CSL Notebook Entry

To	CSL Cedar Reviewers	Date	January 16, 1981
From	Cedar Language Features Design Committee	Location	PARC/CSL, Palo Alto
Subject	Cedar Mesa Version 6T5	Files	[Ivy]<CedarDocs>Lang>6T5> *.bravo, Design.press

XEROX

Archive ID: yyCSL-xxx

Attributes: technical, Cedar, Mesa, Programming research

References: [Ivy]<CedarDocs>Lang>6C2>Design.press (February 15, 1980)
Mesa Language Manual, Version 5.0, CSL-79-3, April 1979
[Iris]<Mesa>Doc>Compiler60.press (October 27, 1980)

Abstract: This document reports the current state of the design and implementation of the Cedar Mesa language. It is a snapshot prepared in mid-January 1981 for the Cedar Internal Design Review. To provide some background, the motivation and required properties of the language are briefly described. Specific changes to Mesa fall into three major areas: safe language, delayed binding, and exported types. The currently implemented language, called Mesa 6T5, is specified below. Mesa 6T5 is a partial implementation of Mesa 6C2; working notes on the status of the still-missing features and on open design issues are also included.

TABLE OF CONTENTS

INTRODUCTION

The design for the Mesa 6C2 language evolved through considerable discussion, taking into account reactions to several design documents circulated to CSL and the Cedar Interest group. The Cedar External Design Review in September 1979 provided additional feedback. Mesa 6T5 is the currently implemented step toward 6C2. We have focused upon early implementation of features essential for the construction of Cedar itself. Our design and implementation work subsequent to 6C2 has also suggested minor revisions and, in a few cases, raised some as yet unresolved issues regarding the 6C2 design. These issues, as well as projections for implementation progress, are discussed below.

Cedar Mesa is very closely related to *current Mesa*, as implemented and supported by SDD. This document assumes knowledge of Mesa Version 6.0. In fact, a number of proposals originally made in 6C2 have already appeared in Mesa 6 and are not described in detail here.

Probably the largest single extension is the provision of garbage-collected dynamic storage (and a concomitant reduction in the role of heap storage). We intend that most code will be written in the *safe subset*, which imposes a number of restrictions to ensure the safe operation of the garbage collector and introduces some new (safe) features to make these restrictions tolerable. The full (unsafe) language is generally "upward compatible" with Mesa 6.0.

We have generated many further ideas for the improvement of Cedar Mesa and have received large numbers of suggestions. However, we believe that the present design addresses the most pressing issues in (at least) a satisfactory manner, and that it is more important to continue with the timely implementation of this design than to refine it further.

The remainder of this introduction summarizes the required properties of Cedar Mesa and briefly enumerates the specific changes and additions that we propose for the langauge. Next we provide an overview of our progress. The final section is a detailed description of the language changes that are currently implemented or about to be implemented.

SAFE LANGUAGE

Motivation

A desirable property of a high-level language is *implementation independence*, which means that the effects of each program are explicable in terms of the *language* (rather than its implementation) even if the program is erroneous. Mesa comes rather close to meeting this goal (as evidenced by the fact that most Mesa programs can be debugged "at the Mesa level," without ever worrying about the format of stack frames or the details of storage management), but it does contain some "unsafe" features whose use can lead to messy implementation dependencies.

It would be desirable to reduce implementation dependencies in Cedar on general grounds. However, the decision to include facilities for automatic storage deallocation ("garbage collection") within Cedar Mesa makes a shift in this direction imperative. The collector can cause storage to be deallocated (permitting its subsequent reallocation and re-use) at times that are completely unpredictable from examination of the source program. A single programming error that smashes a reference used by the collector can effectively bring the whole system down in a rubble of bits, from which it is difficult to reconstruct any evidence of the cause of the crash.

One of the major goals of the Cedar Language Features Design Committee has been to design a subset of Mesa for which it is safe to do automatic storage deallocation. To first approximation, the "safe subset" is that part of the language where even incorrect programs cannot interfere with the reliable operation of the collector.

We intend that the vast majority of programs will be written primarily in the safe subset. However, we recognize that it is not presently feasible to provide acceptably efficient substitutes for all uses of Mesa's unsafe features. Instead, we provide textual means for indicating that some regions of a program are *unchecked*. This will inhibit compiler enforcement of the safety restrictions and simultaneously indicate that the programmer has assumed the additional responsibility of ensuring that these regions of the program cannot violate the integrity of the system.

Invulnerability, Safety, and Checking

An obviously desirable property of a programming system is that no user programming error can reduce its world to a rubble of bits. We will call this property *invulnerability*. In general, it can be ensured only by maintaining the integrity of certain data structures known to the runtime system (e.g., dispatch vectors, core maps). Collectively, the properties that must be maintained to ensure invulnerability are called the safety *invariants;* each portion of the system is responsible for ensuring that they are not destroyed and is free to assume that the rest of the system does likewise.

Invulnerability is not a local property. If any part of the system fails to maintain the invariants, the entire system (including regions that are themselves correct) is potentially vulnerable. We use the term *safety* for the property that the invariants cannot be invalidated locally (even by incorrect programs). *Checked* program regions are portions of a system where safety is guaranteed by language-enforced restrictions. Even *unchecked* regions are supposed to preserve the invariants, but the guarantee must be provided by the programmer, rather than by the language and its implementation. If a system consists entirely of checked regions (and the invariants hold initially), then by induction the program is invulnerable. However, an error in an unchecked region could make even the checked regions vulnerable. Thus the checked/unchecked boundary limits responsibility but not vulnerability.

Unchecked regions may use unsafe language features, and errors may cause violation of system invariants. These regions are sometimes called "trusted regions," because confidence in the correct operation of the system must be based on trust that they are free of such errors. By contrast, our confidence that errors in checked regions will not cause system crashes is based on the knowledge that the safety restrictions have been mechanically enforced.

Type Confusion

A breach of the type system causes a safety problem if it allows mis-typing of some piece of memory (i.e., if the target type contains an address, such as a reference or a procedure value); other uses of loopholes introduce implementation dependencies but do not threaten safety. We have formalized the distinction between these two sorts of breaches. We call the former loopholes; the latter, puns. Loopholes are not allowed in the checked language.

