn, idnl*); ce2>
                       else <idnl*; ce2>
```

Only those idns defined local to the given scoping unit are made undefined, not those defined in a surrounding unit.

```
Bad_stmt(ce) = <none in Stmt; ce[true • err]>

C_statements(statement*, ce) = rec
    let <stmt; cel>  = C_statement(Head(statement*), ce)
        <stmt*; ce2> = C_statements(Tail(statement*), cel)
    in
    if Empty(statement*)
       then <nil in Stmt*; ce>
       else <Cons(stmt, stmt*); ce2>

C_statement[[decl]](ce) = <none in Stmt; C_decls(decl in Decl*, ce)>
```

Declarations will not cause any immediate action during execution.

```
C_decls(decl*, ce) = rec let [[idn+: type_spec]] = Head(decl*)
                             <d_type; cel>       = C_type_spec(type_spec, ce)
                             ce2                 = Add_infos(idn+, d_type in Info, cel)
                         in
                         if Empty(decl*)
                            then ce
                            else C_decls(Tail(decl*), ce2)
```

```
Add_infos(idn*, info, ce) = rec if Empty(idn*)
                                then ce
                                else let cel = Add_info(Head(idn*), info, ce)
                                     in
                                     Add_infos(Tail(idn*), info, cel)

C_statement[idn: type_spec := expression](ce) =
    let <d_type; cel> = C_type_spec(type_spec, ce)
        ce2           = Add_info(idn, d_type in Info, cel)
        <expr; ce3>   = C_expression(expression, ce2)
     in
    if Includes(d_type, Type_of(expr, ce3))
       then <[(idn in Idn*) := (expr in Expr*)] in Stmt; ce3>
       else Bad_stmt(ce3)

C_statement[decl+ := invocation](ce) =
    let cel   = C_decls(decl+, ce)
        idn+  = Delist(decl+↓idns) to_in Idn+
     in
    C_statement([idn+ := invocation] in Statement, cel)

C_statement[idn+ := invocation](ce) =
    let <d_type*; cel> = Get_decls(idn+, ce)
        <expr; ce2>    = C_invocation(invocation, cel)
     in
    case expr
      elem invoke of Invoke
        then if ¬Duplicates(idn+) ∧ Include(d_type*, Result_types(invoke, ce2))
                then <[idn+ := invoke] in Stmt; ce2>
                else Bad_stmt(ce2)
        else Bad_stmt(ce2)
```

```
C_statement[[idn+ := expression+]](ce) =
    let  <d_type1*; ce1>  = Get_decls(idn+, ce)
         <expr*; ce2>     = C_expressions(expression+, ce1)
         d_type2*         = Types_of(expr*, ce2)
     in
    if ¬Duplicates(idn+) ∧ Included(d_type1*, d_type2*)
       then  <[[idn+ := expr*]] in Stmt; ce2>
       else  Bad_stmt(ce2)
```

Each expression must evaluate to a single object.

```
Get_decls(idn*, ce) = rec
    if Empty(idn*)
       then  <nil in D_type*; ce>
       else  let  <d_type*; ce1> = Get_decls(Tail(idn*), ce)
              in
             case ce.info(Head(idn*))
               elem  d_type of D_type
                 then  <Cons(d_type, d_type*); ce1>
               else  <Cons(bad in D_type, d_type*); ce1[true ← err]>
```

The idns must be declared variables.

```
C_statement[[invocation]](ce) = let  <expr; ce1> = C_invocation(invocation, ce)
                                 in
                                case expr
                                  elem  invoke of Invoke
                                    then  <invoke in Stmt; ce1>
                                  else  Bad_stmt(ce1)
```

```
C_statement[[expression1.name := expression2]](ce) =
    let  expression* = <expression1; expression2> in Expression+
         <expr; ce1> = C_op_inv(Add_set_(name), expression+, ce)
     in
    case expr
      elem  invoke of Invoke
        then  <invoke in Stmt; ce1>
      else  Bad_stmt(ce1)
```

*Add_set_*: Name → Name

*Add_set_* simply adds "set_" to the beginning of the name.

```
C_statement⟦expression1[expression2] := expression3⟧(ce) =
    let  expression+ = <expression1; expression2; expression3> in Expression+
         <expr; ce1> = C_op_inv("store" in Name, expression+, ce)
     in
    case expr
      elem invoke of Invoke
        then <invoke in Stmt; ce1>
      else Bad_stmt(ce1)

C_statement⟦if expression then body elseif_arm* end⟧(ce) =
    let  body1 = ⟦(nil in Equate*) (nil in Statement*)⟧
     in
    C_statement⟦if expression then body elseif_arm* else body1 end⟧(ce)

C_statement⟦if expression then body1 elseif_arm* else body2 end⟧(ce) =
    let  <expr; ce1>    = C_expression(expression, ce)
         <unit1; ce2>   = C_body(body1, ce1)
         <elseif*; ce3> = C_elseifs(elseif_arm*, ce2)
         <unit2; ce4>   = C_body(body2, ce3)
     in
    if Boolean(expr, ce4) ∧ Booleans(elseif*↓expr, ce4)
      then <⟦if expr then unit1 elseif* else unit2 end⟧ in Stmt; ce4>
      else Bad_stmt(ce4)

C_elseifs(elseif_arm*, ce) = rec
    let  ⟦elseif expression then body⟧ = Head(elseif_arm*)
         <expr; ce1>                   = C_expression(expression, ce)
         <unit; ce2>                   = C_body(body, ce1)
         elseif                        = ⟦elseif expr then unit⟧
         <elseif*; ce3>                = C_elseifs(Tail(elseif_arm*), ce2)
     in
    if Empty(elseif_arm*)
      then <nil in Elseif*; ce>
      else <Cons(elseif, elseif*); ce3>
```

```
C_statement[[while expression do body end]](ce) =
    let  <expr; ce1>  = C_expression(expression, ce)
         <unit; ce2>  = C_body(body, ce1[true ⊕ loop])
         ce3          = ce2[ce.loop ⊕ loop]
     in
    if  Boolean(expr, ce3)
       then  <[[while expr do unit end]] in Stmt; ce3>
       else  Bad_stmt(ce3)
```

Break and continue statements are allowed in the body.

```
C_statement[[return(expression*)]](ce) =
    let  <expr*; ce1>  = C_expressions(expression*, ce)
         d_type*       = Types_of(expr*, ce1)
     in
    if  (ce.iter ∧ Empty(expression*)) ∨ (¬ce.iter ∧ Include(ce.results, d_type*))
       then let  expr1* = C_cvts(expr*, ce1.cvts, ce1)
                in
            <[[return(expr1*)]] in Stmt; ce1>
       else  Bad_stmt(ce1)
```

An iterator cannot return any objects. *C_cvts* is used to make explicit all implicit ups.

```
C_statement[[yield(expression*)]](ce) =
    let  <expr*; ce1>  = C_expressions(expression*, ce)
         d_type*       = Types_of(expr*, ce1)
     in
    if  ce.iter ∧ Include(ce.results, d_type*)
       then let  expr1* = C_cvts(expr*, ce1.cvts, ce1)
                in
            <[[yield(expr1*)]] in Stmt; ce1>
       else  Bad_stmt(ce1)
```

Only iterators can yield objects.

```
C_statement[[signal name(expression*)]](ce) =
    let  <expr*; cel>   = C_expressions(expression*, ce)
         d_type*        = Types_of(expr*, cel)
         <bool; bool*>  = C_sig(name, d_type*, ce.sigs)
     in
    if bool
       then let  expr1* = C_cvts(exprs*, bool*, cel)
                 in
            <[[signal name(expr1*)]] in Stmt; cel>
       else  Bad_stmt(cel)

C_sig(name, d_type*, sig*) = rec
    let  <d_cond; bool*>     = Head(sig*)
         [[name1(d_type1*)]] = d_cond
     in
    if Empty(sig*)
       then  <false; nil in Bool*>
    else if  name = name1
       then  <Include(d_type1*, d_type*); bool*>
    else  C_sig(name, d_type*, Tail(sig*))

C_cvts(expr*, bool*, ce) = rec
    let  expr   = Head(expr*)
         expr1* = C_cvts(Tail(expr*), Tail(bool*), ce)
     in
    if Empty(expr*)
       then  nil in Expr*
    else if  Head(bool*)
       then  Cons([[up[ce.up](expr)]] in Expr, expr1*)
    else  Cons(expr, expr1*)
```

```
C_statement[[exit name(expression*)]](ce) =
    let  <expr*; cel> = C_expressions(expression*, ce)
         d_type*      = Types_of(expr*, cel)
     in
    if C_exit(name, d_type*, cel.handles)
       then <[[exit name(expr*)]] in Stmt; cel>
       else Bad_stmt(cel)

C_exit(name, d_type*, handle*) = rec
    if Empty(handle*)
       then false
       else case Head(handle*)
              elem name* of Name*
                then ¬(name ∈ name*) ∧ C_exit(name, d_type*, Tail(handle*))
              elem <name*; d_type1*> of D_hand
                then if name ∈ name*
                        then d_type* = d_type1*
                        else C_exit(name, d_type*, Tail(handle*))
              elem others of Others
                then false
              end
```

Local exits must have corresponding handlers, the result objects cannot be thrown away, and the types of the results must equal the types of the receiving variables.

```
C_statement[[break]](ce) = if ce.loop
                              then <break in Stmt; ce>
                              else Bad_stmt(ce)
```

A break statement is legal only in the body of a loop statement.

```
C_statement[[continue]](ce) = if ce.loop
                                 then <continue in Stmt; ce>
                                 else Bad_stmt(ce)
```

A continue statement is legal only in the body of a loop statement.

```
C_statement[[begin body end]](ce) = let <unit; ce1> = C_body(body, ce)
                                    in
                                    <[[begin unit end]] in Stmt; ce1>

C_statement[[tagcase expression tag_arm+ end]](ce) =
    let <expr; ce1>                    = C_expression(expression, ce)
        <tag*; name*; d_comp1*; ce2>   = C_tag_arms(tag_arm+, ce1)
        name1*                         = Order(name*, name*)
     in
    case Type_of(expr, ce2)
      elem [[oneof[d_comp2*]]] of D_oneof
        then if name1* = d_comp2*↓name ∧ d_comp1* ⊂ d_comp2*
                then <[[tagcase expr tag* end]] in Stmt; ce2>
                else Bad_stmt(ce2)
        else Bad_stmt(ce1)
```

The name list returned by *C_tag_arms* contains every name appearing on an arm; the d_comp list contains a name-d_type pair for every name appearing on an arm with a variable. Each tag name must appear exactly once on some arm, and only tag names can appear. If an arm has a variable to receive the value part of the object, the type of the variable must equal the value type corresponding to every tag on the arm.

```
C_statement[[tagcase expression tag_arm+ others: body end]](ce) =
    let  <expr; ce1>                    = C_expression(expression, ce)
         <tag1*; name1*; d_comp1*; ce2> = C_tag_arms(tag_arm+, ce1)
         <unit; ce3>                    = C_body(body, ce2)
         tag2*                          = Append(tag1*, [[others: unit]] in Tag)
     in
    case Type_of(expr, ce3)
      elem [[oneof[d_comp2*]]] of D_oneof
        then if name1* ⊂ d_comp2*↓name ∧ ¬Duplicates(name1*) ∧
                ¬Same_size(name1*, d_comp2*↓name) ∧ d_comp1* ⊂ d_comp2*
                then <[[tagcase expr tag2* end]] in Stmt; ce3>
                else Bad_stmt(ce3)
        else Bad_stmt(ce1)
```

Each tag name can appear at most once, but at least one tag must be missing.

```
C_tag_arms(tag_arm*, ce) = rec
    let  <tag; name1*; d_comp1*; ce1>    = C_tag_arm(Head(tag_arm*), ce)
         <tag1*; name2*; d_comp2*; ce2>  = C_tag_arms(Tail(tag_arm*), ce1)
     in
    if Empty(tag_arm*)
       then <nil in Tag*; nil in Name*; nil in D_comp*; ce>
       else <Cons(tag, tag1*); Concat(name1*, name2*); Concat(d_comp1*, d_comp2*);
             ce2>

C_tag_arm[[tag name+: body]](ce) =
    let  <unit; ce1> = C_body(body, ce)
     in
    <[[tag name+: unit]] in Tag; name+; nil in D_comp*; ce1>

C_tag_arm[[tag name+ (idn: type_spec): body]](ce) =
    let  decl*                 = [[(idn in Idn+): type_spec]] in Decl*
         <d_type*; unit; ce1>  = C_xbody(decl*, body, ce)
         d_comp*               = Get_d_comps(name+, d_type* to D_type)
     in
    <[[tag name+ (idn): unit]] in Tag; name+; d_comp*; ce1>
```

```
C_xbody(decl*)[[equate* statement*]](ce) =
    let  ce1              = C_equates(equate*, ce)
         ce2              = C_decls(decl*, ce1)
         <d_type*; ce3>   = Get_decls(Delist(decl*↓idns), ce2)
         <stmt*; ce4>     = C_statements(statement*, ce3)
         <idn*; ce5>      = Restore(ce, ce4)
     in
    <d_type*; [[stmt* local idn*]]; ce5>
```

The declarations can make use of the equates in the body.

```
C_statement[[for decl* in invocation do body end]](ce) =
    let  statement*  = Make_statements(decl*)
         idn*        = Delist(decl*↓idns)
         statement   = [[for idn* in invocation do body end]] in Statement
         body1       = [[(nil in Equate*) Append(statement*, statement)]]
     in
    C_statement([[begin body1 end]] in Statement, ce)

Make_statements(decl*) = rec
    if Empty(decl*)
      then nil in Statement*
      else Cons(Head(decl*) in Statement, Make_statements(Tail(decl*)))

C_statement[[for idn* in invocation do body end]](ce) =
    let  <d_type*; ce1>  = Get_decls(idn*, ce)
         <expr; ce2>     = C_iter_inv(invocation, ce1)
         <unit; ce3>     = C_body(body, ce2[true • loop])
         ce4             = ce3[ce2.loop • loop]
     in
    case expr
      elem invoke of Invoke
        then let [[expr1(expr*)]] = invoke
             in
               if Include(d_type*, Yield_types(Type_of(expr1, ce4)))
                 then <[[for idn* in invoke do unit end]] in Stmt; ce4>
                 else Bad_stmt(ce4)
      else Bad_stmt(ce4)
```

```
C_iter_inv[[expression(expression*)]](ce) =
    let  <expr; ce1>   = C_expression(expression, ce)
         <expr*; ce2>  = C_expressions(expression*, ce1)
         d_type*       = Types_of(expr*, ce2)
     in
    case Type_of(expr, ce2)
      elem [[itertype (d_type1*) yields (d_type2*) signals (d_cond*)]] of D_iter
        then if Include(d_type1*, d_type*) ∧ C_d_conds(d_cond*, ce2.handles)
                then <[[expr(expr*)]] in Expr; ce2>
                else Bad_expr(ce2)
        else Bad_expr(ce2)
```

For an iterator invocation to be legal, the arguments must be of the correct types and all exceptional conditions that can be raised must be legal with respect to the surrounding handlers.

```
Yield_types(d_type) =
    case d_type
      elem [[itertype (d_type1*) yields (d_type2*) signals (d_cond*)]] of D_iter
        then d_type2*
      end
```

```
C_statement[[statement except when_arm+ end]](ce) =
    let  <catch*; name*; handle*; ce1> = C_when_arms(when_arm+, ce)
         ce2                           = ce1[Concat(handle*, ce.handles) ⊕ handles]
         <stmt; ce3>                   = C_statement(statement, ce2)
         ce4                           = ce3[ce.handles ⊕ handles]
     in
    if ¬Duplicates(name*) ∧ C_d_cond(Fail(), handle*)
       then <[[stmt except catch* end]] in Stmt; ce4>
       else Bad_stmt(ce4)
```

The name list returned by *C_when_arms* contains every name appearing on an arm; the handle list contains an element for every arm. A given exception name can only be listed once in an **except** statement. An exception can be listed even if the exception cannot occur during evaluation. The *failure* exception can be raised at any time, and must be legal with respect to every **except** statement.

