Inter-Office Memorandum

To	Cedar Users	Date	May 20, 1981
From	Ed Satterthwaite	Location	Palo Alto
Subject	Cedar 7T7 Language and Compiler Changes	Organization	CSL

XEROX

Filed on: [Ivy]<CedarDocs>Lang>Cedar7T7.Bravo

DRAFT

There is a new version of the Cedar compiler. This memo summarizes the changes to the language and compiler that you should know about.

Types in Cedar

This section sketches some current thinking about the Cedar type system and might help you to understand the motivation for some of the changes described below. (See also Lampson, *Cedar abstract machine* [CedarAM.memo, February 1980].)

Types as Predicates

Every type is characterized by some predicate; a value *x* has type *T* iff *x* satisfies the predicate for *T*. In general, such predicates are defined in terms of a set of marks (tags, etc.) carried by each value; however, the Mesa type system is designed so that most mark manipulation can be done statically (by the compiler), and the usual representations of most values do not include explicit marks.

A given expression has some fixed *syntactic type* that depends upon the form of the expression and the declared types of constituent identifiers. The value denoted by an expression always satisfies the predicate characterizing its syntactic type, but such a value will often satisfy predicates characterizing other types as well. In this sense, a Cedar value may have an arbitrary number of types. For example:

> If *Thing* is a variant record type with a variant *red*, a reference to a *Thing* might simultaneously satisfy the predicates for REF ANY, REF *Thing*, and REF *Thing*[*red*] (formerly REF *red Thing*, see below).
>
> An opaque type and the corresponding concrete type are distinct, even within an exporter of the concrete type, but the predicates for the two types are identical.

Roughly speaking, the primary job of the predicates associated with types is to provide correct answers to questions about low-level representational conventions so that, e.g., the Cedar garbage collector can operate correctly.

The form ISTYPE[*x*, *T*] returns the result of applying the predicate characterizing *T* to the value *x*. In Cedar 7T7, ISTYPE has been redefined to work in a somewhat more general and uniform way, and the operations of NARROWing and (type-based) SELECTion have been defined in terms of ISTYPE.

Types as Clusters of Operations

In addition, a *cluster* of operations (sometimes called a *group*) can be associated with a type. The main purposes of this grouping are to provide a number of packaging conveniences and to support so-called "object oriented" notation. If *x* has (syntactic) type *T*, *x.Op*[*args*] means *Op*[*x, args*] where *Op* is found by looking in the cluster associated with *T*. Two types may be characterized by the same predicate but have different associated clusters; in current Cedar, this is true of, e.g., an opaque type and the corresponding concrete type.

Each of the type constructors in Cedar supplies a standard and implicitly defined cluster for each type that it constructs. The only mechanism currently available for the explicit construction of such a cluster is the interface module, and previous versions of Cedar have limited support of this mechanism to opaque types. If *T* is an opaque type declared by, e.g.,

T: TYPE;

in some interface *Defs*, operations (procedures) declared in *Defs* become components of the cluster associated with *T* and may be invoked using object notation. Cedar 7T7 extends this support to allow construction of similar clusters for record types. If *T* is declared in *Defs* by

T: TYPE = RECORD [...];

the operations declared in *Defs* become part of the cluster associated with *T*. In this case, however, they augment the operations already supplied for *T* by the RECORD type constructor.

Defining clusters in this way has some drawbacks. The use of interfaces as the units of grouping somewhat overloads the existing notion of an interface; note that all operations declared in an interface become parts of the clusters of all types declared in that interface. Also, requiring a type and the operations in its cluster to be defined in the same interface occasionally conflicts with other criterea for partitioning interfaces. On the other hand, this method of defining clusters seems to cover the important cases well enough to be acceptable in practice. In addition, there is a fairly well worked-out plan for supporting clusters in a comprehensive, uniform way and for using them to explain parts of the Cedar abstract machine. We therefore recommend the following style guidelines for your Cedar programming:

> Partition interfaces so that a single interface defines both a main type *T* (record or opaque) and all the operations to be provided in the cluster of *T* (or REF *T*). Define multiple main types within an interface only if the sets of meaningful operation names for those types are disjoint.
>
> Use object notation in clients of interfaces designed to support it; i.e., use *x.Op*[*args*] in preference to *Defs.Op*[*x, args*].
>
> (For Humus veterans) Avoid interface designs that require clients to write *x.Op*[*x, args*], *x.ops.Op*[*x, args*] or the like. Use an inline definition of *Op* within *Defs* to achieve such an effect.