Narrowing

Our design introduces a number of new type distinctions into Mesa, frequently leading to a number of separate but closely related types. It is often desirable to *coerce* a value of one of these types into a value of a related type. If there is a guarantee that no information can be lost by such a conversion, we call it a *widening*. Widening is performed automatically whenever demanded by context this is analogous to lengthening an INTEGER to a LONG INTEGER. In general, conversion in the other direction requires a runtime check to ensure that information is not being lost. For each such *narrowing* described below, we indicate the nature of the necessary check; if the check fails, an error is raised.

References and Garbage Collection

Although Mesa pointers are nominally typed, they provide a rich source of opportunities for type confusion, including but not limited to the classical "dangling pointer" problem, where a pointer persists after the storage it refers to has been deallocated. We are introducing a new class of *reference* types that refer to garbage collected (automatically deallocated) storage.

VAR Parameters

One common use of pointers in current Mesa is to pass parameters "by reference," rather than "by value" either to allow the results of assignments to the formal parameters to be visible in the calling context or to avoid the cost of copying large arguments. Mesa 6C2 provides a special parameter attribute, VAR, for this purpose. In checked regions, the "address of" operator, @, is only allowed in the construction of arguments for VAR parameters. This restriction greatly simplifies and reduces the cost of the storage reclamation mechanism.

DELAYED BINDING

Motivation

A desirable property of a high-level language is that it allow a wide range of *binding times*: that is, it should allow the programmer maximal control over when the attributes of a particular variable are determined, with different choices not requiring changes at each occurrence of the variable's identifier. Examples of attributes include type, storage allocation method, implementation (of abstract objects), and actual value; examples of binding times include program-writing time, compilation, program initialization, block entry, and dynamic assignment. Generally speaking, deferring the binding of an attribute leads to greater generality in the program at the cost of decreased static checkability and (often) runtime efficiency.

Experience with languages like Lisp and Smalltalk, which provide only very late binding, suggests that even though late binding may actually be required only in few situations and have significant runtime cost, there are certain kinds of programs that are extremely clumsy to write if late binding of type or implementation is not available. In particular, programming tools (debuggers, performance monitors) and knowledge representation systems seem to be of this nature. Thus, in addition to wanting more binding time flexibility in Cedar Mesa on general grounds, we believe such flexibility will greatly simplify the writing of an important class of programs.

Dynamic typing

Current Mesa provides very limited variability in the binding time of an object's type. Variant records allow a deferred choice between specific enumerated alternatives, and sequences and array descriptors allow deferring the choice of an object's length. Otherwise, all types must be fixed at compile time. This requirement makes it impossible to avoid LOOPHOLEs and *ad hoc* type tagging schemes when writing schedulers, searchers, printers, etc. that must operate on objects of unpredictable type. Our solution to this problem requires two new mechanisms: a runtime representation for types, and a way to associate a type with an object at runtime that is guaranteed consistent with the type system established at compile time. Note that we explicitly embrace the traditional view that an object's type is inherent in the object itself (rather than the Russell view that its type is a syntactic property of the variable through which it is referenced).

Runtime Types

In Mesa 6C2, we have kept the current limitation that types used in declarations must be compile-time constants. Cedar provides several library packages that manipulate and interpret the runtime representations of types. We have extended Mesa to allow conversion from a type constant to an ordinary Mesa value that is understood by these packages.

Atoms and Lists

The languages, such as Lisp that provided our models for delayed binding derive part of their power from the easy availability of lists and atoms and from the flexible packages that can be built around them. We have extended Mesa by adding a new type constructor, LIST OF, for conveniently creating list types, and a predeclared type ATOM.

EXPORTED TYPES AND OBJECTS

Motivation

Current Mesa systems are constructed by binding together collections of separately written and compiled modules that communicate through explicit and precisely specified interfaces. Specifications of operations, which appear in interfaces, are decoupled from their implementations, which are hidden within program modules. In versions of Mesa prior to Mesa 6, specifications of types did not have similar properties; all information about types used in interfaces appeared within the interface definitions themselves and had to be fixed before any client code could be compiled. This shortcoming violated the principle of information hiding and also created awkward compilation dependencies. Exported types were introduced in Mesa 6 to address this problem, but they impose a number of restrictions and are insufficiently flexible for some applications.

Support of the so-called "object-oriented" style of programming, as developed in such languages as Simula and (especially) Smalltalk, is an important goal of Cedar. Attempts to use Mesa for such programming particularly suffer from the inability to hide representational detail. In addition, Mesa does not distinguish between abstract and concrete types. The result is that dealing with an abstract class of objects with multiple and coexisting concrete implementations requires considerable circumlocution; the code is awkward and error-prone.

Opaque Types

Types may be exported from implementors into interfaces, just as procedures are. An interface may contain a declaration of an *opaque type*, the structure of which is largely unspecified and thus invisible to importers of that interface. Every implementation of the interface must supply a fully specified *concrete type* for each opaque type in the interface, as well as a set of procedures implementing the operations of that type.

We propose changing the type checking rules of Mesa to avoid type confusion even when several distinct concrete types correspond to a single opaque type.

Notational Conventions

The notation used for object-oriented programming in Mesa depends upon the degree of generality anticipated by the programmer. If only one implementation is anticipated, he uses standard functional notation, e.g., Interface.Op[object, args]. If multiple implementations are anticipated, an object-oriented notation, e.g., object.Op[args] is more appropriate. We introduce conventions that allow these notations to be used interchangeably, given suitable declarations. This allows the programmer to adopt a uniform style and thereby to delay decisions about representation.

Generic Types

Opaque types do not handle the situation in which multiple implementations of an abstract type coexist within a system and clients wish to use the various implementations interchangeably.

We would like to extend the opaque type mechanism to allow the dynamic intermixing of different implementations and thereby accommodate a more general style of object-oriented programming. For each opaque interface type, we introduce a *generic type*, the values of which can have any of the concrete types supplied for the opaque type. Such a value is represented by a value of the concrete type plus an interface providing its operations.

Subclasses

Sometimes several classes of objects share a number of abstract operations; these can all be viewed as subclasses of some other class for which just the common operations are defined. Algorithms applicable to the superclass automatically extend to all the subclasses. Part of our design is a means for defining an interface as an extension of other interfaces. The effects of a subclassing can be achieved if one of the interfaces being extended includes an opaque type.