```
C_statement[[statement except when_arm+ others_arm end]](ce) =
    let  <catch1*; name*; handle1*; ce1>  = C_when_arms(when_arm+, ce)
         <catch; ce2>                     = C_others_arm(others_arm, ce1)
         catch2*                          = Append(catch1*, catch)
         handle2*                         = Append(handle1*, others in Handle)
         ce3                              = ce2[Concat(handle2*, ce.handles) o handles]
         <stmt; ce4>                      = C_statement(statement, ce3)
         ce5                              = ce4[ce.handles o handles]
     in
    if ¬Duplicates(name*) ∧ C_d_cond(Fail(), handle1*)
       then <[[stmt except catch2* end]] in Stmt; ce5>
       else Bad_stmt(ce0)

Fail() = let  name    = "failure" in Name
             d_type*  = [[du_string []]] in D_type in D_type*
         in
         [[name(d_type*)]]

C_when_arms(when_arm*, ce) = rec
    let  <catch; name1*; handle; ce1>     = C_when_arm(Head(when_arm*), ce)
         <catch*; name2*; handle*; ce2>   = C_when_arms(Tail(when_arm*), ce1)
     in
    if Empty(when_arm*)
       then <nil in Catch*; nil in Name*; nil in Handle*; ce>
       else <Cons(catch, catch*); Concat(name1*, name2*); Cons(handle, handle*); ce2>

C_when_arm[[when name+(decl*): body]](ce) =
    let  <d_type*; unit; ce1>  = C_xbody(decl*, body, ce)
         handle                = <name*; d_type*> in Handle
         idn*                  = Dclist(decl*, idns)
     in
    <[[when name+(idn*): unit]] in Catch; name+; handle; ce1>

C_when_arm[[when name+(*): body]](ce) =
    let  <unit; ce1>  = C_body(body, ce)
     in
    <[[when name+(*): unit]] in Catch; name+; name+ in Handle; ce1>
```

C_others_arm⟦**others:** body⟧(ce) ≡ *let* <unit; cel> = *C_body*(body, ce)
in
<⟦**others:** unit⟧ *in* Catch; cel>

C_others_arm⟦**others** (idn: type_spec): body⟧(ce) ≡
let decl* = ⟦(idn *in* Idn+): type_spec⟧ *in* Decl*
<d_type*; unit; cel> = *C_xbody*(decl*, body, ce)
catch = ⟦**others** (idn): unit⟧ *in* Catch
in
if (d_type* *to* D_type) = (⟦$\mathbf{du}_{\mathbf{string}}$ []⟧ *in* D_type)
then <catch; cel>
else <catch; cel[**true** ⊛ err]>

5.7 Interface Specifications

The major functions defined in this section are:

Derive_specs: Full_module* × CE → DU_spec* × CE

Same_spec: DU_spec × DU_spec → Bool

Interface specifications are obtained in the following manner. First, the existing interfaces for the abstractions being implemented are collected, and the special interfaces type and routine are installed. Then an external and an internal interface are derived from each module. The external and internal interfaces are identical for a routine; the internal interface for a cluster contains the interfaces of all cluster routines, while the external interface contains only the interfaces of the primitive operations. The external interface of each module is checked against the interface for the corresponding abstraction, if that abstraction has an interface. After all interfaces are derived, the external interfaces are installed; the internal interfaces are returned for use when checking the modules in their entirety, as defined in the next section.

```
Derive_specs(full_module*, ce) =
    let du_spec* = Get_du_specs(full_module*↓module, ce)
        cel      = Init_specs(full_module*↓module, ce)
     in
    Get_specs(full_module*, du_spec*, cel)
```

```
Get_du_specs(module*, ce) = rec
    if Empty(module*)
       then nil in DU_spec*
       else let du_spec* = Get_du_specs(Tail(module*), ce)
                idn      = Get_mod_idn(Head(module*))
            in
            case ce.info(idn)
              elem du of DU
                then Cons(ce.specs(du), du_spec*)
              else Cons(none in DU_spec, du_spec*)
```

Get_du_specs collects the existing interface of each abstraction.

```
Init_specs(module*, ce) = rec
    if Empty(module*)
       then ce
       else let cel = Init_specs(Tail(module*), ce)
                idn = Get_mod_idn(Head(module*))
            in
            if Head(module*) is Cluster
              then New_spec(idn, type in DU_spec, cel)
              else New_spec(idn, routine in DU_spec, cel)
```

Init_specs installs the special interfaces.

```
Get_mod_idn[[procedure]] = procedure.idn1

Get_mod_idn[[iterator]] = iterator.idn1

Get_mod_idn[[cluster]] = cluster.idn1

New_spec(idn, du_spec, ce) = case ce.info(idn)
                               elem du of DU
                                 then let spec_map = ce.specs[du ← du_spec]
                                      in
                                      ce[spec_map ∘ specs]
                               else ce[true ∘ err]
```

```
Get_specs(full_module*, du_spec*, ce) = rec
    let [[equate* module]]          = Head(full_module*)
        ce1                         = C_equates(equate*, ce)
        <du_spec1; du_spec2; ce2>   = Get_spec(module, ce1)
        ce3                         = ce[ce2.err ● err]
        <du_spec1*; ce4>            = Get_specs(Tail(full_module*), Tail(du_spec*), ce3)
        ce5                         = New_spec(Get_mod_id(module), du_spec2, ce4)
        du_spec2*                   = Cons(du_spec1, du_spec1*)
     in
    if Empty(full_module*)
       then <nil in DU_spec*; ce>
    else if Same_spec(Head(du_spec*), du_spec2)
       then <du_spec2*; ce5>
    else <du_spec2*; ce5[true ● err]>
```

The first du_spec returned by *Get_spec* is the internal interface; the second du_spec is the external interface. *Get_specs* installs the external interfaces and returns the internal interfaces.

```
Get_spec[[procedure]](ce) = let <r_spec; ce1> = Get_r_spec(procedure in Routine, ce)
                            in
                              <r_spec in DU_spec; r_spec in DU_spec; ce1>

Get_spec[[iterator]](ce) = let <r_spec; ce1> = Get_r_spec(iterator in Routine, ce)
                           in
                           <r_spec in DU_spec; r_spec in DU_spec; ce1>
```

```
Get_spec[[idn1 = cluster [decl*] is idn+ where restriction*
                 equate+
                 routine+
                 end idn2]](ce) =
    let  ce1                 = Set_op_names(routine+, ce[true ⊕ cluster])
         ce2                 = C_defs(decl*, equate+, ce1)
         ce3                 = C_restrictions(restriction*, ce2)
         idn*                = Delist(decl*↓idns)
         d_parm*             = Get_d_parms(idn*, ce3)
         ce4                 = Add_up_type(idn1, idn*, ce3)
         <op_spec1*; ce5>    = Get_op_specs(routine+, idn*, ce4)
         op_spec2*           = Get_ext_specs(op_spec1*, idn+)
         op_spec3*           = Order(op_spec2*↓name, op_spec2*)
     in
    <[[[d_parm*]: op_spec1*]] in DU_spec; [[[d_parm*]: op_spec3*]] in DU_spec; ce5>

Set_op_names(routine*, ce) = rec
    if Empty(routine*)
       then ce
       else let  idn = Get_op_idn(Head(routine*))
                 ce1 = Add_info(idn, routine in Info, ce)
            in
            Set_op_names(Tail(routine*), ce1)

Get_op_idn[[procedure]] = procedure.idn1

Get_op_idn[[iterator]] = iterator.idn1
```

```
Add_up_type(idn, idn*, ce) =
    if Empty(idn*)
      then let  <d_type; ce1> = C_type_spec(idn in Type_spec, ce)
            in
           ce1[d_type • up][nil in Obj* • parms]
      else let  constant+     = Make_constants(idn*) to_in Constant+
                <d_type; ce1> = C_type_spec([idn[constant+]] in Type_spec, ce)
                <obj*; ce2>   = C_constants(constant+, ce1)
            in
           ce2[d_type • up][obj* • parms]

Make_constants(idn*) = rec if Empty(idn*)
                           then nil in Constant*
                           else Cons(Head(idn*) in Constant,
                                     Make_constants(Tail(idn*)))

Get_op_specs(routine*, idn*, ce) = rec
    let  <r_spec; ce1>           = Get_r_spec(Head(routine*), ce)
         [[d_parm1*]: d_type]    = r_spec
         name                    = Make_name(Get_op_id(Head(routine*)))
         d_parm2*                = Get_d_parms(idn*, ce1)
         d_parm3*                = Concat(d_parm2*, d_parm1*)
         op_spec                 = [name[d_parm3*]: d_type]
         <op_spec*; ce2>         = Get_op_specs(Tail(routine*), idn*, ce[ce1.err • err])
     in
    if Empty(routine*)
      then <nil in Op_spec*; ce>
      else <Cons(op_spec, op_spec*); ce2>
```

The interface of each cluster routine is first derived essentially as if the routine were a module, and then the d_parms for the cluster parameters are added to the interface.

```
Get_ext_specs(op_spec*, idn*) = rec
    if Empty(op_spec*)
       then nil in Op_spec*
       else let [name[d_parm*]: d_type] = Head(op_spec*)
            in
            if Make_idn(name) ∈ idn*
               then Cons(Head(op_spec*), Get_ext_specs(Tail(op_spec*), idn*))
               else Get_ext_specs(Tail(op_spec*), idn*)
```

The check that each idn names an operation, and that no idn is listed twice, is made later as part of the check of the entire module.

Make_idn: Name → Idn

Make_idn returns the idn corresponding to the given name.

```
Get_r_spec[idn1 = proc [decl1*] (decl2*) returns (type_spec*) signals (cond_spec*)
                                                  where restriction*
                  equate*
                  statement*
                  end idn2](ce) =
    let  ce1              = C_defs(decl1*, equate*, ce)
         ce2              = C_restrictions(restriction*, ce1)
         d_parm*          = Get_d_parms(Delist(decl1*↓idns), ce2)
         ce3              = C_head_decls(decl2*, ce2)
         <d_type1*; ce4>  = Get_decls(Delist(decl2*↓idns), ce3)
         <d_type2*; ce5>  = C_head_types(type_spec*, ce4)
         <d_cond*; ce6>   = C_head_conds(cond_spec*, ce5)
         <d_type; ce7>    = C_proc_type(d_type1*, d_type2*, d_cond*, ce6)
         name             = Make_name(idn1)
     in
     <[name[d_parm*]: d_type]; ce7>
```

Get_r_spec is used for all procedures and iterators, including cluster routines. The use of cvt is controlled by a flag in the CE. The environment is not restored at the end so that any restrictions on cluster parameters imposed by the routine can be collected by *Get_op_specs*.

```
Get_r_spec[idn1 = iter [decl1*] (decl2*) yields (type_spec*) signals (cond_spec*)
                                        where restriction*
                  equate*
                  statement*
                  end idn2](ce) =
    let  ce1              = C_defs(decl1*, equate*, ce)
         ce2              = C_restrictions(restriction*, ce1)
         d_parm*          = Get_d_parms(Delist(decl1*↓idns), ce2)
         ce3              = C_head_decls(decl2*, ce2)
         <d_type1*; ce4>  = Get_decls(Delist(decl2*↓idns), ce3)
         <d_type2*; ce5>  = C_head_types(type_spec*, ce4)
         <d_cond*; ce6>   = C_head_conds(cond_spec*, ce5)
         <d_type; ce7>    = C_iter_type(d_type1*, d_type2*, d_cond*, ce6)
         name             = Make_name(idn1)
     in
    <[name[d_parm*]: d_type]; ce7>

C_defs(decl*, equate*, ce) =
    let  ce1 = ce[false ● err]
         ce* = {ce2 | ∃(equate1*, equate2*, perm)
                          [equate* = perm(Concat(equate1*, equate2*)) ∧
                           ce2 = C_defs1(equate1*, decl*, equate2*, ce1) ∧
                           ¬ce2.err]}
     in
    if  Empty(ce*)
      then  ce[true ● err]
      else  Head(ce*)[ce.err ● err]
```

Since parameter declarations can involve equated identifiers, we treat the list of declarations essentially as a single equate, and look for a legal reordering of all equates.

```
C_defs1(equate1*, decl*, equate2*, ce) = let  ce1 = C_equates(equate1*, ce)
                                              ce2 = C_parm_decls(decl*, ce1)
                                         in
                                         C_equates(equate2*, ce2)
```

```
C_parm_decls(decl*, ce) = rec
    let [[idn+: type_spec]] = Head(decl*)
        cel                 = C_parm_decls(Tail(decl*), ce)
     in
    if Empty(decl*)
       then ce
    else if type_spec is Type
       then Add_infos(idn+, nil in Op_decl* in Info, cel)
    else let <d_type; ce2> = C_type_spec(type_spec, cel)
          in
         case d_type
           elem [[du[obj*]]] of D_mod
             then if du ∈ <du_null ; du_bool ; du_int ; du_real ; du_char ; du_string >
                     then Add_infos(idn+, d_type in Constraint in Info, ce2)
                     else ce2[true ● err]
           else ce2[true ● err]

Get_d_parms(idn*, ce) = rec
    let idn      = Head(idn*)
        d_parm*  = Get_d_parms(Tail(idn*), ce)
     in
    if Empty(idn*)
       then nil in D_parm*
       else case ce.info(idn)
              elem op_decl* of Op_decl*
                then Cons([[idn: (op_decl* in Constraint)]], d_parm*)
              elem constraint of Constraint
                then Cons([[idn: constraint]], d_parm*)
              else Cons([[idn: (bad in D_type in Constraint)]], d_parm*)

C_restrictions(restriction*, ce) = rec
    if Empty(restriction*)
       then ce
       else let cel = C_restriction(Head(restriction*), ce)
             in
            C_restrictions(Tail(restriction*), cel)
```

```
C_restriction⟦idn has oper_decl+⟧(ce) =
    let  <op_decl*; ce1> = C_oper_decls(oper_decl*, ce)
     in
    Add_op_decls(idn, op_decl*, ce1)

C_restriction⟦idn in type_set⟧(ce) =
    let  <tset; ce1>              = C_type_set(type_set, ce)
         ⟦idn1 has op_decl*⟧      = tset
         obj                      = ⟦(idn in D_type in Val): (type in D_type)⟧
         op_decl1*                = Subst(obj in Obj*, idn1 in Idn*, op_decl*)
     in
    Add_op_decls(idn, op_decl1*, ce1)

C_type_set⟦(idn1 | idn2 has oper_decl+ equate*)⟧(ce) =
    let  ce1                  = Add_info(idn1, nil in Op_decl* in Info, ce)
         ce2                  = C_equates(equate*, ce1)
         <op_decl*; ce3>      = C_oper_decls(oper_decl+, ce2)
         info                 = op_decl* in Constraint in Info
         ce4                  = Add_info(idn1, info, ce[ce3.err ∘ err])
         ce5                  = C_equates(equate*, ce4)
         <op_decl1*; ce6>     = C_oper_decls(oper_decl+, ce5)
         tset                 = ⟦idn1 has op_decl1*⟧
         <idn*; ce7>          = Restore(ce, ce6)
     in
    if  idn1 = idn2
      then  <tset; ce7>
      else  <tset; ce7[true ∘ err]>
```

A type_set is treated much like a module heading. First the oper_decls are evaluated assuming the dummy parameter has any and all operations, and then the oper_decls are rechecked with complete information about the operations.

```
C_type_set[[idn]](ce) = let  cel = ce[Cons(idn, ce.used) • used]
                        in
                       case ce.info(idn)
                         elem  tset of Tset
                           then  <tset; cel>
                         else  <[[idn has (nil in Op_decl*)]]; cel[true • err]>

C_oper_decls(oper_decl*, ce) = rec
    let  [[oper_name+: type_spec]] = Head(oper_decl*)
         <d_type; cel>             = C_type_spec(type_spec, ce)
         <op_decl1*; ce2>          = C_oper_names(oper_name+, d_type, cel)
         <op_decl2*; ce3>          = C_oper_decls(Tail(oper_decl*), ce2)
     in
    if  Empty(oper_decl*)
      then  <nil in Op_decl*; ce>
    else if  d_type is D_proc ∨ d_type is D_iter
      then  <Concat(op_decl1*, op_decl2*); ce3>
    else  <op_decl2*; ce3[true • err]>

C_oper_names(oper_name*, d_type, ce) = rec
    let  [[name[constant*]]] = Head(oper_name*)
         <obj*; cel>         = C_constants(constant*, ce)
         op_decl             = [[name[obj*]: d_type]]
         <op_decl*; ce2>     = C_oper_names(Tail(oper_name*), d_type, cel)
     in
    if  Empty(oper_name*)
      then  <nil in Op_decl*; ce>
    else if  L_parms(obj*, ce2)
      then  <Cons(op_decl, op_decl*); ce2>
    else  <op_decl*; ce2[true • err]>
```

References to type parameters and to the abstractions being implemented are not allowed in the evaluated constants of restriction oper_names. *L_parms* performs this check.