LANGUAGE CHANGES

Syntax for Discriminated Types

If *V* is a type expression designating some variant record type with variant *a*, *V*[*a*] is a type expression designating the discriminated type. Thus forms such as

Object[*red*] *Object*[*red*][*short*] *Object*[*red*][*long*][*80*]

are equivalent to the old forms

red Object *short red Object* *long red Object*[*80*].

In Cedar 7T7, both forms are acceptable, but you will eventually have to convert to the former as Cedar moves toward a unified syntax for expressions and type expressions.

Type Discrimination

Cedar 7T7 unifies the mechanisms for discriminating variant records with those for discriminating values with type REF ANY. This unification affects the operators ISTYPE and NARROW as well as discriminating selection.

Type Testing

The primitive function ISTYPE tests whether a given value satisfies the predicate characterizing a specified type. You will probably have little direct use for ISTYPE; its importance lies in its use to define other, more common operations as described below. Let *x* be an expression with syntactic type *S*. In Cedar 7T7, the value of ISTYPE[*x*, *T*] is determined as follows, where *V* is any variant record type:

(1) It is TRUE (at compile time) if

S and *T* are equivalent types; or
S is an opaque type and *T* is the corresponding concrete type; or
S is a concrete type exported as the opaque type *T*.

The last two cases are recognized only within program modules that export the concrete type.

(2) It is determined dynamically by a test of the value *x*, yielding TRUE or FALSE, if

S is REF ANY and *T* is REF *U* for any *U* except ANY; or
S is equivalent to *V* and *T* is equivalent to *V*[*a*]; or
S is equivalent to REF *V* and *T* is equivalent to REF *V*[*a*]; or
S is equivalent to (LONG) POINTER TO *V* and *T* is equivalent to (LONG) POINTER TO *V*[*a*];

where *V*[*a*] is a particular variant of *V*, perhaps discriminated to several levels. Note that the result is TRUE if the value of *x* is NIL.

(3) In all other cases, ISTYPE is unimplemented and is treated as a compile-time error.

Subsequent versions of Cedar will provide a more general definition and implementation of ISTYPE. Note in particular that ISTYPE cannot currently be used to test a value for membership in a subrange.

Narrowing

NARROW[*x*, *T*] allows a value *x* to be viewed as a value of type *T* and succeeds iff ISTYPE[*x*, *T*] is TRUE. More precisely, NARROW[*x*, *T*] has (syntactic) type *T*, and its value is given by

IF ISTYPE[*x*, *T*] THEN *x* ELSE ERROR <Error>

where <Error> is

RTTypesBasic.NarrowRefFault[*x*, CODE[*T*]]	if ISTYPE[*x*, REF ANY]
RTTypesBasic.NarrowFault[]	otherwise.

The following situations correspond to the three cases enumerated in the definition of ISTYPE above:

(1) NARROW[*x, T*] is guaranteed (at compile time) to succeed.
(2) NARROW[*x, T*] may succeed or fail at run time.
(3) NARROW[*x, T*] is unimplemented.

Case (2) arises only when the syntactic type of *x* is related to *T* in one of the ways described above for ISTYPE. In Cedar 7T7, case (3) is treated as a compile-time type error. Fine point: NARROW[*x, T*] is also considered a compile-time error if the only possible value of *x* yielding TRUE is NIL. Use *x* = NIL instead.