CURRENT STATUS

This section summarizes the current (mid-January 1981) state of our design and implementation. In general, we have focused upon early implementation of those features thought essential for the construction of Cedar itself; we are aware of a number of temporary inelegances and loose ends that have resulted from this approach. We have also been influenced by our desire to remain compatible with Mesa 6.0 and to take advantage of the system software supplied by SDD for as long as possible. A certain number a changes or new features with straight forward implementations have been deferred simply to avoid compatibility problems.

Note also that Mesa 6.0 adopted, at least in part, a number of the additions to Mesa proposed in 6C2. These include the ability to control the initialization of a variable according to its type, generic NIL, zones with NEW and FREE operations, sequence types, and a restricted form of opaque types.

Descriptions of restricted and unimplemented features in 6T5 are repeated as fine points in the section *Language Changes*, which we intend to use as an interim programming manual for Cedar Mesa.

SAFE LANGUAGE

We have placed highest priority upon implementing automatic storage management. We believe that this is the most important single change proposed in 6C2 and that its performance will be critical to the success of Cedar Mesa. Implementation involves not only the garbage collector itself but also a fairly pervasive set of changes to the compiler, both to gather the information needed by the collector and to generate modified code for counted operations. By focusing upon an early implementation, we have been able to identify the areas critical to performance, to demonstrate the feasibility of garbage collection in Mesa and to begin experimenting with alternative strategies and algorithms.

We have been more concerned with understanding safety than enforcing it. We believe that we understand the system invariants and the restrictions to be applied to checked regions, and the following section documents them. The 6T5 compiler does not yet support checked regions, and most of the checks required to identify safety violations remain unimplemented. In essence, *all* code is unchecked; users of Mesa 6T5 must maintain the required invariants themselves and, where possible, heed the restrictions that will be applied to safe regions. We have found that this is not a major problem for current Cedar implementors, but this area obviously requires attention before Cedar receives widespread use. We see no serious problems in this area.

References and Other Address-Containing Types

Automatic Deallocation and Reference Counting

Mesa 6T5 supports frame, heap and collectible (counted) storage. A debugged garbage collector with reasonable (and improving) performance is part of its runtime support. The language provides references (addresses to be traced by the garbage collector) as proposed in 6C2, and the compiler generates code to initialize and update counted references in the manner required by the collector.

In the current implementation, the static links embedded within procedure and signal values are not treated as counted references. We have arranged to give global frames "infinite" reference counts.

Thus assigning procedures or signals declared at the module level to collectible storage cannot create dangling references in 6T5 (unless the programmer explicitly destroys the global frame). We plan to detect assignments of nested procedures that are potentially "out of scope" when we implement enforcement of the safety restrictions in checked regions. Proper retention and subsequent collection of retained local frames will be provided in conjunction with the implementation of VAR parameters.

Variants of Reference Types and Values

Mesa 6T5 recognizes the distinction between counted and uncounted references and provides a universal reference type (REF ANY). The runtime system implements both the prefix and quantum encodings of the types of universal references.

We have chosen to defer relative counted references. Relative references with bases other than MDS or with offsets packed into fields smaller than a word, have a high implementation cost and may be deferred indefinitely.

Bases and Zones

Mesa 6T5 does not support BASE types (although the runtime does); at the language level, they are needed only to describe relative references, which we are deferring. The ZONE types of Mesa 6.0 have been extended to allow counted zones; the runtime provides standard implementations of zones with a variety of attributes.

VAR Parameters

We have not yet implemented VAR parameters, and Mesa 6T5 retains the Mesa 6.0 interpretation of @ and DESCRIPTOR. VAR parameters and parameters with procedure types provide a mechanism by which the retention of a local frame can require retention of other local frames in its call chain. We have devoted considerable effort to working out several designs that deal with this related set of issues. These designs differ in their performance trade-offs; we plan to choose the simplest and to implement it.

Persistent Closures

Mesa 6T5 does not have persistent closures (but global frames are forced to be persistent). We plan to implement VAR parameters and retained frames at the same time, as described above.

Other Safety Changes

PUN and NARROW

We do not currently enforce the distinction between safe and unsafe type breaches, and the current langauge does not recognize PUN. We plan to remedy these defects when we implement checked regions.

In Mesa 6T5, NARROW may be applied only to values with type REF ANY, and the NARROW operation must be explicit.

Variant Records

The various loopholes associated with variant records have been long-standing problems. We have a design (inspired by Euclid) that appears to be sound. Unfortunately, it is semantically incompatible with Mesa 6.0; certain existing constructs are given a subtly different interpretation. The 6T5 version of Cedar Mesa does not implement the new design. The runtime system does, however, deal with part of the safety problem by disallowing any assignment to an RC variant record in collectible storage that would change the tag of that record.

Strings, Text, and Sequence Types

Sequence types were introduced by Mesa 6.0, and its restrictions on sequence usage are retained in 6T5. We do, however, guarantee that all elements of RC sequences are set to NIL upon allocation.

We have provided string constants (TEXT literals) that are compatible with collectible storage; our attempts to build more elaborate string packages have raised several langauge issues regarding the proper treatment of the attribute READONLY (and possibly of a new CONST attribute).

DELAYED BINDING

Type Values

The 6C2 design proposes making types full-fledged values, with the restriction that type expressions appearing in declarations must be compile-time constants. It also suggests unifying the syntax of expressions and type expressions. While we believe that both of these changes are desirable, we have deferred them, primarily because of the syntactic problems that arise without some (incompatible) changes to the syntax of type expressions.

Our interim solution provides a bracketing operator (CODE) that allows a type expression to stand as an ordinary expression. This operator both avoids the syntactic problems and generates the representation of the type that is assumed by the Cedar runtime. We have not yet implemented the similar operator TYPE, proposed in 6C2 for extracting the type attribute of an arbitrary variable.

Dynamic typing

Universal references (type REF ANY) have been implemented essentially as proposed in 6C2. We have redesigned the language form for discriminating a universal reference so that the value is always copied to a discriminated variable before it is used.

Atoms and Lists

Mesa 6T5 implements atoms and lists essentially as proposed in 6C2.

EXPORTED TYPES AND OBJECTS

Most of the 6C2 proposals related to exported types and objects have been deferred or withdrawn for reconsideration. We still believe that these are important areas that must be addressed, and we intend to propose revisions or replacements of the 6C2 features. Our current thinking builds upon some ideas about polymorphism, but we do not currently have a complete and coherent design. Furthermore, any implementation would appear to require a substantial revision of the Mesa 6 implementation framework from which 6T5 was derived. This is an area of ongoing research.