```
Add_op_decls(idn, op_decl*, ce) =
    case ce.info(idn)
      elem op_decl1* of Op_decl*
        then let info_map = ce.info[idn ← Conca(op_decl*, op_decl1*)]
             in
             ce[info_map • info]
        else ce[true • err]

C_head_decls(decl*, ce) = rec
    if Empty(decl*)
       then ce
       else let [idn+: type_spec] = Head(decl*)
                <d_type; ce1>     = C_head_type(type_spec, ce)
                ce2               = Add_infos(idn+, d_type in info, ce1)
            in
            C_head_decls(Tail(decl*), ce2)

C_head_types(type_spec*, ce) = rec
    let <d_type; ce1>   = C_head_type(Head(type_spec*), ce)
        <d_type*; ce2>  = C_head_types(Tail(type_spec*), ce1)
    in
    if Empty(type_spec*)
       then <nil in D_type*; ce>
       else <Cons(d_type, d_type*); ce2>

C_head_type(type_spec, ce) = if type_spec is Cvt ∧ ¬(ce.up is Bad)
                             then <ce.up; ce>
                             else C_type_spec(type_spec, ce)
```

Cvt is permitted as a top-level type specification in declaring arguments and results (including yield and signal results) of cluster routines. When deriving interfaces, cvt is replaced by the abstract type.

```
C_head_conds(cond_spec*, ce) = rec
    let [[name(type_spec*)]] = Head(cond_spec*)
        <d_type*; ce1>       = C_head_types(type_spec*, ce)
        d_cond               = [[name(d_type*)]]
        <d_cond*; ce2>       = C_head_conds(Tail(cond_spec*), ce1)
     in
    if Empty(cond_spec*)
       then <nil in D_cond*; ce>
       else <Cons(d_cond, d_cond*); ce2>

Same_spec(du_spec1, du_spec2) =
    if du_spec1 is R_spec ∧ du_spec2 is R_spec
       then Same_r_spec(du_spec1 to R_spec, du_spec2 to R_spec)
    else if du_spec1 is T_spec ∧ du_spec2 is T_spec
       then Same_t_spec(du_spec1 to T_spec, du_spec2 to T_spec)
    else du_spec1 is None
```

The first argument is the interface from the library; the second argument is the derived interface.

```
Same_r_spec(r_spec1, r_spec2) =
    let [[[d_parm1*]: d_type1]] = r_spec1
        [[[d_parm2*]: d_type2]] = r_spec2
        idn1*                   = d_parm1*↓idn
        constraint1*            = d_parm1*↓constraint
        idn2*                   = d_parm2*↓idn
        constraint2*            = Replace(idn1*, idn2*, d_parm2*↓ constraint)
     in
    if Same_size(idn1*, idn2*)
       then Same_constraints(constraint1*, constraint2*) ∧
            d_type1 = Replace(idn1*, idn2*, d_type2)
       else false
```

Replace: $Idn^* \times Idn^* \times D \rightarrow D$

Replace is defined for all domains D. *Replace* does a simple substitution in the third argument, replacing each occurrence of an idn from the second list with the corresponding idn from the first (the two lists must be of equal length).

```
Same_constraints(constraint1*, constraint2*) = rec
    if Empty(constraint1*) ∨ Empty(constraint2*)
       then constraint1* = constraint2*
    else if Same_constraints(Tail(constraint1*), Tail(constraint2*))
       then case Head(constraint1*)
              elem op_decl1* of Op_decl*
                then case Head(constraint2*)
                       elem op_decl2* of Op_decl*
                         then op_decl1* ⊂ op_decl2* ∧ op_decl2* ⊂ op_decl1* ∧
                              Same_size(op_decl1*, op_decl2*)
                       else false
              else Head(constraint1*) = Head(constraint2*)
    else false
```

Corresponding type parameters must have identical op_decls, though the order in which the op_decls are listed in unimportant. Other corresponding parameters must be of the same type.

```
Same_t_spec(t_spec1, t_spec2) =
    let [[[d_parm1*]: op_spec1*]] = t_spec1
        [[[d_parm2*]: op_spec2*]] = t_spec2
        idn1*                     = d_parm1*↓idn
        constraint1*              = d_parm1*↓constraint
        idn2*                     = d_parm2*↓idn
        constraint2*              = Replace(idn1*, idn2*, d_parm2*↓constraint)
     in
    if Same_size(idn1*, idn2*)
       then Same_constraints(constraint1*, constraint2*) ∧
            Same_op_specs(op_spec1*, op_spec2*)
       else false
```

```
Same_op_specs(op_spec1*, op_spec2*) = rec
    let [[name1[d_parm1*]: d_type1]] = Head(op_spec1*)
        [[name2[d_parm2*]: d_type2]] = Head(op_spec2*)
     in
    if Empty(op_spec1*) ∨ Empty(op_spec2*)
       then op_spec1* = op_spec2*
       else name1 = name2 ∧
            Same_r_spec([[d_parm1*]: d_type1], [[d_parm2*]: d_type2]) ∧
            Same_op_specs(Tail(op_spec1*), Tail(op_spec2*))

L_parms(obj*, ce) = rec if Empty(obj*)
                          then true
                        else if L_parms(Tail(obj*), ce)
                          then let [[val: d_type]] = Head(obj*)
                                in
                               if d_type is Type
                                 then L_type(val to D_type, ce)
                                 else true
                        else false
```

Illegal references can occur only in types.

```
L_type(d_type, ce) =
    case d_type
      elem [[record[d_comp*]]] of D_record
        then L_types(d_comp*↓d_type, ce)
      elem [[oneof[d_comp*]]] of D_oneof
        then L_types(d_comp*↓d_type, ce)
      elem [[proctype (d_type1*) returns (d_type2*) signals (d_cond*)]] of D_proc
        then L_types(d_type1*, ce) ∧ L_types(d_type2*, ce) ∧
             L_types(Delist(d_cond*↓d_types), ce)
      elem [[itertype (d_type1*) returns (d_type2*) signals (d_cond*)]] of D_iter
        then L_types(d_type1*, ce) ∧ L_types(d_type2*, ce) ∧
             L_types(Delist(d_cond*↓d_types), ce)
      elem [[du[obj*]]] of D_mod
        then ¬(ce.specs(du) is Type) ∧ L_parms(obj*, ce)
      elem idn of Idn
        then false
      else true
```

Formal type parameters and instantiations of the (data) abstractions being implemented are illegal.

```
L_types(d_type*, ce) = rec if Empty(d_type*)
                           then true
                           else L_type(Head(d_type*), ce) ∧
                                L_types(Tail(d_type*), ce)
```

5.8 Modules

The major function defined in this section is:

Compile: Full_module* × Info_map × Library → Library

The domain Spec_map was defined in section 5.1. The domain Library is defined as:

Library: specs: Spec_map × imps: Imps_map

Imps_map: DU → Imp*

The domain Imp is defined in the next chapter.

To check a set of full_modules, we first derive interfaces as defined in the previous section. Then each module is checked in its entirety as follows. For procedure and iterator modules, the formal parameters are bound to their constraints as listed in the internal interface, and the header is reevaluated to ensure that the interface is consistent. The body of the routine is then checked. For cluster modules, the formal cluster parameters are bound to their constraints and the idns naming cluster routines are bound to the interfaces for those routines; then each routine is checked, as for procedure and iterator modules.

```
Compile(full_module*, info_map, library) =
    let ce1              = Create_ce(info_map, library.specs)
        <du_spec*; ce2>  = Derive_specs(full_module*, ce1)
        <mod*; ce3>      = C_full_modules(full_module*, du_spec*, ce2)
     in
    if ce3.err
       then library
       else <ce3.specs; Add_imps(mod*, library.imps)>
```

```
Add_imps(mod*, imps_map) = rec
    if Empty(mod*)
       then imps_map
       else let [[du = mod_form]] = Head(mod*)
                imp               = Meaning(mod_form)
                imp*              = Cons(imp, imps_map(du))
            in
            Add_imps(Tail(mod*), imps_map[du ← imp*])
```

Meaning is defined in the next chapter.

```
C_full_modules(full_module*, du_spec*, ce) = rec
    let [[equate* module]] = Head(full_module*)
        ce1                = C_equates(equate*, ce)
        ce2                = Install_spec(Head(du_spec*), ce1)
        <mod_form; ce3>    = C_module(module, ce2)
        du                 = Get_du(Get_mod_idn(module), ce3)
        mod                = [[du = mod_form]]
        ce4                = ce[ce3.err ⊕ err]
        <mod*; ce5>        = C_full_modules(Tail(full_module*), Tail(du_spec*), ce4)
     in
    if Empty(full_module*)
       then <nil in Mod*; ce>
       else <Cons(mod, mod*); ce5>

Install_spec[[[d_parm*]: d_type]](ce) = Add_parms(d_parm*, ce)

Install_spec[[[d_parm*]: op_spec*]](ce) = let ce1 = Add_parms(d_parm*, ce)
                                          in
                                          Add_ops(op_spec*, ce1)

Install_spec[[routine]](ce) = Install_spec[[type]](ce) = ⊥CE
```

The special interfaces exist only when deriving interfaces.

```
Install_spec[[none]](ce) = ce
```

```
Get_du(idn, ce) = case ce.info(du)
                    elem du of DU
                      then du
                    else du_?
```

It is an error if the idn does not name a DU, but the error is caught elsewhere.

```
Add_parms(d_parm*, ce) = rec
    if Empty(d_parm*)
      then ce
      else let [[idn: constraint]] = Head(d_parm*)
               cel                 = Add_info(idn, constraint in Info, ce)
           in
           Add_parms(Tail(d_parm*), cel)

Add_ops(op_spec*, ce) = rec
    if Empty(op_spec*)
      then ce
      else let [[name[d_parm*]: d_type]] = Head(op_spec*)
               idn                       = Make_idn(name)
               r_spec                    = [[d_parm*]: d_type]
               cel                       = Add_info(idn, r_spec in Info, ce)
           in
           Add_ops(Tail(op_spec*), cel)

C_module[[procedure]](ce) = let <op_form; cel> = C_routine(procedure in Routine, ce)
                            in
                            <op_form in Mod_form; cel>

C_module[[iterator]](ce) = let <op_form; cel> = C_routine(iterator in Routine, ce)
                           in
                           <op_form in Mod_form; cel>
```

```
C_module[[idn1 = cluster [decl*] is idn+ where restriction*
                 equate+
                 routine+
                 end idn2]](ce) =
    let  idn*          = Delist(decl*↓idns)
         ce1           = Add_up_type(idn1, idn*, ce[true • cluster])
         ce2           = C_equates(equate+, ce1)
         ce3           = L_parm_decls(decl*, ce2)
         ce4           = L_restrictions(restriction*, ce3)
         ce5           = Erase(idn*, ce4)
         <oper*; ce6>  = C_routines(routine+, ce5)
         mod_form      = [[cluster [idn*] oper* end]] in Mod_form
     in
    if idn1 = idn2 ∧ ¬(ce2.down is Bad) ∧ ¬Duplicates(idn+) ∧ idn+ ⊂ oper*↓idn
      then <mod_form; ce6>
      else <mod_form; ce6[true • err]>
```

A rep equate must occur in the list of equates. The cluster parameter bindings are removed before checking the routines; new bindings for these parameters are obtained from the interface of each routine.

```
Erase(idn*, ce) = rec if Empty(idn*)
                      then ce
                      else let info_map = ce.info[Head(idn*) ← none in Info]
                           in
                           Erase(Tail(idn*), ce[info_map • info])
```

```
C_routines(routine*, ce) = rec let  routine          = Head(routine*)
                                    idn              = Get_op_idn(routine)
                                    cel              = Install_op_spec(idn, ce)
                                    <op_form; ce2>   = C_routine(routine, cel)
                                    oper             = [idn = op_form]
                                    <idn*; ce3>      = Restore(ce, ce2)
                                    <oper*; ce4>     = C_routines(Tail(routine*), ce3)
                               in
                              if Empty(routine*)
                                then <nil in Oper*; ce>
                                else <Cons(oper, oper*); ce4>

Install_op_spec(idn, ce) = case ce.info(idn)
                             elem [[d_parm*]: d_type] of R_spec
                               then Add_parms(d_parm*, ce)
                             else ce[true ⊕ err]

L_parm_decls(decl*, ce) = rec
    if Empty(decl*)
      then ce
      else let [idn+: type_spec] = Head(decl*)
             in
           if type_spec is Type
             then L_parm_decls(Tail(decl*), ce)
             else let <d_type; cel> = C_type_spec(type_spec, ce)
                    in
                  L_parm_decls(Tail(decl*), cel)
```

The type specifications used to declare parameters are rechecked solely to gather uses of external references.

```
C_routine[idn1 = proc [decl1*] (decl2*) returns (type_spec*) signals (cond_spec*)
                                        where restriction*
                equate*
                statement*
                end idn2][ce) =
    let  ce1                    = ce[false • iter][Cons(idn1, ce.used) • used]
         ce2                    = C_equates(equate*, ce1)
         ce3                    = L_parm_decls(decl1*, ce2)
         <expr*; ce4>           = C_cvt_decls(decl2*, ce3)
         <d_type*; bool*; ce5>  = C_cvt_types(type_spec*, ce4)
         <sig*; ce6>            = C_cvt_conds(cond_spec*, ce5)
         sig1*                  = Append(sig*, <Fail(); false in Bool*>)
         ce7                    = L_restrictions(restriction*, ce6)
         ce8                    = ce7[d_type* • results][bool* • cvts][sig1* • sigs]
         idn1*                  = Delist(decl1*↓idns)
         idn2*                  = Delist(decl2*↓idns)
         <stmt1*; ce9>          = C_statements(statement*, ce8)
         stmt2*                 = Cons([idn2* := expr*] in Stmt, stmt1*)
         stmt3*                 = if Empty(type_spec*)
                                    then stmt2*
                                    else Append(stmt2*,
                                              [return(nil in Expr*)] in Stmt)
         <idn3*; ce10>          = Restore(ce, ce9)
         unit                   = [stmt3* local idn3*]
         op_form                = [routine [idn1*] (idn2*) unit end]
     in
    if idn1 = idn2
      then <op_form; ce10>
      else <op_form; ce10[true • err]>
```

C_routine is used for all procedures and iterators, including cluster routines. The use of cvt is controlled by a flag in the CE. In checking the argument declarations, a multiple assignment is constructed to make explicit all implicit downs. During execution a formal argument declared with cvt initially will be assigned an object of the abstract type, and then is reassigned an object of the representation type when the multiple assignment is performed.

```
C_routine[idn1 = iter [decl1*] (decl2*) yields (type_spec*) signals (cond_spec*)
                                                    where restriction*
                  equate*
                  statement*
                  end idn2](ce) =
    let  ce1                    = ce[true • iter][Cons(idn1, ce.used) • used]
         ce2                    = C_equates(equate*, ce1)
         ce3                    = L_parm_decls(decl1*, ce2)
         <expr*; ce4>           = C_cvt_decls(decl2*, ce3)
         <d_type*; bool*; ce5>  = C_cvt_types(type_spec*, ce4)
         <sig*; ce6>            = C_cvt_conds(cond_spec*, ce5)
         sig1*                  = Append(sig*, <Fail(); false in Bool*>)
         ce7                    = L_restrictions(restriction*, ce6)
         ce8                    = ce7[d_type* • results][bool* • cvts][sig1* • sigs]
         idn1*                  = Delist(decl1*↓idns)
         idn2*                  = Delist(decl2*↓idns)
         <stmt1*; ce9>          = C_statements(statement*, ce8)
         stmt2*                 = Cons([idn2* := expr*] in Stmt, stmt1*)
         stmt3*                 = Append(stmt2*, [return(nil in Expr*)] in Stmt)
         <idn3*; ce10>          = Restore(ce, ce9)
         unit                   = [stmt3* local idn3*]
         op_form                = [routine [idn1*] (idn2*) unit end]
     in
    if  idn1 = idn2
       then  <op_form; ce10>
       else  <op_form; ce10[true • err]>

L_restrictions(restriction*, ce) = rec
    if  Empty(restriction*)
       then  ce
       else  let  ce1 = L_restriction(Head(restriction*), ce)
              in
             L_restrictions(Tail(restriction*), ce1)
```

```
L_restriction[[idn has oper_decl+]](ce) =
    let  <op_decl*; cel> = C_oper_decls(oper_decl+, ce)
     in
    cel
```

The check that the idn is a type parameter was performed when deriving interfaces.