In case (1), NARROW is an identity operation but can be useful to change the (syntactic) type of *x* without using a LOOPHOLE or requiring any code to be executed. Example:

```
Defs: DEFINITIONS = {
  T: TYPE;
  R: TYPE = RECORD [g: REF T, ... ];
  Pn: PROC [r: REF R];
  ... }.

Impl: PROGRAM EXPORTS Defs = {
  T: PUBLIC TYPE = RECORD [n: NAT, ...];
  Pn: PUBLIC PROC [r: REF Defs.R] = {
    r.g.n _ 0;                    -- invalid; r.g^ is opaque, with no field selection operations
    NARROW[r.g, REF T].n _ 0;     -- valid (because Impl exports Defs)
    ...};
  }.
```

As before, NARROW[*x, T*] may be written as NARROW[*x*] when the target type *T* is implied by context.

Discriminating Selection

The syntactic form of WITH ... SELECT that is currently used for REF ANY discrimination has been extended to discriminate any value for which ISTYPE performs a dynamic test of that value (see case (2) in the discussion of ISTYPE). The form

```
WITH v SELECT FROM
  v1: T1 => s1;
  v2: T2 => s2;
  ...
  vn: Tn => sn;
  ENDCASE => se;
```

is, by definition, equivalent to

```
u: T = v;
IF u # NIL AND ISTYPE[u, T1] THEN {v1: T1 _ NARROW[u]; s1}
ELSE IF u # NIL AND ISTYPE[u, T2] THEN {v2: T2 _ NARROW[u]; s2}
  ...
ELSE IF u # NIL AND ISTYPE[u, Tn] THEN {vn: Tn _ NARROW[u]; sn}
ELSE se;
```

where *T* is the (syntactic) type of *v*. The tests against NIL are omitted if *T* does not have a NIL value.

Note that this form always copies the discriminated value. Thus

```
r: REF V;
. . .
WITH r SELECT FROM
  x: REF V[a] => { ... x ...};      -- x is a copy of r with type REF V[a]
  ...
  ENDCASE;
WITH r^ SELECT FROM
  x: V[a] => { ... x ...};      -- x is a copy of r^ with type V[a]
  ...
  ENDCASE;
```

Contrast these with the old form of variant record discrimination, which does not copy the discriminated value and reevaluates the discriminating expression each time that it is used:

```
WITH x: r SELECT FROM
  a => { ... x ... };      -- x is a synonym for r^ (but with syntactic type V[a])
  ...
  ENDCASE;
```

The new forms are easier to make type-safe, and you should use them whenever possible. Unfortunately, the old form is still required, at least outside the checked language, for dealing with computed variants and with pointers having non-standard dereferencing operations, such as the current relative pointers).

Interaction with Opaque Types

If *T* is any exported type, REF *T* must have the "standard" implementation of type discrimination. We impose this requirement in anticipation of making REF ANY discrimination work correctly with opaque types (it still doesn't in 7T7). As a consequence, discriminated variant record types cannot be exported as the concrete values of opaque types.

Object Notation

In Cedar, the form *x.Op*[*args*] is interpreted as *Defs.Op*[*x, args*] if the type of *x* is (REF | POINTER TO)* *T* for some opaque type *T* declared in an interface, the principal instance of which is *Defs*. In other words, all the operations defined in *Defs* become part of the cluster of the type *T*.

In Cedar 7T7, this convention applies within the corresponding DEFINITIONS module (for writing inlines, etc.) as well as within importers of such modules. This is only a notational extension; the bindings of implicitly imported values are determined as before.

The clustering mechanism has also been extended in Cedar 7T7 so that all operations declared in an interface become components of the clusters of any record types defined in that interface. With this extension, *Op* can be inline in more interesting ways. In addition, you may now be able to use object notation more extensively to invoke operations in existing interfaces, many of which are written in terms of (concrete) record types.

Note that every operation declared in an interface module becomes part of the cluster of every (record or opaque) type declared in that interface. Although the type of a particular operation normally will make it a useful component of only one cluster, its name appears in every other cluster and potentially hides or precludes a more appropriate definition of that name for that cluster. You therefore should define more than one main type per interface only if the sets of meaningful operation names for those types are disjoint.