Opaque Types

The opaque types in Mesa 6T5 are identical to the exported types of Mesa 6. In particular, guaranteeing type safety still requires the restriction that, within any system, there must be a unique concrete type for each opaque type. We would like to allow different implementations to supply different concrete types and to modify the type checking rules to prevent unintentional mixing of different concrete types. Work on this area has been deferred; there is considerable interaction with issues raised by system modelling.

Notational Conventions

The proposed convention for object-oriented notation has been implemented only for objects with types that are (references to) opaque types. A more general implementation requires further understanding of the interaction between types defined in interfaces and the imported instances of such interfaces. There is considerable interaction with system modelling.

Generic Types and Subclasses

The specific 6C2 proposals for generic types and subclasses have been withdrawn for reconsideration. Our design for polymorphism, if successful, may dominate those proposals.

LANGUAGE CHANGES

The changes described here are for Mesa 6T5 and its immediate successors. Omission or deferral of a feature does not imply it will not be included in some future extension.

SAFE LANGUAGE

General Principles

System Invariants

Basically, the invariants that must be preserved are

> the integrity of the code and the various data structures maintained by the present runtime system for storage management, process scheduling, control links, etc.
>
> the integrity of the new data structures introduced for garbage collection (reference counts, symbol table information, etc.).
>
> the type integrity of all RC storage.
>
> the accessibility of all active *reference-containing (RC)* storage to the collector by tracing links, starting from a known set of locations.

An elaboration of the last two points, precisely stating the invariants required for safe garbage collection, appears in the 6C2 document.

Checked and Unchecked Regions

We believe that we understand the restrictions to be applied to checked regions and their implications for the compiler. Users of Mesa 6T5 should heed the restrictions described below in their programming. The 6T5 compiler generates range, bounds and NIL checks by default but does not otherwise enforce the restrictions or check for violations.

Each block of a program will belong to a *checked* or *unchecked* region, with checked being the default. An unchecked region may contain embedded checked regions, but not conversely. Our goal is to have checked regions predominate, and to make it easy for the reader to locate all unchecked regions.

We have added the key words CHECKED and UNCHECKED to the language, to precede the BEGIN of a block. Unless a module body's block begins with UNCHECKED, it is automatically checked. Unless a nested block in an unchecked block begins with CHECKED, it is automatically unchecked. UNCHECKED may not appear within a checked block.

The key words CHECKED and UNCHECKED are not recognized in 6T5.

Within a checked block, all the restrictions imposed on the safe subset will be enforced by the compiler (possibly generating some runtime checks for such things as array bounds). In unchecked regions, the full language may be used, but the programmer is responsible for ensuring that he both understands and preserves all the system invariants. Even in unchecked regions it is likely that the bulk of most programs can be in the safe subset, and features outside the subset should be used only for good reasons and with due caution, since an error in an unchecked region can bring down all of Cedar.

Unchecked Regions Providing Safe Interfaces

Although a checked module is not allowed to contain any unchecked regions, it may *import* interfaces to unchecked modules. We distinguish between unchecked procedure bodies and unsafe procedure calls. A procedure that uses unsafe features to accomplish some safe operation (i.e., one that will not violate the safety invariants, even in the presence of errors in the calling program) has a *safe interface*, and such procedures may be called from within checked regions. Of course, the compiler cannot guarantee that an unchecked procedure body actually provides a safe interface; this is the responsibility of the programmer who declares the interface to be safe.

The key words SAFE and UNSAFE are new access attributes. Preceding a procedure, port, process, or signal type constructor, SAFE indicates an interface that may be used within checked regions. Unsafe interfaces are private to unchecked regions, and invocation from checked regions is prohibited. Interfaces declared in checked regions are automatically safe, while those declared in unchecked regions are unsafe unless declared SAFE.

The key words SAFE and UNSAFE are not recognized in 6T5.

Signals are like procedures: checked regions are not allowed to raise unsafe signals, and unchecked regions may only catch safe signals with safe catch phrases, i.e., those whose statement is a block that is preceded by SAFE. Note that this SAFE applies to the *interface*. If the block is to be a checked region, it must be preceded by CHECKED (which implies SAFE). An UNWIND, however, may be caught anywhere, because the region that raised it is being terminated.

References and Other Address-Containing Types

AC Types

We need to impose restrictions on the creation and use of values of *address-containing (AC)* types, which are either *reference-containing (RC)* or *pointer-containing (PC)*.

References (addresses) are AC.

String, procedure, frame, descriptor, signal, port, and list types implicitly contain addresses and are therefore AC.

Counted AC types (the default) are RC and are the primary concern of the collector.

Uncounted AC types (including pointer types) are PC.

Arrays whose component type is RC (PC) are RC (PC).

Records with any RC (PC) fields are RC (PC).

In 6T5, there are no RC descriptor or port types and no PC list types. Procedure and signal types are RC by default but are treated as PC by the compiler; i.e., creating dangling references by assigning procedure values out of scope remains possible. See the discussion of persistent closures.

Whenever an RC value is created, all its address fields must be validly initialized (possibly to NIL) to avoid assigning through, tracing, or deallocating, non-existent structures. If an RC type has variants, the tag fields of the corresponding values (sequences or variant records) must also be initialized. In Cedar Mesa, all address fields of RC types that are not explicitly initialized will be implicitly initialized to NIL (or to a non-NULL default value specified in the corresponding type definition).

In Mesa 6T5, procedure and signal variables do not have automatic initialization to NIL.

To free the garbage collector from concern for PC values, pointers may not refer to RC values; i.e., if T is RC, then POINTER TO T is not a valid type. Conversely, PC types may not be declared or used in checked regions.

We must also guard against both "forgery" of AC values and the preservation of RC values in locations unknown to the collector. The immediate implication is that we must prohibit LOOPHOLE and restrict UNSPECIFIED in checked regions.

> As long as there is no mechanism for later converting them back to AC values, there is no *safety* problem in allowing AC values (indeed, any values) to be interpreted as non-AC values, and we provide PUN as the safe replacement for LOOPHOLE.
>
> In checked regions, values of type UNSPECIFIED will not conform to any AC type, although values of AC types may be coerced to UNSPECIFIED. REF ANY provides type-carrying references, which can safely be used to replace many uses of UNSPECIFIED.