```
L_restriction[[idn in type_set]](ce) = let  <tset; cel> = C_type_set(type_set, ce)
                                        in
                                       cel
```

```
C_cvt_decls(decl*, ce) = rec
    let  [[idn+: type_spec]] = Head(decl*)
         <d_type; bool; cel> = C_cvt_type(type_spec, ce)
         ce2                 = Add_infos(idn+, d_type in Info, cel)
         expr1*              = Get_exprs(idn+, bool)
         <expr2*; ce3>       = C_cvt_decls(Tail(decl*), ce2)
     in
    if  Empty(decl*)
      then  <nil in Expr*; ce>
      else  <Concat(expr1*, expr2*); ce3>
```

```
Get_exprs(idn*, bool) = rec if Empty(idn*)
                            then  nil in Expr*
                            else  let  expr* = Get_exprs(Tail(idn*), bool)
                                       expr  = Head(idn*) in Expr
                                   in
                                  if  bool
                                    then  Cons([[down(expr)]] in Expr, expr*)
                                    else  Cons(expr, expr*)
```

```
C_cvt_types(type_spec*, ce) = rec
    let  <d_type; bool; ce1>     = C_cvt_type(Head(type_spec*), ce)
         <d_type*; bool*; ce2>   = C_cvt_types(Tail(type_spec*), ce1)
     in
    if Empty(type_spec*)
       then  <nil in D_type*; nil in Bool*; ce>
       else  <Cons(d_type, d_type*); Cons(bool, bool*); ce2>

C_cvt_type(type_spec, ce) = if type_spec is Cvt ∧ ¬(ce.down is Bad)
                            then <ce.down; true; ce>
                            else let <d_type; ce1> = C_type_spec(type_spec, ce)
                                 in
                                 <d_type; false; ce1>
```

Cvt is permitted as a top-level type specification in declaring arguments and results (including yield and signal results) of cluster routines. When checking the entire module, cvt is replaced by the representation type.

```
C_cvt_conds(cond_spec*, ce) = rec
    let  ⟦name(type_spec*)⟧ = Head(cond_spec*)
         <d_type*; bool*; ce1> = C_cvt_types(type_spec*, ce)
         d_cond                = ⟦name(d_type*)⟧
         sig                   = <d_cond; bool*>
         <sig*; ce2>           = C_cvt_conds(Tail(cond_spec*), ce1)
     in
    if Empty(cond_spec*)
       then  <nil in Sig*; ce>
       else  <Cons(sig, sig*); ce2>
```

6. Meaning

In this chapter we define the meaning of legal modules. Again, to minimize the number of forward references, we proceed from the bottom up. We begin by defining execution environments, and then we define the meaning of expressions, statements, and finally modules.

6.1 Environments

The domain Env of execution environments is defined as follows:

Env:	imps: Imp_map × vars: V_map × loops: Loop* × array: Array × arrays: A_map × record: Record × records: R_map
Imp_map:	DU → Imp
Imp:	Op + Type
Op:	Obj* × Obj* × Env → Result
Type:	Obj* × Name → Op
V_map:	Idn → Obj
Loop:	vars: Idn* × body: Unit × env: Env
A_map:	Array → Int × Obj*
R_map:	Record → Obj*
Result:	term: Term × objs: Obj* × env: Env
Term:	Normal + Return + Break + Continue + Signal + Exit + Loop_term
Signal:	signal: Signal × name: Name
Exit:	exit: Exit × name: Name
Loop_term:	Return + Signal + Exit

The meaning of the various components of an execution environment are:

imps - a mapping from DUs to implementations. This is not the mapping that is part of the library; rather, it is generated from the library by selecting a specific implementation for each abstraction used in the program.

vars - a mapping that defines the current object denoted by each variable. Uninitialized variables denote the bad object.

loops - a stack of closures for for statements. Each closure consists of the list of loop variables, the loop body, and the variable bindings active for the loop body.

array - a seed for generating new unique ids for array objects. Every id less than this seed is the id of an existing array object. (The domain Array is isomorphic to the natural numbers.)

arrays - a mapping that defines the current state (low bound and elements) of each existing array object.

record - a seed for generating new unique ids for record objects. Every id less than this seed is the id of an existing record object. (The domain Record is isomorphic to the natural numbers.)

records - a mapping that defines the current state of each existing record object. The components are in increasing lexicographic order according to their selector names.

We note that in a more "standard" semantic definition one would not coalesce all of this information into a single domain. Rather, the domain Env would consist solely of variable bindings (i.e., Env = V_map) and a separate domain, State, would be defined to consist of the other information. The principal advantage of such a scheme is that, although all of the information is needed to define the meaning of most constructs, certain evaluations (e.g., expression evaluation) cannot change (and therefore need not return) any variable bindings. However, as discussed in section 2.2.3, using a single domain greatly simplifies the task of propagating exceptions.

Primitive_env() = <*Primitive_imps*(); $\perp_{V\_map}$; $\perp_{Loops}$; $\perp_{Array}$; $\perp_{A\_map}$; $\perp_{Record}$; $\perp_{R\_map}$>

Primitive_env is used in the evaluation of constant expressions. Constant expressions do not contain variables and do not create or manipulate mutable objects.

Primitive_imps: → Imp_map

Primitive_imps returns an imp_map with implementations for the built-in types, the array type generator, and the procedure generator forms. Implementations of the **record**, **oneof**, **proctype**, and **itertype** type generators, and the type **routine** will be handled specially, similar to the way their interfaces were handled in section 5.3.

6.2 Expressions

The major functions defined in this section are:

E_expr:	Expr × Env → Result
E_exprs:	Expr* × Env → Result
E_invoke:	Invoke × Env → Result

E_exprs(expr*, env) = *rec*
  if *Empty*(expr*)
    then <**normal** *in* Term; nil *in* Obj*; env>
    else *res* <term1; obj1*; env1> = *E_expr*(*Head*(expr*); env)
      <term2; obj2*; env2> = *E_exprs*(*Tail*(expr*), env1)
      in
      <**normal** *in* Term; *Concat*(obj1*, obj2*); env2>

The *res* notation, defined in section 3.4, is used to propagate exceptions (and bottom, in the event of nontermination).

E_expr[[obj]](env) = <**normal** *in* Term; obj *in* Obj*; env>

Actual parameters are substituted for formal parameters throughout a module before evaluation takes place, as defined in section 6.4.

```
E_expr[[idn]](env) = let obj = env.vars(idn)
                     in
                     if obj = [[(bad in Val): (bad in D_type)]]
                       then Failure("uninitialized variable" in String, env)
                       else <normal in Term; obj in Obj*; env>

Failure(string, env) = let term   = <exit; "failure" in Name> in Term
                           d_type = [[du_string []]] in D_type
                           obj    = [[(string in Val): d_type]]
                       in
                       <term; obj in Obj*; env>

E_expr[[invoke]](env) = E_invoke(invoke, env)

E_invoke[[expr(expr*)]](env) =
    res <term1; obj1*; env1> = E_expr(expr, env)
        <term2; obj2*; env2> = E_exprs(expr*, env1)
    let <op; obj3*>          = Get_op(obj1* to Obj, env2)
        v_map                = λidn. [[(bad in Val): (bad in D_type)]]
        <term3; obj4*; env3> = op(obj3*, obj2*, env2[v_map ∘ vars])
    in
    Pass(term3, obj4*, env3[env2.vars ∘ vars])
```

Get_op returns the implementation and the actual parameters for the routine. *Pass* is used to propagate bottom in case the routine does not terminate. (The *res* notation cannot be used because the variable bindings must be restored.)

```
Pass(term, obj*, env) = if term is Normal ∨ true
                          then <term; obj*; env>
                          else ⊥Result
```

If the termination condition is bottom, then bottom is produced; otherwise the usual result is formed.

```
Get_op(obj, env) = case obj.val
                     elem [du[obj*]] of D_mod
                       then <env.imps(du) to Op; obj*>
                     elem [d_type$name[obj*]] of D_oper
                       then case d_type
                              elem d_record of D_record
                                then <Record_op(d_record, name); obj*>
                              elem d_oneof of D_oneof
                                then <Oneof_op(d_oneof, name); obj*>
                              elem d_proc of D_proc
                                then <Proctype_op(d_proc, name); obj*>
                              elem d_iter of D_iter
                                then <Itertype_op(d_iter, name); obj*>
                              elem routine of Routine
                                then <Routine_op(name); obj*>
                              elem [du[obj]*] of D_mod
                                then let type = env.imps(du) to Type
                                     in
                                       <type(obj]*, name); obj*>
                              end
                     end
```

Record_op: D_record × Name → Op
Oneof_op: D_oneof × Name → Op
Proctype_op: D_proc × Name → Op
Itertype_op: D_iter × Name → Op

Implementations of the **record**, **oneof**, **proctype**, and **itertype** type generators cannot be represented as elements of the domain Type, but implementations for the operations of any particular instantiation can be represented as elements of Op, as defined in section 6.5.

Routine_op: Name → Op

Routine_op represents the implementation of the type **routine**, as defined in section 6.5.

```
E_expr⟦d_type$(comp*)⟧(env) =
    res  <term; obj*; env1>  = E_exprs(comp*↓expr, env)
    let  obj1*               = Order(comp*↓name, obj*)
         record              = env1.record
         r_map               = env1.records[record ← obj1*]
         env2                = env1[New_record(record) • record][r_map • records]
     in
    <normal in Term; ⟦(record in Val): d_type⟧ in Obj*; env2>
```

New_record: Record → Record

New_record returns the successor to the given element. (The domain Record is isomorphic to the natural numbers.)

```
E_expr⟦d_type$[expr: expr*]⟧(env) =
    res  <term1; obj1*; env1>  = E_expr(expr, env)
         <term2; obj2*; env2>  = E_exprs(expr*, env1)
    let  obj                   = obj1* to Obj
         int                   = obj.val to Int
         array                 = env2.array
         a_map                 = env2.arrays[array ← <int; obj2*>]
         env3                  = env2[New_array(array) • array][a_map • arrays]
     in
    if L_state(int, obj2*)
      then <normal in Term; ⟦(array in Val): d_type⟧ in Obj*; env3>
      else Failure(? in String, env2)
```

The string argument to *Failure* is arbitrary. *L_state* is defined in section 6.5.

New_array: Array → Array

New_array returns the successor to the given element. (The domain Array is isomorphic to the natural numbers.)

```
E_expr⟦up[d_type](expr)⟧(env) = res <term; obj*; env1> = E_expr(expr, env)
                                let  val                = obj* to Obj in Val
                                 in
                                <normal in Term; ⟦val: d_type⟧ in Obj*; env1>
```

```
E_expr[[down(expr)]](env) = res <term; obj*; env1> = E_expr(expr, env)
                            let obj                = obj* to Obj
                             in
                            <normal in Term; obj.val to Obj in Obj*; env1>
```

If the type any could not be used as a representation type, then up could simply replace the d_type component of the object with the abstract type instead of adding a new layer; down would then replace the d_type component with the representation type instead of removing a layer. However, if the representation type were any, then information would be lost by such an "optimization", and correct invocations of force on down'ed objects would fail.

```
E_expr[[expr1 cand expr2]](env) = res <term; obj*; env1> = E_expr(expr1, env)
                                  let obj                = obj* to Obj
                                   in
                                  if obj.val to Bool
                                     then E_expr(expr2, env1)
                                     else <normal in Term; obj*; env1>

E_expr[[expr1 cor expr2]](env) = res <term; obj*; env1> = E_expr(expr1, env)
                                 let obj                = obj* to Obj
                                  in
                                 if obj.val to Bool
                                    then <normal in Term; obj*; env1>
                                    else E_expr(expr2, env1)
```

6.3 Statements

The major functions defined in this section are:

E_stmt:	Stmt × Env → Result
E_unit:	Unit × Env → Result
Assn1:	Idn × Obj × Env → Env
Assn:	Idn* × Obj* × Env → Env

Assn1 and *Assn* are used for assigning objects to variables.

```
E_unit[[stmt* local idn*]](env) = let <term; obj*; env1> = E_stmts(stmt*, env)
                                      env2              = Unassn(idn*, env1)
                                  in
                                  Pass(term, obj*, env2)
```

Pass is used to propagate bottom in case evaluation of the body does not terminate.

```
E_stmts(stmt*, env) = rec if Empty(stmt*)
                      then <normal in Term; nil in Obj*; env>
                      else res <term; obj*; env1> = E_stmt(Head(stmt*), env)
                           in
                           E_stmts(Tail(stmt*), env1)

Unassn(idn*, env) = rec if Empty(idn*)
                    then env
                    else let obj  = [[bad in Val): (bad in D_type)]]
                             env1 = Assn1(Head(idn*), obj, env)
                         in
                         Unassn(Tail(idn*), env1)
```

Variable bindings are removed by rebinding the variables to the bad object.

```
Assn1(idn, obj, env) = let v_map = env.vars[idn ← obj]
                       in
                       env[v_map • vars]

E_stmt[[none]](env) = <normal in Term; nil in Obj*; env>

E_stmt[[idn* := expr*]](env) = res <term; obj*; env1> = E_exprs(expr*, env)
                               let env2               = Assn(idn*, obj*, env1)
                               in
                               <normal in Term; nil in Obj*; env2>

E_stmt[[idn* := invoke]](env) = res <term; obj*; env1> = E_invoke(invoke, env)
                                let env2               = Assn(idn*, obj*, env1)
                                in
                                <normal in Term; nil in Obj*; env2>
```

```
Assn(idn*, obj*, env) = rec if Empty(idn*)
                        then env
                        else let env1 = Assn1(Head(idn*), Head(obj*), env)
                             in
                             Assn(Tail(idn*), Tail(obj*), env1)

E_stmt[[invoke]](env) = res <term; obj*; env1> = E_invoke(invoke, env)
                        in
                        <normal in Term; nil in Obj*; env1>

        The objects returned by the invocation are discarded.

E_stmt[[if expr then unit1 elseif* else unit2 end]](env) =
    res <term; obj*; env1> = E_expr(expr, env)
    let obj                = obj* to Obj
     in
    if obj.val to Bool
       then E_unit(unit1, env1)
       else E_elseifs(elseif*, unit2, env1)

E_elseifs(elseif*, unit, env) = rec
    if Empty(elseif*)
       then E_unit(unit, env)
       else let [[elseif expr then unit1]] = Head(elseif*)
            res <term; obj*; env1>         = E_expr(expr, env)
            let obj                        = obj* to Obj
             in
            if obj.val to Bool
               then E_unit(unit1, env1)
               else E_elseifs(Tail(elseif*), unit, env1)
```

```
E_stmt[[while expr do unit end]](env) = rec
    res <term1; obj1*; env1> = E_expr(expr, env)
    let obj                  = obj1* to Obj
     in
    if obj.val to Bool
       then let <term2; obj2*; env2> = E_unit(unit, env1)
             in
            if term2 is Normal ∨ term2 is Continue
               then E_stmt[[while expr do unit end]] in Stmt, env2)
            else if term2 is Break
               then <normal in Term; nil in Obj*; env2>
            else <term2; obj2*; env2>
       else <normal in Term; nil in Obj*; env1>
```

Executing a **break** statement in the body terminates the **while** statement. Executing a **continue** statement in the body causes the rest of the body to be skipped for that cycle, but execution of the **while** statement continues. Other non-**normal** terminations of the body terminate the **while** statement.

```
E_stmt[[return(expr*)]](env) = res <term; obj*; env1> = E_exprs(expr*, env)
                                in
                               <return in Term; obj*; env1>

E_stmt[[signal name(expr*)]](env) = res <term; obj*; env1> = E_exprs(expr*, env)
                                     let term1             = <signal; name> in Term
                                      in
                                     <term1; obj*; env1>

E_stmt[[exit name(expr*)]](env) = res <term; obj*; env1> = E_exprs(expr*, env)
                                   let term1             = <exit; name> in Term
                                    in
                                   <term1; obj*; env1>

E_stmt[[break]](env) = <break in Term; nil in Obj*; env>

E_stmt[[continue]](env) = <continue in Term; nil in Obj*; env>

E_stmt[[begin unit end]](env) = E_unit(unit, env)
```

E_stmt[[tagcase expr tag* end]](env) = *res* <term; obj*; env1> = *E_expr*(expr, env)
let [[val : d_type]] = obj* *to* Obj
[[name: obj]] = val *to* Oneof$_{tn}$
in
E_tags(tag*, name, obj, env1)

E_tags(tag*, name, obj, env) = *rec if Match_tag*(*Head*(tag*), name)
then E_tag(*Head*(tag*), obj, env)
else E_tags(*Tail*(tag*), name, obj, env)

Legality-checking has ensured that one of the arms will be executed.

Match_tag[[tag name* (idn): unit]](name) = name ∈ name*

Match_tag[[tag name*: unit]](name) = name ∈ name*

Match_tag[[others: unit]](name) = true

E_tag[[tag name* (idn): unit]](obj, env) = *let* env1 = *Bind*(idn, obj, env)
in
E_unit(unit, env1)

E_tag[[tag name*: unit]](obj, env) = *E_unit*(unit, env)

E_tag[[others: unit]](obj, env) = *E_unit*(unit, env)

E_stmt[[for idn* in invoke do unit end]](env) =
let loop* = *Cont*(idn*, unit; env.vars>, env.loops)
<term; obj*; env1> = *E_invoke*(invoke, env[loop* ⊕ loops])
<idn1*; unit1; v_map> = *Head*(env1.loops)
in
Pass(term, obj*, env1[v_map ⊕ vars][*Tail*(env1.loops) ⊕ loops])

The invoked iterator handles termination, including execution of break and continue statements. *Pass* is used to propagate bottom in case the iterator for some cycle of the loop body does not terminate.