Other points to note when using this convention with record types include the following:

When looking up *Op*, the field identifiers declared in *T* take precedence over the identifiers declared in the interface *Defs*.

A value *x* with a record type *T* having a single component can be coerced to a value with the type of that component. In the form *x.id*, the lookup of *id* considers first the field identifier of the single component, then identifiers declared in the interface defining *T*, and finally any interpretation given to *id* by applying the coercion. You abuse this feature at your own risk (but see the discussion of clusters above). Example:

```
Defs1: DEFINITIONS = {
  ...
  T1: TYPE = RECORD [f1: REF Defs2.T2];
  ...
  OpN: PROC [self: T1, ...];
  ...}.

Defs2: DEFINITIONS = {
  ...
  T2: TYPE = RECORD [ ... ];
  ...
  OpM: PROC [self: REF T2, ...];
  OpN: PROC [self: REF T2, ...];
  ...}

r1: Defs1.T1;
r2: REF Defs2.T2;

... r1.OpN[...]  means Defs1.OpN[r1, ...]        -- from the cluster defined by Defs1
... r1.OpM[...]  means Defs2.OpM[r1.f1, ...]     -- from the cluster defined by Defs2 (after coercion)
... r2.OpN[...]  means Defs2.OpN[r2, ...]
... r1.f1.OpN[...] means Defs2.OpN[r1.f1, ...]   -- dubious style
```

Rope Literals

The Cedar 7T7 compiler recognizes and correctly translates rope literals. Such a literal is denoted by a quoted string, e.g., "This is a rope literal". Its value is a reference to a rope object in the standard (counted) zone provided by the Cedar system.

The target type established by the context in which a quoted string literal appears determines the interpretation of that literal. There are three cases:

If the target type is *Rope.Ref*, the quoted string denotes a rope literal and has type *Rope.Ref*.

If the target type is any other REF type, the literal has type REF TEXT.

Otherwise, the literal has type STRING.

In the first case, the test is actually for equivalence between the target type and REF *Rope.RopeRep*. The matching is performed on the names of the interface (*Rope*) and referent type (*RopeRep*), not on the structure of the referent type. Since this is a loophole in the type checking, use nonstandard versions of the *Rope* interface very cautiously.

COMPILER CHANGES

Version Stamps

In its intermodule type checking, Cedar uses so-called version stamps to identify independently compiled modules. The version stamps computed by the Cedar 7T7 compiler are functions of the identity of that compiler and of its inputs. You can now recompile the same source file, with the same included modules, the same compiler and the same switch settings to get an object file with the same version stamp.

This stamp, which is essentially a 48 bit hash, is computed recursively as follows. Assume that any existing derived object (including the compiler itself) has a version stamp. The stamp for a new derived object is a hash of

the creation time of the source file
the version stamp of each bcd mentioned in the DIRECTORY clause
the version stamp of the compiler
the compiler switches (with those controlling only compile-time feedback masked off)

There is also a 7T7 binder that computes version stamps for its output in the same way.

Note

In the past, the version stamp has been a concatenation of a machine identifier and the creation time of the derived object. Many existing utility programs therefore print the version stamp formatted as a machine and network number, a date and a time. These programs give strange-looking output but, as far as we know, will perform correctly. (Be sure to update DescribeBcd, from Eric Schmidt, before using it with 7T7 object files).

Compiler Switches

The Cedar compiler is no longer able to generate object code for an Alto (or D-machine emulating an Alto). The switches `/a` and `/l` are ignored.

There is a Cedar switch `/c`; if it is set (the default), the code for FORK and JOIN assumes the availability of the Cedar runtime. If you plan to run your program directly under Pilot, compile with `/-c`. If you are in doubt about how your processes will interact with the Cedar runtime, consult a wizard.

Type Table Entries

The 7T7 compiler guarantees that, for every type *T* that is an argument of NEW (for a counted zone), an entry for REF *T* appears in the run-time type table. (This is a temporary concession, made until the run-time type system has a mechanism for building new types on demand.)

Distribution:
CedarLWG
Cedar Users