Automatic Deallocation and Reference Counting

Cedar Mesa provides three kinds of storage:

> Frame: This is storage that is implicitly allocated upon procedure activation to hold variables declared in the corresponding scope. It is also implicitly deallocated normally on scope exit, although other cases will be discussed below.
>
> Heap: This is storage that is explicitly allocated by NEW and deallocated by (unsafe) FREE statements. Heap storage is referenced by *pointers*, which may declared and used only in unchecked regions.
>
> Collectible: This is storage that is explicitly allocated by NEW, and is implicitly deallocated after there are no more accessible references to it. FREE applied to a collectible reference will cause it (and its RC fields) to be "NILed out," but the object will only be freed when no other references to it remain. Collectible storage is accessed through *references*, which are described below.

For all three types of storage, the smallest unit of deallocation is an entire allocated object.

Mesa 6.0 provides the first two kinds of storage. Freeing the programmer from the necessity of explicitly deallocating storage (and the possibility of making mistakes in such deallocation) has been one of the major goals in the design of Cedar Mesa. We expect the efficiency of automatic deallocation ("garbage collection") to be good enough that collectible storage will replace most present uses of heap storage.

Variants of Reference Types

Cedar Mesa provides a new type constructor, REF, for creating AC types. REF types provide indirect access to objects and generalize the POINTER types of current Mesa. Each such type includes a distinguised value NIL that designates no object. If the type of x is REF T and its value is not NIL, x^ designates a variable of type T. The READONLY attribute may be used in constructing REF types; if the type of x is REF READONLY T, x^ cannot be used as the target of an updating operation. The representation of a REF value is a (virtual) address.

Two aspects of each reference are part of its type:

Counting option = {counted, uncounted}: The default for AC types is counted; however, in unchecked regions, UNCOUNTED may be applied to any AC type to produce the corresponding PC type; LONG POINTER TO is synonymous with UNCOUNTED REF.

Mesa 6T5 does not recognize COUNTED or UNCOUNTED as an attribute of REF. A type constructed using REF is always RC; one using POINTER, always PC. The Mesa 6 operator POINTER TO constructs a PC type that is an UNCOUNTED MDS RELATIVE REF; relative reference types are not otherwise supported in 6T5 (see below).

@ produces an uncounted reference, and hence cannot be used in checked regions (except when creating a VAR parameter, as explained below).

Uncounted references should either refer to heap storage, which is freed explicitly, or be "redundant," i.e., refer to objects that are guaranteed to be kept in existence by other (counted) references.

Objects containing pointers (uncounted references) may only be declared and used in unchecked regions. Furthermore, pointers may not refer to objects whose type is RC (although counted references may in unchecked regions refer to PC objects).

Static typing = {specific, any}: Cedar Mesa does not allow "universal" variables whose type is not known statically. However, *universal references* may be specified by REF ANY, which indicates that the value will be a reference whose type is determined dynamically. REF abbreviates REF ANY.

The only operations on universal references are allocation and deallocation, extraction of the current value's type, discrimination on the current type (in a SELECT statement as described below), narrowing to a particular reference type, assignment (including parameter passing), and test for equality. A REF READONLY ANY may only be narrowed to a particular READONLY reference type. Note that undiscriminated universal references may *not* be dereferenced.

Counted references of any specific type may be widened to universal references. REF T conforms freely to REF ANY or REF READONLY ANY, and REF READONLY T conforms freely to REF READONLY ANY. Narrowing a universal reference to a specific reference type is subject to a runtime type identity check. See the section on delayed binding for further details of the use of universal references.

The type REF ANY is RC; Mesa 6T5 does not support PC universal references.

The 6C2 proposal also defines a *packing* attribute for references, use of which declares causes the values of the corresponding type to be relative to a specified base. Mesa 6T5 does not provide counted relative references and continues to support the Mesa 6.0 version of uncounted relative references.

We plan to implement (counted) MDS RELATIVE REF types. Other relative reference types, particularly those packed into less than 16 bits, have a high implementation cost compared to the expected benefits and may be deferred indefinitely.

Variants of Reference Values

One aspect of the representation and use of references is a property of particular reference values and not part of their type:

Dynamic type encoding mechanism = {unencoded, prefix, quantum}: This is an aspect of the runtime representation of references that is of concern only to the storage management and reclamation subsystem. From the zone into which a reference points, it is possible to determine its type encoding mechanism.

When collectible storage is allocated by NEW, its type is recorded by either the prefix word or the quantum map mechanism; in the former, an extra word containing the type is allocated just before the object, in the latter, the object is allocated with an address in a range containing only objects of that type, so the high-order part of the address implies the type. Uncounted (heap) references are not required to conform to this scheme and may have the unencoded attribute.

The dynamic type encoding mechanism for a zone is specified when it is created. Generally, prefix is preferable for large objects, and quantum for numerous or small ones.

Zones

The ZONE type constructor specifies *heap* or *collectible* management (i.e., explicit or automatic deallocation), with the default being collectible. Heap zones are specified by the adjective UNCOUNTED as in Mesa 6; the type of a counted zone is simply ZONE.

The system provides a standard counted zone, which is the default zone for allocation of collectible storage and which uses the prefixed type representation. Other zone values are obtained by calls on standard library packages specifying the desired attributes of the zone.

A constructor-like notation for generating such calls appears in the 6C2 proposal. Mesa 6T5 does not provide such notation.

The 6C2 proposal also defines a BASE type constructor. BASE types are to be used for limiting the storage available to zones and for defining RELATIVE REF types. Mesa 6T5 does not have BASE types; however, the runtime system does provide base objects to serve the first of these functions.

VAR Parameters

@ is frequently used in Mesa programs to pass parameters "by reference," rather than the "by value" default, either because assignments to the formal parameter should affect the argument, or to avoid the overhead of copying the argument. In Cedar Mesa we raise this usage to the level of a language feature by introducing a VAR attribute for formal parameters.

Operationally, specifying a parameter to be VAR is equivalent to adding REF to its type, and following each use of its name by ^. Note that this implies that the procedure does not have access to the reference created for the argument, but only to the value referenced; in particular, the procedure cannot either change the reference or copy it into a reference variable. It is this restriction that allows VAR parameters to be implemented safely at a lower cost than unrestricted references.

A formal parameter declared to be DESCRIPTOR is similar except that an argument may have any index type that is a subrange of the index type declared for the formal, and the length of the argument is passed with the reference. The argument may be an explicit descriptor (in unchecked regions) or the result of applying DESCRIPTOR to an array or sequence variable.