```
E_stmt[[yield(expr*)]](env) =
    res  <term1; obj1*; env1>  = E_exprs(expr*, env)
    let  <idn*; unit; v_map>   = Head(env1.loops)
         env2                  = env1[v_map • vars][Tail(env1.loops) • loops]
         env3                  = Assn(idn*, obj1*, env2)
         <term2; obj2*; env4>  = E_unit(unit, env3)
         loop*                 = Cons(<idn*; unit; env4.vars>, env4.loops)
         env5                  = env4[env1.vars • vars][loop* • loops]
     in
    if term2 is Normal ∨ term2 is Continue
      then <normal in Term; nil in Obj*; env5>
    else if term2 is Break
      then <return in Term; nil in Obj*; env5>
    else <term2 to_in Loop_term in Term; obj2*; env5>
```

Executing a **break** statement in the body terminates the iterator (and the invoking **for** statement). Executing a **continue** statement in the body causes the rest of the body to be skipped for that cycle, but execution of the iterator continues. Other non-normal terminations of the body terminate the iterator, but the result must be propagated specially through the iterator and back to the caller.

```
E_stmt[[stmt except catch* end]](env) = let <term; obj*; env1> = E_stmt(stmt, env)
                                        in
                                       case term
                                         elem <exit; name> of Exit
                                           then E_catchs(catch*, name, obj*, env1)
                                         else <term; obj*; env1>
```

E_catchs(catch*, name, obj*, env) = *rec*
    if *Empty*(catch*)
      then *let* term = <exit; name> *in* Term
             in
           <term; obj*; env>
    else if *Match_catch*(*Head*(catch*), name)
      then *E_catch*(*Head*(catch*), name, obj*, env)
    else *E_catchs*(*Tail*(catch*), name, obj*, env)

Exits not caught by any handler of the **except** statement are propagated through unchanged.

Match_catch[**when** name* (idn*): unit](name) = name ∈ name*

Match_catch[**when** name* (*): unit](name) = name ∈ name*

Match_catch[**others**: unit](name) = **true**

Match_catch[**others** (idn): unit](name) = **true**

E_catch[**when** name* (idn*): unit](name, obj*, env) =
    let env1 = *Assn*(idn*, obj*, env)
     in
    E_unit(unit, env1)

E_catch[**when** name* (*): unit](name, obj*, env) = *E_unit*(unit, env)

E_catch[**others**: unit](name, obj*, env) = *E_unit*(unit, env)

E_catch[**others** (idn): unit](name, obj*, env) =
    let d_type = $[\![du_{string}\ []]\!]$ *in* D_type
        obj = [(*Make_string*(name) *in* Val): d_type]
        env1 = *Assn1*(idn, obj, env)
     in
    E_unit(unit, env1)

The idn is bound to a string representing the exception name.

Make_string: Name → String

Make_string returns the string corresponding to the given name. (Name and String are isomorphic domains.)

6.4 Modules

The major function defined in this section is:

Meaning: Mod_form → Imp

Meaning⟦op_form⟧ = (λ(obj1*, obj2*, env). *E_op*(op_form, obj1*, obj2*, env)) *in* Imp

Meaning⟦type_form⟧ = (λ(obj*, name). *M_type*(type_form, obj*, name)) *in* Imp

M_type⟦cluster [idn*] oper* end⟧(obj*, name) =
  let op_form1 = *Find_op*(name, oper*)
    op_form2 = *Subst*(obj*, idn*, op_form1)
  in
  λ(obj1*, obj2*, env). *E_op*(op_form2, obj1*, obj2*, env)

Find_op(name, oper*) = *rec let* ⟦idn = op_form⟧ = *Head*(oper*)
  in
  if *Make_idn*(name) = idn
    then op_form
    else *Find_op*(name, *Tail*(oper*))

Legality-checking has ensured that the operation exists.

```
E_op[[routine [idn1*] (idn2*) unit end]](obj1*, obj2*, env) =
    let unit1                    = Subst(obj1*, idn1*, unit)
        env1                     = Assn(idn2*, obj2*, env)
        <term; obj3*; env2>      = E_unit(unit1, env1)
     in
    case term
      elem normal of Normal
        then Failure("no return values" in String, env2)
      elem return of Return
        then <normal in Term; obj3*; env2>
      elem <signal name> of Signal
        then let exit = <exit; name>
             in
             <exit in Term; obj3*; env2>
      elem <exit; name> of Exit
        then if name = ("failure" in Name)
             then <term; obj3*; env2>
             else let string = Concat("unhandled exception: " in String,
                                      Make_String(name))
                  in
                  Failure(string, env2)
      elem loop_term of Loop_term
        then <loop_term to_in Term; obj3*; env2>
      end
```

As part of legality-checking, return statements are added to the ends of routines that do not return any objects; thus a routine body which evaluates to a normal result is in error. Signals are changed to exits at the routine boundary. Failure exits propagate through; any other exit is changed to a failure exit with the original exit name as the result object. A non-normal termination of a for loop body is passed back to the caller.

6.5 Built-in Abstractions

In this section we define the built-in procedure generator **force** and the operations of the built-in types and type generators. Each routine is defined either by giving a routine heading and an element of the domain Op, or by giving a routine written in CLU. The implementations are not necessarily the most efficient possible. We do not define any operations related to input/output, since I/O is not part of the language proper.

Several routines can signal *failure*, but the exact string returned can vary from one system implementation to another. We indicate these strings with a question mark, as in

Failure(? *in* String, env)

When computing with integers, the operators '+', '-', '*', '≤', etc. have their usual mathematical meaning. We also use the operators '/' and '//', defined by

$$x/y = q \wedge x//y = r \quad \text{iff} \quad (0 \le r < |y|) \wedge (x = y*q + r)$$

When computing with reals, we will treat $[\![int1\ e\ int2]\!]$ as the real number $int1*10^{int2}$. For example,

$$[\![int1\ e\ int2]\!] < [\![int3\ e\ int4]\!] \quad \text{iff} \quad int1*10^{int2} < int3*10^{int4}$$

An expression such as

$$[\![int1\ e\ int2]\!] + [\![int3\ e\ int4]\!]$$

means that one should choose the normalized element of Real that represents

$$int1*10^{int2} + int3*10^{int4}$$

and similarly for other operators. The exact normalization algorithm is not important here (though it must be the same as that used by the parser); normalization simply ensures that our use of strict element equality:

$$[\![int1\ e\ int2]\!] = [\![int3\ e\ int4]\!] \quad \text{iff} \quad int1 = int3 \wedge int2 = int4$$

corresponds to the usual meaning of equality.

6.5.1 Auxiliary Functions

For every domain D^* we define the functions

Last:	$D^* \rightarrow D$	get last element
Front:	$D^* \rightarrow D^*$	get all but last element
Size:	$D^* \rightarrow$ Int	get number of elements
Fetch:	$D^* \times$ Int $\rightarrow D$	get *i*th element
Store:	$D^* \times$ Int $\times D \rightarrow D^*$	replace *i*th element

The definitions of all but the last two should be obvious. *Fetch* and *Store* are defined as follows:

Fetch(d*, int) = *rec if* int = 1
then *Head*(d*)
else *Fetch*(*Tail*(d*), int - 1)

Store(d*, int, d) = *rec if* int = 1
then *Cons*(d, *Tail*(d*))
else *Cons*(*Head*(d*), *Store*(*Tail*(d*), int - 1, d))

Min_int:	→ Int
Max_int:	→ Int
Max_char:	→ Int
Min_real:	→ Real
Max_real:	→ Real

The values of these constant functions can vary from one system implementation to another; the only restrictions are:

Min_int() < 0

127 ≤ *Max_char*() ≤ 511

Max_char() ≤ *Max_int*()

0.0 < *Min_real*() < 1.0 < *Max_real*()

L_int(int) = *Min_int*() ≤ int ≤ *Max_int*()

L_real(real) = real = 0.0 ∨ *Min_real*() ≤ |real| ≤ *Max_real*()

L_char[[char int]] ≡ 0 ≤ int ≤ *Max_char*()

L_chars(char*) ≡ *rec if Empty*(char*)
then true
else *L_char*(*Head*(char*)) ∧ *L_chars*(*Tail*(char*))

L_string(string) ≡ *L_chars*(string) ∧ *L_int*(*Size*(string))

The size of the string must be a representable integer.

L_state(int, obj*) ≡ *L_int*(int) ∧ *L_int*(*Size*(obj*)) ∧ *L_int*(int + *Size*(obj*) - 1)

The index of each array element and the size of the array must be representable integers.

Approx: Real → Real

Approx defines the imprecision of using finite approximations to real numbers, and can vary from one system implementation to another. The only restrictions placed on *Approx* are as follows. Given that

L_real(real) ∧ *L_real*(real1) ∧ *L_real*(real2)

is true, then

Approx(real) is normalized

Approx(*Approx*(real)) = *Approx*(real)

real1 < real2 implies *Approx*(real1) ≤ *Approx*(real2)

Approx(-real) = -*Approx*(real)

real ≠ 0.0 implies $|(Approx(real) - real) / real| < 10^{-6}$

Approx(0.0) = 0.0

Approx(1.0) = 1.0

Bad_add: Real × Real → Real

Bad_add is used for real additions when the arguments differ in sign. The only restriction placed on *Bad_add* is as follows. If

Bad_add(real1, real2) = real ∧ *L_real*(real)

then

real = *Approx*(real) ∧ $|(real1 + real2 - real)/(real1 + real2)| < 10^{-6}$

R2i: Real → Int

R2i rounds to the nearest integer, and towards zero in case of a tie:

$$|[\![R2i(\text{real}) \bullet 0]\!] - \text{real}| \le 0.5 \ \wedge \ |[\![R2i(\text{real}) \bullet 0]\!]| < |\text{real}| + 0.5$$

Trunc: Real → Int

Trunc truncates its argument towards zero:

$$|[\![Trunc(\text{real}) \bullet 0]\!] - \text{real}| \le 1.0 \ \wedge \ |[\![Trunc(\text{real}) \bullet 0]\!]| \le |\text{real}|$$

Return_bool(bool, env) = *let* d_type = $[\![du_{bool}\ []]\!]$ *in* D_type
in
<normal *in* Term; $[\![$(bool *in* Val): d_type$]\!]$ *in* Obj*; env>

Return_int(int, env) = *let* d_type = $[\![du_{int}\ []]\!]$ *in* D_type
in
<normal *in* Term; $[\![$(int *in* Val): d_type$]\!]$ *in* Obj*; env>

Int_result(int, env) = *if* *L_int*(int)
then *Return_int*(int, env)
else *Error*("overflow" *in* Name, env)

Return_real(real, env) = *let* d_type = $[\![du_{real}\ []]\!]$ *in* D_type
in
<normal *in* Term; $[\![$(real *in* Val): d_type$]\!]$ *in* Obj*; env>

Real_result(real, env) = *if* *L_real*(real)
then *Return_real*(*Approx*(real), env)
else if |real| > *Max_real*()
then *Error*("overflow" *in* Name, env)
else *Error*("underflow" *in* Name, env)

Return_char(char, env) = *let* d_type = $[\![du_{char}\ []]\!]$ *in* D_type
in
<normal *in* Term; $[\![$(char *in* Val): d_type$]\!]$ *in* Obj*; env>

Return_string(string, env) = *let* d_type = $[\![du_{string}\ []]\!]$ *in* D_type
in
<normal *in* Term; $[\![$(string *in* Val): d_type$]\!]$ *in* Obj*; env>

Eq_obj(obj1*, obj2*, env) = *let* <obj1; obj2> = obj2* *to* Obj²
in
Return_bool(obj1 = obj2, env)

Eq_obj will be used in defining various "equal" operations.

Error(name, env) = *let* term = <exit; name> *in* Term
in
<term; nil *in* Obj*; env>

6.5.2 Force

```
force_ = proc [t: type] (x: any) returns (t) signals (wrong_type)
    λ(obj1*, obj2*, env). let obj1 = obj1* to Obj
                              obj2 = obj2* to Obj
                          in
                          if Includes(obj1.val to D_type, obj2.d_type)
                            then <normal in Term; obj2*; env>
                            else Error("wrong_type" in Name, env)
```

Includes is used instead of strict equality because the type **any** is a legal parameter to **force**.

6.5.3 Null

```
equal = proc (n1, n2: null) returns (bool)
        return(true)
        end equal

similar = proc (n1, n2: null) returns (bool)
          return(true)
          end similar

copy = proc (n: null) returns (null)
       return(nil)
       end copy
```

6.5.4 Bool

```
and = proc (b1, b2: bool) returns (bool)
      return(b1 cand b2)
      end and

or = proc (b1, b2: bool) returns (bool)
     return(b1 cor b2)
     end or

not = proc (b: bool) returns (bool)
      if b
        then return(false)
        else return(true)
        end
      end not

equal = proc (b1, b2: bool) returns (bool)
     Eq_obj

similar = proc (b1, b2: bool) returns (bool)
          return(b1 = b2)
          end similar

copy = proc (b: bool) returns (bool)
       return(b)
       end copy
```

6.5.5 Int

```
add = proc (i1, i2: int) returns (int) signals (overflow)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj2
                          in
                          Int_result((obj1.val to Int) + (obj2.val to Int), env)
```

Note that *add* is not a parameterized procedure, so the parameter list obj1* is not used. This will be true of all of the definitions that follow.

```
sub = proc (i1, i2: int) returns (int) signals (overflow)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj²
                          in
                          Int_result((obj1.val to Int) - (obj2.val to Int), env)

mul = proc (i1, i2: int) returns (int) signals (overflow)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj²
                          in
                          Int_result((obj1.val to Int) * (obj2.val to Int), env)

minus = proc (i: int) returns (int) signals (overflow)
        return(0 - i)
           except when overflow: signal overflow end
        end minus

div = proc (i1, i2: int) returns (int) signals (zero_divide, overflow)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj²
                              int          = obj2.val to Int
                          in
                          if int = 0
                             then Error("zero_divide" in Name, env)
                             else Int_result((obj1.val to Int) / int, env)

mod = proc (i1, i2: int) returns (int) signals (zero_divide, overflow)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj²
                              int          = obj2.val to Int
                          in
                          if int = 0
                             then Error("zero_divide" in Name, env)
                             else Int_result((obj1.val to Int) // int, env)
```

```
power = proc (i, e: int) returns (int) signals (negative_exponent, overflow)
        if e < 0 then signal negative_exponent end
        pow: int := 1;
        for j: int in int$from_to(1, e) do
            pow := pow * i
              except when overflow: signal overflow end
            end
        return(pow)
        end power

from_to_by = iter (from, to, by: int) yields (int)
             while by > 0  &  from <= to  |  by <= 0  &  from >= to do
                yield(from)
                from := from + by
                end
             end from_to_by

from_to = iter (from, to: int) yields (int)
          for i: int in int$from_to_by(from, to, 1) do
             yield(i)
             end
          end from_to

lt = proc (i1, i2: int) returns (bool)
     λ(obj1*, obj2*, env). let  <obj1; obj2> = obj2* to Obj²
                                in
                           Return_bool((obj1.val to Int) < (obj2.val to Int), env)

le = proc (i1, i2: int) returns (bool)
     return(i1 < i2  |  i1 = i2)
     end le

ge = proc (i1, i2: int) returns (bool)
     return(i2 <= i1)
     end ge

gt = proc (i1, i2: int) returns (bool)
     return(i2 < i1)
     end gt

equal = proc (i1, i2: int) returns (bool)
     Eq_obj
```

```
similar = proc (i1, i2: int) returns (bool)
          return(i1 = i2)
          end similar

copy = proc (i: int) returns (int)
       return(i)
       end copy
```

6.5.6 Real

```
add = proc (r1, r2: real) returns (real) signals (overflow, underflow)
    λ(obj1*, obj2*, env). let  <obj1; obj2> = obj2* to Obj²
                               real1        = obj1.val to Real
                               real2        = obj2.val to Real
                          in
                          if (real1 < 0.0 ∧ real2 > 0.0) ∨ (real1 > 0.0 ∧ real2 < 0.0)
                            then Real_result(Bad_add(real1, real2), env)
                            else Real_result(real1 + real2, env)

sub = proc (r1, r2: real) returns (real) signals (overflow, underflow)
      return(r1 + -r2)
      except when overflow: signal overflow
             when underflow: signal underflow
             end
      end sub

mul = proc (r1, r2: real) returns (real) signals (overflow, underflow)
    λ(obj1*, obj2*, env). let  <obj1; obj2> = obj2* to Obj²
                          in
                          Real_result((obj1.val to Real) * (obj2.val to Real), env)

minus = proc (r: real) returns (real)
    λ(obj1*, obj2*, env). let  obj = obj2* to Obj
                          in
                          Return_real(-(obj.val to Real), env)
```

```
div = proc (r1, r2: real) returns (real) signals (zero_divide, overflow, underflow)
    λ(obj1*, obj2*, env). let  <obj1; obj2> = obj2* to Obj²
                               real         = obj2.val to Real
                          in
                          if real = 0.0
                             then Error("zero_divide" in Name, env)
                             else Real_result((obj1.val to Real) / real, env)

power = proc (r, e: real) returns (real)
                          signals (zero_divide, complex_result, overflow, underflow)
    λ(obj1*, obj2*, env). let  <obj1; obj2> = obj2* to Obj²
                               real1        = obj1.val to Real
                               real2        = obj2.val to Real
                          in
                          if real1 = 0.0 ∧ real2 < 0.0
                             then Error("zero_divide" in Name, env)
                          else if real1 < 0.0 ∧ ¬(real2 = I2R2i(real2) • 0I)
                             then Error("complex_result" in Name, env)
                          else % the rest is system dependent

i2r = proc (i: int) returns (real) signals (overflow)
    λ(obj1*, obj2*, env). let  obj = obj2* to Obj
                          in
                          Real_result(I(obj.val to Int) • 0I, env)

r2i = proc (r: real) returns (int) signals (overflow)
    λ(obj1*, obj2*, env). let  obj = obj2* to Obj
                          in
                          Int_result(R2i(obj.val to Real), env)

trunc = proc (r: real) returns (int) signals (overflow)
    λ(obj1*, obj2*, env). let  obj = obj2* to Obj
                          in
                          Int_result(Trunc(obj.val to Real), env)

lt = proc (r1, r2: real) returns (bool)
    λ(obj1*, obj2*, env). let  <obj1; obj2> = obj2* to Obj²
                          in
                          Return_bool((obj1.val to Real) < (obj2.val to Real), env)
```

```
le = proc (r1, r2: real) returns (bool)
     return(r1 < r2 | r1 = r2)
     end le

ge = proc (r1, r2: real) returns (bool)
     return(r2 <= r1)
     end ge

gt = proc (r1, r2: real) returns (bool)
     return(r2 < r1)
     end gt

equal = proc (r1, r2: real) returns (bool)
     Eq_obj

similar = proc (r1, r2: real) returns (bool)
          return(r1 = r2)
          end similar

copy = proc (r: real) returns (real)
       return(r)
       end copy
```

6.5.7 Char

```
i2c = proc (i: int) returns (char) signals (illegal_char)
     λ(obj1*, obj2*, env). let  obj  = obj2* to Obj
                                char = [char (obj.val to Int)]
                           in
                          if  L_char(char)
                            then  Return_char(char, env)
                            else  Error("illegal_char" in Name, env)

c2i = proc (c: char) returns (int)
     λ(obj1*, obj2*, env). let  obj          = obj2* to Obj
                                [char int]   = obj.val to Char
                           in
                          Return_int(int, env)
```

```
lt = proc (c1, c2: char) returns (bool)
     return(char$c2i(c1) < char$c2i(c2))
     end lt

le = proc (c1, c2: char) returns (bool)
     return(c1 < c2 | c1 = c2)
     end le

ge = proc (c1, c2: char) returns (bool)
     return(c2 <= c1)
     end ge

gt = proc (c1, c2: char) returns (bool)
     return(c2 < c1)
     end gt

equal = proc (c1, c2: char) returns (bool)
     Eq_obj

similar = proc (c1, c2: char) returns (bool)
          return(c1 = c2)
          end similar

copy = proc (c: char) returns (char)
       return(c)
       end copy
```

6.5.8 String

```
size = proc (s: string) returns (int)
     λ(obj1*, obj2*, env). let  obj = obj2* to Obj
                           in
                              Return_int(Size(obj.val to String), env)

indexs = proc (pat, str: string) returns (int)
         z: int := string$size(pat)
         for i: int in int$from_to(1, string$size(str)) do
            if pat = string$substr(str, i, z)
               then return(i) end
            end
         return(0)
         end indexs
```

```
indexc = proc (c: char, s: string) returns (int)
         return(string$indexs(string$c2s(c), s))
         end indexc

c2s = proc (c: char) returns (string)
    λ(obj1*, obj2*, env). let  obj  =  obj2* to Obj
                          in
                         Return_string(obj.val to Char in Char*, env)

concat = proc (s1, s2: string) returns (string)
    λ(obj1*, obj2*, env). let  <obj1; obj2>  =  obj2* to Obj²
                               string        =  Concat(obj1.val to String,
                                                       obj2.val to String)
                          in
                         if  L_string(string)
                           then  Return_string(string, env)
                           else  Failure(? in String, env)

append = proc (s: string, c: char) returns (string)
         return(s || string$c2s(c))
         end append

fetch = proc (s: string, i: int) returns (char) signals (bounds)
    λ(obj1*, obj2*, env). let  <obj1; obj2>  =  obj2* to Obj²
                               string        =  obj1.val to String
                               int           =  obj2.val to Int
                          in
                         if  1 ≤ int ≤ Size(string)
                           then  Return_char(Fetch(string, int), env)
                           else  Error("bounds" in Name, env)

rest = proc (s: string, i: int) returns (string) signals (bounds)
       return(string$substr(s, i, string$size(s)))
          except when bounds: signal bounds end
       end rest
```

```
substr = proc (s: string, nlow: int, nsize: int) signals (bounds, negative_size)
        if nsize < 0 then signal negative_size end
        z: int := string$size(s)
        if nlow < 1  cor  nlow - 1 > z
          then signal bounds end
        ss: string := ""
        last: int := z
        if nsize < z - nlow + 1
          then last := nlow + nsize - 1 end
        for i: int in int$from_to(nlow, last) do
            ss := string$append(ss, s[i])
            end
        return(ss)
        end substr

s2ac = proc (s: string) returns (ac)
       ac = array[char]
       a: ac := ac$new()
       for c: char in string$chars(s) do
          ac$addh(a, c)
          end
       return(a)
       end s2ac

ac2s = proc (a: ac) returns (string)
       ac = array[char]
       s: string := ""
       for c: char in ac$elements(a) do
          s := string$append(s, c)
          end
       return(s)
       end ac2s

chars = proc (s: string) yields (char)
        for i: int in int$from_to(1, string$size(s)) do
            yield(s[i])
            end
        end chars
```

```
lt = proc (s1, s2: string) returns (bool)
     z1: int := string$size(s1)
     z2: int := string$size(s2)
     min: int := z1
     if z1 > z2 then min := z2 end
     for i: int in int$from_to(1, min) do
        if s1[i] < s2[i] then return(true) end
        end
     return(z1 < z2)
     end lt

le = proc (s1, s2: string) returns (bool)
     return(s1 < s2 | s1 = s2)
     end le

ge = proc (s1, s2: string) returns (bool)
     return(s2 <= s1)
     end ge

gt = proc (s1, s2: string) returns (bool)
     return(s2 < s1)
     end gt

equal = proc (s1, s2: string) returns (bool)
     Eq_obj

similar = proc (s1, s2: string) returns (bool)
          return(s1 = s2)
          end similar

copy = proc (s: string) returns (string)
       return(s)
       end copy
```

6.5.9 Array Types

The heading of the **array** type generator begins

```
array_ = cluster [t: type] is ...
         at = array[t]
```

We will make use of "at" and "t" in the definitions that follow.

```
create = proc (nlow: int) returns (at)
         return(at$[nlow: ])
         end create

new = proc () returns (at)
      return(at$create(1))
      end new

predict = proc (nlow, nsize: int) returns (at)
          return(at$create(nlow))
          end predict

low = proc (a: at) returns (int)
    λ(obj1*, obj2*, env). let  obj          = obj2* to Obj
                               <int; obj3*> = env.arrays(obj.val to Array)
                          in
                          Return_int(int, env)

high = proc (a: at) returns (int)
       return(at$low(a) + (at$size(a) - 1))
       end high

size = proc (a: at) returns (int)
    λ(obj1*, obj2*, env). let  obj          = obj2* to Obj
                               <int; obj3*> = env.arrays(obj.val to Array)
                          in
                          Return_int(Size(obj3*), env)

set_low = proc (a: at, nlow: int)
    λ(obj1*, obj2*, env). let  <obj1; obj2> = obj2* to Obj2
                               array        = obj1.val to Array
                               int1         = obj2.val to Int
                               <int2; obj3*> = env.arrays(array)
                               a_map        = env.arrays[array ← <int1; obj3*>]
                          in
                          if L_state(int1, obj3*)
                            then <normal in Term; nil in Obj*; env[a_map e arrays]>
                            else Failure(3 in String, env)
```

```
trim = proc (a: at, nlow: int, nsize: int) signals (bounds, negative_size)
       if nlow < at$low(a) | nlow > at$high(a) cand nlow > at$high(a) + 1
         then signal bounds end
       if nsize < 0 then signal negative_size end
       while at$low(a) < nlow do
          at$reml(a)
          end
       while at$size(a) >= nsize do
          at$remh(a)
         end
       end trim

fill = proc (nlow: int, nsize: int, elt: t) returns (at) signals (negative_size)
      if nsize < 0 then signal negative_size end
      a: at := at$create(nlow)
      for i: int in int$from_to(1, nsize) do
         at$addh(a, elt)
         end
      return(a)
      end fill

fill_copy = proc (nlow: int, nsize: int, elt: t) returns (at) signals (negative_size)
                                          where t has copy: proctype (t) returns (t)
            if nsize < 0 then signal negative_size end
            a: at := at$create(nlow)
            for i: int in int$from_to(1, nsize) do
               at$addh(a, t$copy(elt))
               end
            return(a)
            end fill_copy

fetch = proc (a: at, i: int) returns (t) signals (bounds)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj²
                              <int1; obj3*> = env.arrays(obj1.val to Array)
                              int2          = (obj2.val to Int) - int1 + 1
                              obj3          = Fetch(obj3*, int2)
                          in
                          if 1 ≤ int2 ≤ Size(obj3*)
                            then <normal in Term; obj3 in Obj*; env>
                            else Error("bounds" in Name, env)
```

```
bottom = proc (a: at) returns (t) signals (bounds)
         return(a[at$low(a)])
            except when bounds: signal bounds end
         end bottom

top = proc (a: at) returns (t) signals (bounds)
      return(a[at$high(a)])
         except when bounds: signal bounds end
      end top

store = proc (a: at, i: int, elt: t) signals (bounds)
    λ(obj1*, obj2*, env). let <obj1; obj2; obj3> = obj2* to Obj³
                              array*             = obj1.val to Array
                              <int1; obj3*>      = env.arrays[array]
                              int2               = (obj2.val to Int) - int1 + 1
                              obj4*              = Store(obj3*, int2, obj3)
                              a_map              = env.arrays[array ← <int1; obj4*]
                          in
                          if 1 ≤ int2 ≤ Size(obj3*)
                            then <normal in Term; nil in Obj*; env[a_map • arrays]>
                            else Error("bounds" in Name, env)

addh = proc (a: at, elt: t)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj²
                              array        = obj1.val to Array
                              <int; obj3*> = env.arrays[array]
                              obj4*        = Append(obj3*, obj2)
                              a_map        = env.arrays[array ← <int; obj4*>]
                          in
                          if L_state(int, obj4*)
                            then <normal in Term; nil in Obj*; env[a_map • arrays]>
                            else Failure(? in String, env)
```

```
addl = proc (a: at, elt: t)
    λ(objl*, obj2*, env). let  <objl; obj2>  = obj2* to Obj*
                               array         = objl.val to Array
                               <intl; obj3*> = env.arrays(array)
                               int2          = intl - 1
                               obj4*         = Cons(obj2, obj3*)
                               a_map         = env.arrays[array ← <int2; obj4*>]
                           in
                          if L_state(int2, obj4*)
                            then <normal in Term; nil in Obj*; env[a_map e arrays]>
                            else Failure(? in String, env)

remh = proc (a: at) returns (t) signals (bounds)
    λ(objl*, obj2*, env). let  obj           = obj2* to Obj
                               array         = obj.val to Array
                               <int; obj3*>  = env.arrays(array)
                               obj4*         = Front(obj3*)
                               obj5*         = Last(obj3*) in Obj*
                               a_map         = env.arrays[array ← <int; obj4*>]
                           in
                          if Empty(obj3*)
                            then Error("bounds" in Name, env)
                            else <normal in Term; obj5*; env[a_map e arrays]>

reml = proc (a: at) returns (t) signals (bounds)
    λ(objl*, obj2*, env). let  obj           = obj2* to Obj
                               array         = obj.val to Array
                               <int; obj3*>  = env.arrays(array)
                               obj4*         = Tail(obj3*)
                               obj5*         = Head(obj3*) in Obj*
                               a_map         = env.arrays[array ← <int; obj4*>]
                           in
                          if Empty(obj3*)
                            then Error("bounds" in Name, env)
                            else <normal in Term; obj5*; env[a_map e arrays]>
```

```
elements = proc (a: at) yields (t)
           for i: int in at$indexes(a) do
              yield(a[i])
              end
           end elements

indexes = proc (a: at) yields (int)
          for i: int in int$from_to(at$low(a), at$high(a)) do
             yield(i)
             end
          end indexes

equal = proc (a1, a1: at) returns (bool)
    Eq_obj

similar = proc (a1, a2: at) returns (bool)
                            where t has similar: proctype (t, t) returns (bool)
          if at$low(a1) ~= at$low(a2) | at$size(a1) ~= at$size(a2)
            then return(false) end
          for i: int in at$indexes(a1) do
             if ~t$similar(a1[i], a2[i]) then return(false) end
             end
          return(true)
          end similar

similar1 = proc (a1, a2: at) returns (bool)
                             where t has equal: proctype (t, t) returns (bool)
           if at$low(a1) ~= at$low(a2) | at$size(a1) ~= at$size(a2)
             then return(false) end
           for i: int in at$indexes(a1) do
              if a1[i] ~= a2[i] then return(false) end
              end
           return(true)
           end similar

copy1 = proc (a: at) returns (at)
        aa: at := at$create(at$low(a))
        for elt: t in at$elements(a) do
           at$addh(aa, elt)
           end
        return(aa)
        end copy1
```

```
copy = proc (a: at) returns (at) where t has copy: prootype (t) returns (t)
       aa: at := at$copyl(a)
       for i: int in at$indexes(aa) do
         aa[i] := t$copy(aa[i])
         end
       return(aa)
       end copy
```

6.5.10 Record Types

The definitions given in this section are schemas. Each definition is given in terms of the record type

$$RT = \textbf{record}[N_1: T_1, ..., N_n: T_n]$$

where the N_i are in increasing lexicographic order.

```
get_N_i = proc (r: RT) returns (T_i)
    λ(obj1*, obj2*, env). let  obj1  = obj2* to Obj
                               obj3* = env.records(obj1.val to Record)
                               obj2  = Fetch(obj3*, i)
                          in
                         <normal in Term; obj2 in Obj*; env>
```

There is a get_N_i operation for every N_i.

```
set_N_i = proc (r: RT, elt: T_i)
    λ(obj1*, obj2*, env). let  <obj1; obj2> = obj2* to Obj²
                               record       = obj1.val to Record
                               obj3*        = env.records(record)
                               obj4*        = Store(obj3*, i, obj2)
                               r_map        = env.records[record ← obj4*]
                          in
                         <normal in Term; nil in Obj*; env[r_map e records]>
```

There is a set_N_i operation for every N_i.