@ may be used in checked regions only to create VAR arguments. That is, it may appear only in arguments corresponding to VAR formal parameters. Similarly, the operator DESCRIPTOR may be used in checked regions only to create DESCRIPTOR arguments.

Mesa 6T5 does not have VAR parameters and retains the Mesa 6.0 interpretation of DESCRIPTOR. Adequate and detailed designs of several possible implementations, with differing performance trade-offs, have been worked out and we expect to add VAR parameters.

The "Pointer to" Operator, @

In unchecked regions, @ may be applied to any variable to produce a pointer that refers to it, and DESCRIPTOR may be applied to any array or sequence variable to produce a (PC) descriptor.

> As in current Mesa, @ may be applied to a component of a (record, sequence, or array) value only if it is word-aligned.

It is a safety requirement to be ensured by the unsafe language programmer that these pointers not *dangle*:

> A heap object that (has a component that) is the referent of an accessible pointer may not be freed.
>
> A procedure may not return leaving any accessible pointers to its local variables.
>
> If @ or DESCRIPTOR is applied to (a component of) a collectible object, a reference to the entire object must remain accessible for at least as long as the pointer does.

Reference Assignment

For safety in the presence of multiple processes, reference assignment (including any required reference-counting transactions) is made into an atomic (non-interruptible) operation.

NEW

Cedar Mesa extends the NEW and FREE operations to deal with collectible storage. We view NEW as an operation associated with a zone. As in Mesa 6.0, the argument of NEW is a type, with an optional initial value, and its result is a reference with attributes determined by the zone. The NEW operation of a (counted) ZONE returns REFs, with the type encoding specified for the zone, while the NEW operation of an UNCOUNTED ZONE returns LONG POINTERs.

NEW's value is a reference that points to a properly-initialized object of the argument type. Each zone's allocator must ensure that the storage for a newly allocated object does not overlap with any other currently allocated storage. The new object is in the specified zone, and the reference is counted or uncounted, depending on the zone. The default zone is a counted prefix zone automatically supplied by the system.

When creating an immutable object of some variant record type, the argument type should be fully discriminated (using "adjective" notation). When creating a mutable object (disallowed in the checked language; see below), the argument type should not be discriminated.

As in Mesa 6.0, explicit initialization of sequence elements cannot be specified in the argument of NEW; however, all elements are set to NIL if the sequence is RC.

Examples:

... NEW[T] ...	--Returns a REF T in systemZone
... zone1.NEW[T] ...	--Returns a REF T in zone1
... zone1.NEW[TEXT[ln]] ...	--Returns a REF TEXT of length ln in zone1

FREE

Cedar Mesa includes the FREE statement of Mesa 6.0. FREE will usually be applied to pointers, which are allowed only in unchecked regions. (It would be unsafe to free storage if pointers to it had been passed to checked regions, since there is no way to ensure that copies of pointers will not be kept; unchecked regions must be "trusted," anyhow.)

Storage should only be freed if all code that has access to it is trusted. Although this is a strong restriction, and makes the use of pointers much less attractive, we have not found any acceptable alternative restrictions that make it possible to explicitly and safely free storage. This is the cost of allowing pointer assignments.

FREE may also be applied to counted references, but in this case serves only as a hint to the allocator that the programmer believes that this is the last accessible reference. FREE's argument (which is passed as a VAR) and any component RC fields of the object it refers to will be reset to NIL, but the storage will only be freed when no accessible references to it remain. This is a safe and useful way to break circular structures.

Persistent Closures

Mesa currently allows local procedure values to be assigned to global procedure variables and to be returned as the results of procedures. However, if such a local procedure references non-local variables, the result can be arbitrary, since the frames referenced via the its static links may have been deallocated on return from its enclosing procedures. Rather than outlaw the technique, we will make it safe in Cedar Mesa. Procedure values, static links and VAR parameters will be treated as references, and frames will (conceptually) be reference counted like other collectible objects.

Mesa 6T5 does not have persistent closures; however, detailed designs of several possible implementations have been worked out and we expect to add them. There are interactions with the design of VAR parameters.

Since procedure values are RC but treated as PC in 6T5, it currently is possible to create dangling references by assigning such values "out of scope." However, global frames are treated as collectible objects with infinite reference counts, however, and procedures declared at the module level can safely be assigned to variables in collectible storage.

Other Safety Changes

Bounds and NIL Checking

The Mesa compiler has options for generating bounds checking for all assignments to subrange variables and all indexing operations, and NIL checking for all dereferencing operations. In Cedar Mesa, the compiler will generate checks (except those it can prove redundant) for all indexing and dereferencing operations in checked regions that are used to access RC types or to store values, whether these options are selected or not.

In Mesa 6T5, bounds and NIL checks are selected by default but are not mandatory.

PUN and NARROW

Although LOOPHOLE is prohibited in checked regions, there is no safety problem in allowing one type to be interpreted as another as long as

both types occupy the same amount of storage, and

each AC field of the target type conforms freely to the corresponding field in the source type.

An opaque type from an unsafe interface may not be the target type of a PUN, since the concrete type is potentially PC.

If these conditions are met, PUN may be used as a type transfer function, even in checked regions. Like LOOPHOLE, it merely causes a reinterpretation of the type of a value, it does not cause any code to be generated.

Mesa 6T5 does not recognize the key word PUN.

NARROW, on the other hand, causes a runtime check that a value known statically to be of some general type (e.g., REF ANY), actually contains a value of a specified type. If the test fails, the signal RangeError is generated. The type need not be specified explicitly if it can be statically inferred from the context; in such cases the explicit NARROW is optional (unless a catch phrase for the signal

is to be attached).

Widening is always implicit, and supplied automatically when demanded by context.

In Mesa 6T5, NARROW may be applied only to values with type REF ANY. The NARROW operation must be explicit, but the target type may be omitted when it can be inferred from context.

Variant Records

COMPUTED and OVERLAID variant record types provide another way to achieve the effect of a LOOPHOLE. Such types may be used in checked regions only if each of the variants may be PUNned into each of the others, i.e., if confusion about the variant cannot change the interpretation of the RC fields.

In current Mesa, even ordinary variant record types with tag fields provide opportunities for type confusion within the scope of a discriminating selection. Although Mesa prohibits direct assignment to the tag field, there are a number of underhand ways of causing either the tag or the identity of the designated record to change, such as calling a procedure that changes a global variable or accessing the record through a different name (aliasing). Even worse, such actions may be triggered by interrupts even if the selection appears perfectly innocuous. Changing the language to prohibit all such possibilities would have required a rather more thorough redesign than we were prepared to attempt.

We distinguish two different sorts of unbound variant record types: those where the variant is *immutable*, i.e., fixed when the object is allocated, and those where the variant is *mutable*, i.e., changeable by assignment to the value. If T is a variant record type, unqualified use of T denotes the former case, and ANY T the latter. Mutable variant records may not be used in checked regions.

T values may be freely assigned to ANY T variables; assignment in the reverse direction requires a run-time check that the current tags are equal. However, REF T values (and REF bound T values) may *not* be assigned to REF ANY T variables (since this would allow assignments that attempted to change the tag); assignment in the reverse direction is also disallowed. In the terminology of the Mesa manual, T (and bound T) conform to ANY T but do not freely conform.

In a discriminating selection, the semantics of the OpenItem will be changed in Cedar Mesa so that the expression is evaluated only once (not on each use, as in current Mesa). Thus there will be no possibility that subsequent references to the discriminated object will access a different record.

The scheme described above is semantically incompatible with current Mesa. The 6T5 version of Cedar Mesa does not implement these changes but retains the Mesa 6.0 treatment of variant records. The runtime system does, however, disallow any assignment to an RC variant record that would change the tag of that record; i.e., all RC variant records in collectible storage are forced to be immutable by runtime checks.

Strings, Text, and Sequence Types

Cedar Mesa supports the sequence types introduced in Mesa 6. Sequences are the usual way of dealing safely with objects in collectible storage that have dynamically computed sizes. COMPUTED sequence types may not be used in checked regions if the sequence elements are RC.

Mesa 6T5 retains the restrictions on sequence usage found in Mesa 6.0; it does, however, guarantee that all elements of RC sequences are set to NIL upon allocation. We plan to allow sequences that are not embedded within records and to provide sequence constructors.

Sequences of characters are a frequently-used programming construct. In present Mesa, STRING denotes a pointer to such a sequence (generally allocated on the heap). The fact that "strings" are actually pointers is a frequent source of errors, as is the explicit management of the storage containing the characters. We expect the availability of garbage collection to substantially change the

normal programming style associated with these sequences.

We have introduced a predeclared type TEXT for a (packed) sequence of characters. REF TEXT will generally replace most uses of STRING, which, however, remains available in unchecked regions as a synonym for POINTER TO StringBody. To reduce signed/unsigned number problems, the maximum length of the sequence in a TEXT value is 2^15.

A "string literals" denotes either a STRING or a REF TEXT constant with the READONLY attribute; context is used to make the distinction (as with CARDINALs and INTEGERs).

In Mesa 6T5, the type of a string literal in a context requiring a REF is REF TEXT; i.e., the literal is not protected against updates. There is currently some confusion about the proper interpretation of READONLY and (perhaps) CONST attributes.

Processes

Process handles offer the same problems as other potentially dangling pointers, but we wish to keep them in the safe subset. The implementation will be changed to include additional checking for erroneous situations (e.g., JOINing a process more than once). ABORT will be treated as a signal in the receiving process (with the additional property that the process is awakened to handle it, if it is waiting on a condition variable); it will cause termination of the process (after unwinding) if there is no handler for it in the aborted process.

The implementation has not yet been changed.

Unsafe Features

Although some of them have safe uses, the following Mesa language features may be used only in unchecked regions:

LOOPHOLE

UNCOUNTED, POINTER

Arithmetic to produce references

MACHINE CODE

TRANSFER WITH

RETURN WITH

STATE

Many procedures provided Pilot or by the current (unchecked) Mesa runtime system have unsafe interfaces, and hence cannot be called from checked regions. Examples are UnNew, UnNewConfig, and SelfDestruct.

DELAYED BINDING

Type Values

Cedar defines a representation of Mesa types in terms of ordinary Mesa values. For the most part, such representations are manipulated only by standard library packages, which can, e.g., discover the internal structure of an existing type or construct a representation of a new type. We extend the language to interpret a constant type expression as a denotation of such a representation, but we treat those representations opaquely, with assignment (including argument passing) and comparison for equality being the only available operations within the language itself.

In Mesa 6T5, the syntactic form CODE[T], where T is any statically evaluable type expression, denotes a value that is the internal representation of type T; that value has a type compatible with CedarTypes.Type.

We propose a language construct TYPE[x] which gives the type of a variable x as an object of type TYPE. This computation can always be done at compile time, except that the final mapping to the run time representation must be done at bind or load time. TYPE[x], however, must still obey "scope" restrictions: if x's type cannot be named in a scope (e.g., because it is private), then TYPE cannot be applied to x.

Mesa 6T5 does not provide the operator TYPE.

Dynamically Typed Entities

We have added dynamically typed entities to Mesa in the form of universal references. Recall that a universal reference is one with type REF ANY or REF READONLY ANY.

Use of Dynamically Typed Entities

As discussed above, counted references of any specific type may be widened to universal references, while narrowing a universal reference to a specific reference type requires a runtime check and may raise the error RangeError. In either case, the type of the target must have the READONLY attribute if the source does. The value NIL has the same representation for all reference types and can be narrowed without error to any specific type.

In Mesa 6T5, the narrowing must be explicit.

Cedar Mesa also provides a mechanism for using the dynamic type of a universal reference to select from an enumerated list of alternatives. We extend the WITH ... SELECT construct to allow the OpenItem to be a universal reference. The discrimination has the additional effect of copying the dynamically typed value to a statically typed reference variable.

In this form of selection, the TestList of each arm must be a declaration of a REF variable, the scope of which is just that arm. The universal reference is first evaluated; then the variables declared in the TestLists are considered in order. When a variable with a static type equivalent to the dynamic type of the universal reference is discovered, that reference value is assigned to the variable and then the corresponding arm is executed or evaluated. For example,

```
pTC: REF ANY;

WITH pTC SELECT FROM
```

```
ri: REF INTEGER => {
        --implicit ri _ NARROW[pTC]-- ...};
rr: REF REAL => {
        --implicit rr _ NARROW[pTC]-- ...};
ENDCASE => ERROR;
```

The test being applied to the dynamic type of the universal reference is type equivalence (i.e., identical representation at run time). If the value of a universal reference it NIL, its dynamic type is not equivalent to any specific reference type and the ENDCASE alternative is selected.