```
equal = proc (r1, r2: RT) returns (bool)
    Eq_obj
```

```
similar = proc (r1, r2: RT) returns (bool)
                        where T1 has similar: proctype (T1, T1) returns (bool),
                              ...
                              Tn has similar: proctype (Tn, Tn) returns (bool)
        return(T1$similar(r1.N1, r2.N1) cand ... cand Tn$similar(r1.Nn, r2.Nn))
        end similar

similar1 = proc (r1, r2: RT) returns (bool)
                        where T1 has equal: proctype (T1, T1) returns (bool),
                              ...
                              Tn has equal: proctype (Tn, Tn) returns (bool)
         return(r1.N1 = r2.N1 cand ... cand r1.Nn = r2.Nn)
         end similar1

copy1 = proc (r: RT) returns (RT)
      return(RT${N1: r.N1, ..., Nn: r.Nn})
      end copy1

copy = proc (r: RT) returns (RT) where T1 has copy: proctype (T1) returns (T1),
                                       ...
                                       Tn has copy: proctype (Tn) returns (Tn)
       rr: RT := RT$copy1(r)
       rr.N1 := T1$copy(r.N1)
       ...
       rr.Nn := Tn$copy(r.Nn)
       return(rr)
       end copy
```

6.5.11 Oneof Types

The definitions given in this section are schemas. Each definition is given in terms of the oneof type

$$OT = \text{oneof}[N_1: T_1, \ldots, N_n: T_n]$$

where the N_i are in increasing lexicographic order.

```
make_N_i = proc (T_i) returns (OT)
    λ(obj1*, obj2*, env). let  obj    = obj2* to Obj
                               oneof  = [N_i: obj1]
                          in
                          <normal in Term; [(oneof in Val); OT] in Obj*; env>
```

We have informally used OT as a d_type. There is a make_N_i operation for every N_i.

```
is_N_i = proc (x: OT) returns (bool)
        tagcase x
          tag N_i: return(true)
          others: return(false)
          end
        end is_N_i

value_N_i = proc (x: OT) returns (T_i) signals (wrong_tag)
            tagcase x
              tag N_i (v: T_i): return(v)
              others: signal wrong_tag
              end
            end value_N_i

equal = proc (x1, x2: OT) returns (bool)
                          where T_1 has equal: proctype (T_1, T_1) returns (bool),
                                ...
                                T_n has equal: proctype (T_n, T_n) returns (bool)
        tagcase x1
          tag N_1 (v: T_1): return(v = OT$value_N_1(x2))
          ...
          tag N_n (v: T_n): return(v = OT$value_N_n(x2))
          end
          except when wrong_tag: return(false) end
        end equal
```

```
similar = proc (x1, x2: OT) returns (bool)
                          where T1 has similar: proctype (T1, T1) returns (bool),
                                ...
                                Tn has similar: proctype (Tn, Tn) returns (bool)
        tagcase x1
          tag N1 (v: T1): return(T1$similar(v, OT$value_N1(x2)))
          ...
          tag Nn (v: Tn): return(Tn$similar(v, OT$value_Nn(x2)))
          end
          except when wrong_tag: return(false) end
        end similar

copy = proc (x: OT) returns (OT) where T1 has copy: proctype (T1) returns (T1),
                                       ...
                                       Tn has copy: proctype (Tn) returns (Tn)
      tagcase x1
        tag N1 (v: T1): return(OT$make_N1(T1$copy(v)))
        ...
        tag Nn (v: Tn): return(OT$make_Nn(Tn$copy(v)))
        end
      end copy
```

6.5.12 Procedure Types

The definitions given in this section are schemas. Each definition is given in terms of a procedure type T.

```
equal = proc (r1, r2: T) returns (bool)
    λ(obj1*, obj2*, env). let <obj1, obj2> = obj2* to Obj2
                          in
                          Return_bool(obj1.val = obj2.val, env)
```

We ignore the type descriptors because one or both descriptors can be routine.

```
similar = proc (r1, r2: T) returns (bool)
        return(r1 = r2)
        end similar

copy = proc (r: T) returns (T)
      return(r)
      end copy
```

6.5.13 Procedure Types

The definitions given in this section are schemas. Each definition is given in terms of a procedure type T.

```
equal = proc (r1, r2: T) returns (bool)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj²
                          in
                          Return_bool(obj1.val = obj2.val, env)
```

We ignore the type descriptors because one or both descriptors can be routine.

```
similar = proc (r1, r2: T) returns (bool)
          return(r1 = r2)
          end similar

copy = proc (r: T) returns (T)
       return(r)
       end copy
```

6.5.14 Routine

We will informally use routine in CLU text below, even though one cannot reference the special type routine in this way.

```
equal = proc (r1, r2: routine) returns (bool)
    λ(obj1*, obj2*, env). let <obj1; obj2> = obj2* to Obj²
                          in
                          Return_bool(obj1.val = obj2.val, env)
```

We ignore the type descriptors because one or both descriptors can be a procedure or iterator type.

```
similar = proc (r1, r2: routine) returns (bool)
          return(r1 = r2)
          end similar
```

```
copy = proc (r: routine) returns (routine)
       return(r)
       end copy
```

7. Conclusions

The goal of this thesis was to develop a precise, formal definition of CLU, with the hope that such a definition would be a useful tool in other work. This hope has already been met in three ways. First, this research has provided us with the opportunity to evaluate various features of CLU from a new viewpoint. Although our understanding of the meaning of CLU programs has not really changed, we did discover several places where our understanding of legality-checking was faulty or incomplete, and changes were made to CLU as a result. Second, our definition is now being used to verify (informally) the correctness of the legality-checking phase of the CLU compiler. Third, our definition provides a basis for evaluating the usefulness of the definition method proposed by Schaffert[Schaffert78a]. We discuss each of these points briefly below, and then finish with some directions for future research.

7.1 Meaning

Our definition is divided into two major parts, one defining what constitutes a legal module and one defining the meaning of legal modules. If we ignore the definition of the built-in abstractions then the meaning part is quite short, less than 12 pages of actual equations. This indicates that the model of computation CLU presents to the user is simple, uniform, and hence easily understood. It also indicates that the definition method we have chosen is an apt one. Most of the mechanisms used in the meaning part are direct "implementations" of the informal descriptions given in [Liskov78], particularly those for variables, parameters, and exception handling. For example, we have used explicit termination conditions to propagate exceptions through intermediate computations, rather than introducing continuations to effect transfers of control.

However, our treatment of iterators is not completely satisfying. Although the way we have defined iterators is simple, and appears to be the simplest possible given the underlying theory, it is somewhat backwards from the usual view that an iterator actually yields items to the caller. Having the iterator return a continuation to the caller, as discussed in section 2.2.4, is perhaps a truer representation of this view, but defining

iterators in this way would involve changing the definitions of all the other statements to create and use continuations. If some other construct of CLU forced us to use continuations, we might feel less strongly about avoiding their use for iterators; we might even use continuations for exception handling. In the absence of such a construct, the use of continuations seems overly complicated.

In defining meaning we have omitted two aspects of real systems. The first omission is input/output primitives. We have not defined any I/O primitives because implementors are free to choose their own, although a standard has been proposed [Liskov78]. However, I/O is essentially trivial to incorporate into our definition. Aside from simply adding definitions of the primitive types and routines, the only changes to our definition would be the addition of one or more unique id domains and state mappings to represent new primitive mutable objects (such as streams).

The other omission is more subtle, and concerns failures in real systems. In an actual implementation it is permissible for the system to generate a *failure* exception at any time. For example, when attempting to multiply two real numbers, an exception such as

failure("floating point hardware broken")

could be generated. We could represent this by making *E_invoke* nondeterministically choose between executing normally and failing with a randomly chosen string, but in real systems the choice is deterministic. Since the exact points at which such failures can occur and the exact reasons for such failures are completely implementation-dependent, we have chosen simply to omit such failures from our definition.

7.2 Legality

In contrast to the concise definition of meaning, our definition of legality is rather long. This is primarily just an indication of the large amount of error-checking being performed; for the most part the definition is very straightforward. However, two aspects of CLU merit comment here: interface specifications and parameters.

Our legality-checking algorithm basically performs a single pass over the syntax tree. The only exception is that two passes are necessary to check module headings. Although clever schemes could be used to minimize the work performed on the second pass, there does not appear to be any good way to avoid the second pass. In particular, we know of no simple, sensible rule that avoids the problem without unduly restricting the class of legal modules. Of course, an algorithm is not bad merely because it requires two passes, but it took a substantial effort to determine and understand the minimum assumptions necessary to derive and check interface specifications, and this leaves us with the feeling that perhaps something better could (and should) be done.

On the subject of parameters, we wish to point out by way of two examples that in some respects there is a very thin line between having a well-defined mechanism and an ill-defined one. The first example deals with the permissible types of parameters. CLU is now defined so that parameters can only be declared with the types **type**, **null**, **bool**, **int**, **real**, **char**, and **string**. However, at one point procedure and iterator types, and certain oneof types, were also allowed. These types were removed when we discovered that one could write "legal" generators that could never be instantiated, as in

```
X = cluster [t1: oneof[a: t2], t2: oneof[a: t1]] is ...
```

(Finite oneof type specifications cannot be written for any actual parameters.)

This problem could be "solved" by allowing such generators (even though they could never be used), or by imposing rules to forbid cyclic dependencies in parameter declarations, but it appears that no computational power is lost by eliminating structured type specifications. Indeed, given the dynamic behavior of CLU objects, it is difficult to think of many uses for non-type parameters in general, much less procedures, iterators, and oneofs. For example, the type of an array does not include any information about the size of the array because an array can vary dynamically in size.

Our second example deals with parameterized operations of clusters. Having decided that parameterized procedures and iterators were useful as individual modules, the designers felt that parameterized operations should also be allowed for completeness, though no real use for such operations is yet known. It then made sense to allow specific

instantiations of parameterized operations to be listed in **where** clause restrictions. Unfortunately, we discovered that this last step led to an infinite recursion in our legality-checking algorithm when applied to certain pathological modules [Scheifler77]. For example, the type X[X[t]] in

```
X = cluster [t: type] is f, ...  where t has f[X[X[t]]]: proctype ();
    ...
    f = proc [u: type] (); ... end f;
    end X;
```

is legal only if X[t] is a legal parameter to X, which is true only if X[t] has an operation f[X[X[X[t]]]], which is true only if X[X[X[t]]] is a legal type, which leads to an infinite recursion.

This problem could be solved by giving full specifications for each parameterized operation. (This is the approach taken in Alphard [Wulf76], though apparently for syntactic uniformity rather than in response to this problem.) For example, one could state

```
where t has op = proc [u: type] (u) returns (u)  where u has ...
```

instead of

```
where t has op[int]: proctype (int) returns (int),
            op[real]: proctype (real) returns (real)
```

Since this solution requires additional syntax for an extremely rare case, an additional legality rule was formulated instead. (The function *L_parms* performs this check in our definition.)

7.3 Compiler Verification

With respect to legality-checking, no distinction can be made between "denotational" definitions and "operational" definitions; there are only differences in the precise algorithm used. Our definition of legality-checking was done with compiler verification in mind, which is why a (basically) one pass algorithm was chosen. Although the actual implementation of legality-checking in the CLU compiler differs from our definition in a number of respects (primarily for efficiency reasons), the overall algorithm is identical and equivalence is fairly easy to establish.

Second, Schaffert uses a producer-consumer pair of processes to define the execution of an iterator and its invoking for statement. A special object is used to represent the queue between the producer and the consumer; this object is suitably defined so that the two processes are constrained to execute alternately rather than simultaneously. Although this definition is closer than ours to the informal description of iterators, we feel it is more complex and harder to understand. However, we should note that while our method of definition can also be done in Schaffert's formalism, the opposite is not really true. This indicates that Schaffert's formalism should be superior to denotational methods for defining parallel programming languages.

Third, the concept of mutable objects is built into Schaffert's formalism; an explicit representation of the universal state is not present in the equations, but rather is implicit throughout. Although this could tend to simplify the equations, another factor offsets this benefit. Specifically, a list of active variables (the arguments to the computation) is passed explicitly from computation to computation. In our opinion, passing around the kind of execution environment we have defined is no more complicated and no harder to deal with than passing around a list of variables.

In summary, the CLU definition given here appears to be a little better than the definition given by Schaffert. Except in the treatment of variables and iterators, however, the differences between the two definitions are not significant, and really have nothing to do with the underlying theories. Furthermore, Schaffert's method appears to be much better than current denotational techniques for defining programming languages that deal with parallelism.

7.5 Directions for Future Research

One of the most important tasks left in defining object-oriented languages is the development of a clean, usable, axiomatic definition method. An axiomatic method should not be extremely difficult to devise, but no such method seems to exist. However, a proposed method will be given in [Schaffert78b] with which a straightforward axiomatic definition of CLU should be possible.

Our definition of meaning, on the other hand, was designed to be as simple as possible, with the result that it is not particularly well-suited as an implementation standard. Our view here is that our definition is well-suited to be the standard definition of CLU, but that for verifying the correctness of implementations and for proving CLU programs correct, other formal definitions should be constructed.

7.4 Comparison to Schaffert's Work

There are major differences between our definition and the one done by Schaffert [Schaffert78a], but for the most part they are differences in algorithm design, and have nothing to do with which underlying theory was used. Schaffert's legality-checking algorithm is multi-pass, and simply assumes the existence of legal interface specifications in the library rather than incorporating an initial pass to derive and install them. Transformations of the parse tree are separated out as much as possible from the actual legality checks. Further, CLU has changed somewhat since the time his definition was made: the **exit** and **continue** statements were added, as was the type **real**; the rule mentioned above concerning infinite recursion was adopted; and a mechanism for renaming exceptions was removed. These and other differences result in Schaffert's definition being a little shorter than ours, but in practice his definition of legality seems slightly harder to understand, primarily due to the separation of the transformations from the legality checks.

The two definitions of meaning are the same in many respects, particularly in the treatment of parameters and exception handling, but there are three important differences. First, Schaffert is constrained by his formalism to treat variables as objects of an abstract type, whereas we are free to treat them simply as names for objects. Although treating variables as objects gives a uniformity and simplicity to the underlying semantics, it is counter to way CLU variables are normally explained [Liskov78]. It may be that treating variables as objects yields a simpler axiomatic description, but proving equivalence to an axiomatic definition would not appear to be any harder with our definition than with Schaffert's.

In addition, a better understanding of how to define general control structures is needed. Neither Schaffert's definition of iterators nor ours is wholly satisfactory. Perhaps future work in defining parallel programming language constructs will lead to some answers.

References

Blum73 — Blum, E.K. SEMANOL: a formal system for the semantics of programming languages, *TRW Software Series 73-05*. TRW Systems Group, Redondo Beach, Ca. (August 1973)

Church51 — Church, A. The calculi of lambda-conversion, *Annals of Math. Studies 6*. Princeton University Press, Princeton (1951)

Donahue76 — Donahue, J.E. Complementary definitions of programming language semantics, *Lecture Notes in Computer Science, 42*. Springer-Verlag, Berlin (1976)

Hoare69 — Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM, 12, 10* (October 1969), 576 - 583

Hoare74 — Hoare, C.A.R., and Lauer, P.E. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica, 1* (1972), 271 - 281

Knuth68 — Knuth, D.E. Semantics of context-free languages. *Math. Systems Theory, 2, 2* (June 1968), 127 - 145)

Ledgard77 — Ledgard, H.F. Production systems: a notation for defining syntax and translation. *IEEE Trans. on Soft. Eng., SE-2, 2* (March 1977), 105 - 124

Liskov77a — Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. *Comm. ACM, 20, 8* (August 1977), 564 - 576

Liskov77b — Liskov, B., and Snyder, A. Structured exception handling, *Computation Structures Group Memo 155*, Laboratory for Computer Science, M.I.T., Cambridge, Ma. (December 1977)

Liskov78 — Liskov, B., et al. *CLU Reference Manual*. Laboratory for Computer Science, M.I.T., Cambridge, Ma. (forthcoming)

Milne76 — Milne, R., and Strachey, C. *A Theory of Programming Language Semantics*. Halsted Press, John Wiley, New York (1976)

Schaffert78a — Schaffert, C. A formal definition of CLU, *Technical Report 193*. Laboratory for Computer Science, MIT, Cambridge, Ma. (1978)

Schaffert78b — Schaffert, C. Specifying meaning in object-oriented languages. Ph.D. Thesis, MIT, Cambridge, Ma. (forthcoming)

Scheifler77 — Scheifler, R.W. Type parameters and infinite recursion, *CLU Design Note 65*. Laboratory for Computer Science, MIT, Cambridge, Ma. (November 1977)

Scott70 — Scott, D.S. Outline of a mathematical theory of computation. *Proc. of the Fourth Annual Princeton Conf. on Information Science and Systems*, Princeton (1970), 169 - 176

Scott71a — Scott, D.S., and Strachey, C. Toward a mathematical semantics for computer languages. Proc. Symp. on Computers and Automata, *Microwave Res. Inst. Symp. Series, 21*, Brooklyn Polytechnic Inst. (1971) 19 - 46

Scott71b — Scott, D.S. Continuous lattices, *Technical Memo PRG-7*. Programming Research Group, University of Oxford (1971)

Scott72 — Scott, D.S. Lattice theory, data types and formal semantics. *N.Y.U. Symp. on Formal Semantics*, Prentice-Hall (1972) 64 - 106

Scott76 — Scott, D.S. Data types as lattices. *SIAM J. Computing 5, 3* (September 1976) 522 - 587

Stoy77 — Stoy, J. Denotational semantics: the Scott-Strachey approach to programming language theory. MIT Press, Cambridge, Mass. (1977)

Strachey73 — Strachey, C. Varieties of programming languages, *Technical Memo PRG-10*. Programming Research Group, University of Oxford (1973)

vanWijngaarden76 — van Wijngaarden, A., et al. *Revised Report on the Algorithm Language Algol 68*. Springer-Verlag, Berlin (1976)

Wadsworth73 — Wadsworth, C., and Strachey, C. Continuations: a mathematical semantics for handling full jumps, *Technical Memo PRG-11*. Programming Research Group, Oxford University (1973)

Wegner72 — Wegner, P. The Vienna Definition Language. *Computing Surveys, 4, 1* (March 1972), 5 - 63

Wulf78 — Wulf, A. (editor). An informal definition of Alphard, *CMU-CS-78-105*. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa. (February 1978)

Appendix - Parsing

We use an extended BNF grammar to define the relationship between the abstract syntax and text strings. The general form of a production is:

```
nonterminal   ::= alternative
                | alternative
                | ...
                | alternative
```

The following extensions are used:

a , ...	stands for	$(a \mid a\ ,\ a \mid a\ ,\ a\ ,\ a \mid \cdots)$
$\{a\}$	stands for	$(\epsilon \mid a \mid a\ a \mid a\ a\ a \mid \cdots)$
$[a]$	stands for	$(\epsilon \mid a)$

All semicolons are optional in CLU, but for simplicity they appear below without enclosing meta-brackets. Nonterminal symbols appear in normal face. Reserved words appear in bold face. All other terminal symbols are non-alphabetic and appear in normal face.

In defining the syntax of expressions, we will simply indicate the precedence of the various operators with comments rather than explicitly incorporating precedence into the productions. Higher precedence operations are performed first. All binary operators are left associative except **, which is right associative.