Alternatively, the programmer who wishes to treat failure of conformity as an exceptional condition may use NARROW[pTC] or NARROW[pTC, type]. This raises a signal RangeError if pTC is not of the appropriate type. Thus if the dynamic type of pTC were expected to be REF INTEGER, the above program fragment could be written as follows:

```
ri: REF INTEGER;
pTC: REF ANY;

ri _ NARROW[pTC ! RangeError => ERROR];
                                -- target type for NARROW is REF INTEGER
```

Note that the above constructs are the only ways to convert a TC entity back to a statically typed entity, which can be operated on by normal Mesa constructs.

As a syntactic convenience, Mesa 6T5 also provides a predicate for testing the dynamic type of a universal reference. The value of the expression ISTYPE[x, T] is true iff x is not NIL and the dynamic type of x is equivalent to the type T.

Procedure Application

Current Mesa requires that any invocation of a procedure supply an explicit list of arguments and result locations: only the identity of the procedure itself is left variable. In Cedar Mesa, the arguments and results may be manipulated as records (corresponding to the original Mesa observation that they are just ordinary data objects).

APPLY

APPLY [Proc, ArgRec] will apply a procedure to a pre-constructed argument record. ArgRec must conform to the declared argument record type for Proc. The conformance rules for procedure arguments and results have been generalized to allow records in place of explicit lists, with the record conforming if all its fields do. I.e., if a value of a record's type (written as a constructor) would be an acceptable argument list, then the record is an acceptable argument to APPLY; if it would be an acceptable extractor, the value of APPLY may be assigned to it.

APPLY is not implemented in Mesa 6T5. Note that similar operators are needed to invoke other transfer mechanisms, such as FORK, SIGNAL and RETURN, with preconstructed argument records.

Lists

LIST OF is a new type constructor. For any type T, LIST OF T and LIST OF READONLY T are also types. All LIST types are RC; the declaration

```
L: TYPE = LIST OF T;
```

is approximately equivalent to the (recursive) type definition

```
L: TYPE = REF %N;
%N: TYPE = RECORD [first: T, rest: L];
```

except that the record type %N is anonymous and such record types are *not* "painted." If T and U are equivalent types, LIST OF T and LIST OF U are also; in particular, writing LIST OF T in different places does not give rise to distinguishable types.

> If T conforms freely to U, LIST OF T conforms freely to LIST OF READONLY U. (This is an extension of the standard Mesa rule for conformance of pointer types.) In particular, LIST OF REF T conforms freely to LIST OF READONLY REF ANY (not to LIST OF REF ANY) for any type T.

The built-in list operators are .first, .rest, CONS and LIST. The first two are analogous to record field selectors; if tList: LIST OF T, tList.first is a variable with type T and tList.rest is a variable with type LIST OF T. If trList: LIST OF READONLY T, neither of these variables can be updated. Note that trList.rest is also of type LIST OF READONLY T and the entire list structure is protected from updates through tList.

For any list type L, NIL[L] is a constant of type L, designating no list at all, and can be written as NIL when the context implies the type.

CONS and LIST construct new list values and implicitly invoke NEW operations to create the required list cells. CONS is a binary operator and creates a single node, initialized with the values of its arguments. LIST is an *n*-ary operator an abbreviates a series of CONS operations. If

```
t, t1, ..., tn: T;
L: TYPE = LIST OF T;
tList: L;
```

the properties of CONS and LIST are defined by the following identities, where expressions separated by "==" are equivalent:

```
CONS[t, tList].first  ==  t
CONS[t, tList].rest  ==  tList
CONS[t, tList]  ##  NIL[L]

LIST[ ]  ==  NIL[L]
LIST[t1, t2, ..., tn]  ==  CONS[t1, LIST[t2, ..., tn]]
```

CONS and LIST may be written as operations of counted zones, e.g., zone.CONS[t, tList], in which case the NEW operation of the specified zone is used to allocate the list nodes. The standard counted prefix zone supplied by the system is used if the zone specification is omitted.

CONS and LIST are polymorphic operators; the selection of the list type is governed by context. If the context does not imply a specific list type, the selected operator is the CONS or LIST that creates lists of type LIST OF REF ANY.

Mesa 6T5 provides no notational convention for naming the CONS or LIST operation that produces a particular list type.

No further additions to the type system or syntax are proposed for lists. Polymorphism for lists will not be provided until the problem is solved for the language in general. (Note that the lack of polymorphic array procedures has not kept ARRAY from being a useful type constructor.)

Atoms

We have introduced a predeclared type ATOM. Each atom has a print name, which is a sequence of characters, and further attributes defined by the library package that implements operations upon atoms. An atom is represented by a counted reference to an opaque object and thus is RC. The library package enforces the following invariant: two atoms are equal as reference values if, and only if, their print names are identical.

Cedar Mesa provides a denotation of atom literals. If id is any Mesa identifier (including a reserved word), $id denotes the atom literal with print name id.

EXPORTED TYPES AND OBJECTS

Exported Opaque Types

The opaque types in Mesa 6T5 are identical to the exported types of Mesa 6. In particular, guaranteeing type safety requires the restriction that, within any system, there must be a unique concrete type for each opaque type. We would like to allow different implementations to supply different concrete types and to modify the type checking rules to prevent unintentional mixing of different concrete types. Work on this area has been deferred; there is considerable interaction with issues raised by system modelling.

Object Notation

We adopt a convention that allows "object-oriented" notation to be used in conjunction with types imported from interfaces. Let T be a type declared in an interface definition D. If I is some instance of D and if the type of x is I.T, then x.Op[args] is interpreted as I.Op[x, args] if Op has no other interpretation as a field of x. (If the argument list uses keyword notation, the keyword supplied for x is the identifier of the positionally first formal of Op).

Note that x.Op abbreviates x.Op[] according to this convention, but in some instances empty brackets may be required to resolve ambiguity, e.g., x.ReturnsAProc[][args].

Mesa 6T5 supports the rewriting only if the type of x, perhaps after dereferencing, is opaque. In addition, the procedure Op is always obtained from the principal imported instance of D as defined in Mesa 6.0. Removal of either of these restrictions requires resolution of some issues that also involve system modelling.

Generic Types

This heading is a place holder. We are currently reconsidering the 6C2 proposal for generic types.

Subclasses

This heading is also a place holder. We are currently reconsidering the 6C2 proposal for subclasses.