```
full_module ::= { equate } module

module      ::= procedure
              | iterator
              | cluster

procedure   ::= idn = proc [ parms ] args [ returns ] [ signals ] [ where ] ;
                    body
                    end idn ;
```

```
iterator    ::= idn = iter [ parms ] args [ yields ] [ signals ] [ where ] ;
                    body
                    end idn ;

cluster     ::= idn = cluster [ parms ] is idn , ... [ where ] ;
                    cluster_body
                    end idn ;

parms       ::= [ decl , ... ]

args        ::= ( [ decl , ... ] )

decl        ::= idn , ... : type_spec

returns     ::= returns ( type_spec , ... )

yields      ::= yields ( type_spec , ... )

signals     ::= signals ( cond_spec , ... )

cond_spec   ::= name [ ( type_spec , ... ) ]

where       ::= where restriction , ...

restriction ::= idn has oper_decl , ...
              | idn in type_set

type_set    ::= { idn | idn has oper_decl , ... ; { equate } }
              | idn

oper_decl   ::= oper_name , ... : type_spec

oper_name   ::= name [ [ constant , ... ] ]

constant    ::= expression
              | type_spec

body        ::= { equate } { statement }
```

```
cluster_body ::= equate { equate }
                 routine { routine }

routine      ::= procedure
               | iterator

equate       ::= idn = constant ;
               | idn = type_set ;
               | rep = type_spec ;

type_spec    ::= null
               | bool
               | int
               | real
               | char
               | string
               | any
               | rep
               | cvt
               | type
               | array [ type_spec ]
               | record [ field_spec , ... ]
               | oneof [ field_spec , ... ]
               | proctype ( [ type_spec , ... ] ) [ returns ] [ signals ]
               | itertype ( [ type_spec , ... ] ) [ yields ] [ signals ]
               | idn [ constant , ... ]
               | idn

field_spec   ::= name , ... : type_spec
```

```
statement   ::= decl ;
              | idn : type_spec := expression ;
              | decl , ... := invocation ;
              | idn , ... := invocation ;
              | idn , ... := expression , ... ;
              | invocation ;
              | primary . name := expression ;
              | primary [ expression ] := expression ;
              | if expression then body
                  { elseif expression then body }
                  [ else body ]
                  end ;
              | while expression do body end ;
              | return [ ( expression , ... ) ] ;
              | yield [ ( expression , ... ) ] ;
              | signal name [ ( expression , ... ) ] ;
              | exit name [ ( expression , ... ) ] ;
              | break ;
              | continue ;
              | begin body end ;
              | tagcase expression
                  tag_arm { tag_arm }
                  [ others : body ]
                  end ;
              | for [ decl , ... ] in invocation do body end ;
              | for [ idn , ... ] in invocation do body end ;
              | statement except { when_arm }
                                 [ others_arm ]
                                 end ;
```

```
tag_arm     ::= tag name , ... [ ( idn : type_spec ) ] : body

when_arm    ::= when name , ... [ ( decl , ... ) ] : body
              | when name , ... ( * ) : body

others_arm  ::= others [ ( idn : type_spec ) ] : body

expression  ::= primary
              | ( expression )
              | ~ expression                    % 6    (precedence)
              | - expression                    % 6
              | expression ** expression        %  5
              | expression // expression        %   4
              | expression / expression         %   4
              | expression * expression         %   4
              | expression || expression        %     3
              | expression + expression         %     3
              | expression - expression         %     3
              | expression < expression         %       2
              | expression <= expression        %       2
              | expression = expression         %       2
              | expression >= expression        %       2
              | expression > expression         %       2
              | expression ~< expression        %       2
              | expression ~<= expression       %       2
              | expression ~= expression        %       2
              | expression ~>= expression       %       2
              | expression ~> expression        %       2
              | expression & expression         %         1
              | expression cand expression      %         1
              | expression | expression         %           0
              | expression cor expression       %           0
```

```
primary     ::= nil
              | true
              | false
              | int_literal
              | real_literal
              | char_literal
              | string_literal
              | type_spec $ name [ [ constant , ... ] ]
              | idn [ constant , ... ]
              | idn
              | invocation
              | type_spec $ { field , ... }
              | type_spec $ [ [ expression : ] [ expression , ... ] ]
              | force [ type_spec ]
              | up ( expression )
              | down ( expression )
              | primary . name
              | primary [ expression ]

invocation  ::= primary ( [ expression , ... ] )

field       ::= name , ... : expression
```

Reserved word: one of the identifiers appearing in bold face in the syntax. Upper and lower case letters are not distinguished in reserved words.

Name, idn: a sequence of letters, digits, and underscores that begins with a letter or underscore, and that is not a reserved word. Upper and lower case letters are not distinguished in names and idns.

Int_literal: a sequence of one or more decimal digits.

Real_literal: a mantissa with an (optional) exponent. A mantissa is either a sequence of one or more decimal digits, or two sequences (one of which may be empty) joined by a period. The mantissa must contain at least one digit. An exponent is 'E' or 'e', optionally

followed by '+' or '-', followed by one or more decimal digits. An exponent is required if the mantissa does not contain a period. Real literals are parsed to their (normalized) exact values.

Char_literal: either a "printing" ASCII character (octal 40 thru octal 176) other than single quote or backslash, enclosed in single quotes, or one of the following escape sequences enclosed in single quotes:

escape sequence	character
\'	' (single quote)
\"	" (double quote)
\\	\ (backslash)
\n	NL (newline)
\t	HT (horizontal tab)
\p	FF (newpage)
\b	BS (backspace)
\r	CR (carriage return)
\v	VT (vertical tab)
***	specified by octal value (* is an octal digit)

The escape sequences may be written using upper case letters. The ASCII characters in their usual order correspond to octal values 0 through 177.

String_literal: a sequence of zero or more character representations, enclosed in double quotes. A character representation is either a printing ASCII character other than double quote or backslash, or one of the escape sequences listed above.

Comment: a sequence of characters that begins with a percent sign, ends with a newline character, and contains only "printing" ASCII characters and horizontal tabs in between.

Separator: a "blank" character (space, vertical tab, horizontal tab, carriage return, newline, form feed) or a comment. Zero or more separators may appear between any two tokens, except that at least one separator is required between any two adjacent non-self-terminating tokens: reserved words, identifiers, integer literals, and real literals.

Domain Index

Function Index

C_cvt_conds	111	Cond_spec* × CE → Sig* × CE
C_cvt_decls	110	Decl* × CE → Expr* × CE
C_cvt_type	111	Type_spec × CE → D_type × Bool × CE
C_cvt_types	111	Type_spec* × CE → D_type* × Bool* × CE
C_decls	75	Decl* × CE → CE
C_defs	94	Decl* × Equate* × CE → CE
C_defs1	94	Equate* × Decl* × Equate* × CE → CE
C_du_parms	54	DU × Obj* × CE → Obj × CE
C_d_cond	65	D_cond × Handle* → Bool
C_d_conds	64	D_cond* × Handle* → Bool
C_d_oper	62	D_oper × CE → Expr × CE
C_elseifs	78	Elseif_arm* × CE → Elseif* × CE
C_equates	72	Equate* × CE → CE
C_exit	81	Name × D_type* × Handle* → Bool
C_expression	61	Expression × CE → Expr × CE
C_expressions	61	Expression* × CE → Expr* × CE
C_fields	66	Field* × CE → Comp* × CE
C_field_specs	47	Field_spec* × CE → D_comp* × CE
C_full_modules	104	Full_module* × DU_spec* × CE → Mod* × CE
C_head_conds	99	Cond_spec* × CE → D_cond* × CE
C_head_decls	98	Decl* × CE → CE
C_head_type	98	Type_spec × CE → D_type × CE
C_head_types	98	Type_spec* × CE → D_type* × CE
C_invocation	64	Invocation × CE → Expr × CE
C_iter_inv	85	Invocation × CE → Expr × CE
C_iter_type	49	D_type* × D_type* × D_cond* × CE → D_type × CE
C_module	105	Module × CE → Mod × CE
C_oper_decls	97	Oper_decl* × CE → Op_decl* × CE
C_oper_names	97	Oper_name* × D_type × CE → Op_decl* × CE
C_op_call	69	Name × Expression+ × CE → Expr × CE
C_op_decls	56	D_type × Op_decl* × CE → Bool
C_op_inv	69	Name × Expression+ × CE → Expr × CE
C_op_parms	54	Op_spec × Obj* × CE → Obj × CE
C_op_type	58	Name × Obj* × Op_spec* × CE → D_type
C_others_arm	87	Others_arm × CE → Catch × CE
C_parms	55	Obj* × D_parm* × CE → Bool
C_parm_decls	95	Decl* × CE → CE
C_parm_op	58	Name × Obj* × Op_decl* → D_type

C_proc_type	48	D_type* × D_type* × D_cond* × CE → D_type × CE
C_restriction	96	Restriction × CE → CE
C_restrictions	95	Restriction* × CE → CE
C_routine	108	Routine × CE → Op_form × CE
C_routines	107	Routine* × CE → Oper* × CE
C_sig	80	Name × D_type* × Sig* → Bool × Bool*
C_statement	75	Statement × CE → Stmt × CE
C_statements	75	Statement* × CE → Stmt* × CE
C_tag_arm	83	Tag_arm × CE → Tag × Name* × D_comp* × CE
C_tag_arms	83	Tag_arm* × CE → Tag* × Name* × D_comp* × CE
C_type_set	96	Type_set × CE → Tset × CE
C_type_spec	45	Type_spec × CE → D_type × CE
C_type_specs	45	Type_spec* × CE → D_type* × CE
C_when_arm	86	When_arm × CE → Catch × Name* × Handle × CE
C_when_arms	86	When_arm* × CE → Catch* × Name* × Handle* × CE
C_xbody	84	Decl* × Body × CE → D_type* × Unit × CE
Delist	25	[D*]* → D*
Derive_specs	88	Full_module* × CE → DU_spec* × CE
Duplicates	48	D* → Bool
Empty	25	D* → Bool
Eq_obj	131	Obj* × Obj* × Env → Result
Erase	106	Idn* × CE → CE
Error	131	Name × Env → Result
E_catch	124	Catch × Name × Obj* × Env → Result
E_catchs	124	Catch* × Name × Obj* × Env → Result
E_elseifs	120	Elseif* × Unit × Env → Result
E_equate	73	Equate × CE → CE
E_equates	73	Equate* × CE → CE
E_expr	114	Expr × Env → Result
E_exprs	114	Expr* × Env → Result
E_invoke	115	Invoke × Env → Result
E_op	126	Op_form × Obj* × Obj* × Env → Result
E_stmt	119	Stmt × Env → Result
E_stmts	119	Stmt* × Env → Result
E_tag	122	Tag × Obj × Env → Result
E_tags	122	Tag* × Name × Obj × Env → Result
E_unit	119	Unit × Env → Result
Fail	86	→ D_cond

Failure	115	String × Env → Result
Fetch	128	D* × Int → D →
Find_op	125	Name × Oper* → Op_form
Fix_info	75	Idn* × Info_map × CE → Idn* × CE
Front	128	D* → D*
Get_comps	66	Name* × Expr → Comp*
Get_decls	77	Idn* × CE → D_type* × CE
Get_du	105	Idn × CE → DU
Get_du_specs	89	Module* × CE → DU_spec*
Get_d_comps	47	Name* × D_type → D_comp*
Get_d_parms	95	Idn* × CE → D_parm*
Get_exprs	110	Idn* × Bool → Expr*
Get_ext_specs	93	Op_spec* × Idn* → Op_spec*
Get_mod_idn	89	Module → Idn
Get_op	116	Obj × Env → Op × Obj*
Get_op_idn	91	Routine → Idn
Get_op_specs	92	Routine* × Idn* × CE → Op_spec* × CE
Get_op_type	57	D_oper × CE → D_type
Get_r_spec	93	Routine × CE → R_spec × CE
Get_spec	90	Module × CE → DU_spec × DU_spec × CE
Get_specs	90	Full_module* × DU_spec* × CE → DU_spec* × CE
Has_bad	45	D_type* → Bool
Head	25	D* → D
Include	71	D_type* × D_type* → Bool
Includes	71	D_type × D_type → Bool
Includes_all	70	D_type × D_type* → Bool
Init_specs	89	Module* × CE → CE
Install_op_spec	107	Op_spec × CE → CE
Install_spec	104	DU_spec × CE → CE
Integer	70	Expr × CE → Bool
Int_result	130	Int × Env → Result
Itertype_op	116	D_iter × Name → Op
Itertype_op_specs	57	D_iter → Op_spec*
Last	128	D* → D
L_char	129	Char → Bool
L_chars	129	Char* → Bool
L_int	128	Int → Bool
L_parms	101	Obj* × CE → Bool

Real_result	130	Real × Env → Result
Record_op	116	D_record × Name → Op
Record_op_specs	57	D_record → Op_spec*
Replace	99	Idn* × Idn* × D → D
Restore	74	CE × CE → CE
Result_types	70	Invoke × CE → D_type*
Return_bool	130	Bool × Env → Result
Return_char	130	Char × Env → Result
Return_int	130	Int × Env → Result
Return_real	130	Real × Env → Result
Return_string	131	String × Env → Result
Return_types	59	D_type → D_type*
Routine_op	116	Name → Op
Routine_op_specs	58	→ Op_spec*
Same_constraints	100	Constraint* × Constraint* → Bool
Same_op_specs	101	Op_spec* × Op_spec* → Bool
Same_r_spec	99	R_spec × R_spec → Bool
Same_size	25	D* × D* → Bool
Same_spec	99	DU_spec × DU_spec → Bool
Same_t_spec	100	T_spec × T_spec → Bool
Set_op_names	91	Routine* × CE → CE
Size	128	D* → Int
Store	128	D* × Int × D → D*
Subst	60	Obj* × Idn* × D → D
Tail	25	D* → D*
Trunc	130	Real → Int
Types_of	70	Expr* × CE → D_type*
Type_of	70	Expr × CE → D_type
Unassn	119	Idn* × Env → Env
Yield_types	85	D_type → D_type*

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 10/26/95

Report # LCS-TR-201

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 176 (183-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-176) UNNUMBERED TITLEPAGE, 2-175 UNNUMBERED BLANK
(177-183) SCANCONTROL, COVER, SPINE, PRINTER'S NOTES, TRGT'S (3)

Scanning Agent Signoff:

Date Received: 10/26/95 Date Scanned: 11/6/95 Date Returned: 11/9/95

Scanning Agent Signature: Michael W. Cobb

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**