Inter-Office Memorandum

To	Cedar Interest	Date	October 12, 1982
From	Paul Rovner	Location	Palo Alto
Subject	The Cedar Runtime Type System	Organization	PARC/CSL

XEROX

Filed on: [Indigo]<Cedar>Documentation>RTTypes.tioga

INTRODUCTION

The Cedar runtime type system provides procedures for examining arbitrary Cedar values and TYPEs at runtime and for creating new values and modifying variables. Using its interfaces, one can write a program that deals with any given Cedar value or TYPE without anticipating the specific TYPE when the program is written.

The natural client for such features in a programming system that employs type checking at compile time is the debugger. The primary motivation for the current implementation was to provide a foundation for the Cedar debugger, based on the concepts of the Cedar Abstract Machine for interpreting values and types.

The key concepts of the Cedar Abstract Machine for the purpose of this memo are that a type carries with it a set of operations and that a variable carries with it its type. For a more complete treatment of these and related issues, see, on [Ivy], <Lampson>CedarAM.press, <Morris>SAM.press and <Rovner>RTT>AMFundamentals.press.

The current implementation is too slow to be used effectively by client programs as a substitute for true polymorphism in the language, but is suitable for examining and changing variables interactively with the Cedar debugger. It is our intention to improve the implementation (but retain the spirit of its definition) when the language is prepared to deal adequately with polymorphism. We expect that it will then be more natural for run-of-the-mill Cedar programs to make use of these or similar facilities.

The implementation of the runtime type system is included in the Cedar boot files. The interfaces referenced in this memo are RTBasic, RTTBridge, RTTRemoteBridge, RTTypes and RTSymbols. These are exported by RTT.bcd, which is a component of CedarCore.bcd. All files that constitute the runtime type system are accessible via [Indigo]<Cedar>Top>Runtime.df.

1.0 Concepts and Perspective

Cedar is a strongly-typed language. Every variable (including ones that are immutable after compile-time) has an associated bundle of information that can be used to determine the meaning of the bits of its value. In the Cedar Abstract Machine formulation, this bundle is called a **Type**. Each **Type** corresponds to a Cedar TYPE, and is represented at runtime as an instance of **RTBasic.Type** (for more detail about the Cedar runtime system see [Indigo]<Cedar>Documentation>Runtime.tioga).

For the Cedar type system, it is convenient to categorize Types by what we call their **Class**. For example, this is useful for deciding how to display a value of a given Type.

Every Cedar variable that can affect the outcome of a computation can be viewed as a component of the Cedar Abstract Machine. We call a description of such a Cedar variable a **TypedVariable** . We use the term **Status** for the mutability property of TypedVariables. A TypedVariable describes:

the Type of its value

whether the value is a constant (i.e. immutable after compile-time)

ifso, the actual bits of the value (status **const**)

ifnot, the addresses of the memory cells that contain the bits of the value and whether the variable may be changed (status **readOnly** or **mutable**)

Note that frames (both local and global) are viewed as TypedVariables in the Cedar Abstract Machine. As you will see below, the implementation provides procedures for dealing with frames and control links.

A client of the runtime type system approaches it via the "Bridge" interfaces (RTTBridge and RTTRemoteBridge) from either a local or remote Cedar world with one of the following:

a REF

a LONG POINTER and a RTBasic.Type

an ATOM

a ROPE

a PrincOps.FrameHandle

a PrincOps.GlobalFrameHandle

a PROC ANY RETURNS ANY

an ERROR ANY RETURNS ANY

For REFs, LONG POINTERs, FrameHandles and GlobalFrameHandles, procedures exist for creating TypedVariables that describe the referents. For ATOMs, ROPEs, PROCs and ERRORs, procedures exist for creating TypedVariables that have the specified values. Procedures of the RTTypes interface allow one to examine and modify components of TypedVariables and examine the structure of Cedar/Mesa TYPEs.

Generally, the RTTypes procedures are uniformly applicable both to TV's for elements of the local world and TV's for elements of a remote world. The only exception to this in the current implementation is the restriction that assignment (via RTTypes.Assign) of a REF-containing value to a TV in a remote world is not allowed.

2.0 The Classes of Types

```
Type: TYPE = RTBasic.Type;

RTTypes.Class: TYPE =
 {
  definition,  -- e.g. TypeClass[CODE[T]] where T: TYPE = ...--
   cardinal, longCardinal, integer, longInteger, real, character, --basic
   atom, rope, list, ref, pointer, longPointer, descriptor, longDescriptor,
     basePointer, relativePointer --address--
   procedure, signal, error, program, port, --transfer--
   enumerated, subrange,
  union, sequence,
  record, structure,
  array,
```

```
    countedZone, uncountedZone,
    nil,    -- e.g. TypeClass[TVType[NIL]]
     unspecified,
    process,
    type,
    opaque,
    any,    -- e.g. TypeClass[Range[CODE[REF  ANY]]]
     globalFrame, localFrame    --frame--
  };
```

3.0 Mutability status of TypedVariables

```
RTTypes. Status: TYPE = {mutable, readOnly, const};
```

4.0 The "Bridge" interfaces: Conversion between Cedar values and TypedVariables

4.1 Types and constants defined in RTTRemoteBridge

```
World: TYPE = WorldVM.World;

RemoteRef: TYPE = RECORD
                    [world: World _ ,
                     worldIncarnation: LONG CARDINAL ,
                     ref: WorldVM.Address];
 nilRemoteRef: RemoteRef = [world: NIL, ref: 0, worldIncarnation: 0];

RemotePointer: TYPE = RECORD
                    [world: World _ ,
                     worldIncarnation: LONG CARDINAL ,
                     ptr: WorldVM.Address];
 nilRemotePointer: RemotePointer = [world: NIL, ptr: 0, worldIncarnation: 0];

RemoteFrameHandle: TYPE = RECORD
                    [world: World _ ,
                     worldIncarnation: LONG CARDINAL ,
                     fh: WorldVM.ShortAddress];
 nilRemoteFrameHandle:  RemoteFrameHandle
                  = [world: NIL, fh: 0, worldIncarnation: 0];

RemoteGlobalFrameHandle: TYPE = RECORD
                    [world: World _ ,
                     worldIncarnation: LONG CARDINAL ,
                     gfh: WorldVM.ShortAddress];
 nilRemoteGlobalFrameHandle:  RemoteGlobalFrameHandle
                  = [world: NIL, gfh: 0, worldIncarnation: 0];

RemotePD: TYPE = RECORD
                    [world: World _ ,
                     worldIncarnation: LONG CARDINAL ,
                     pd: UNSPECIFIED ];
 nilRemotePD: RemotePD = [world: NIL, pd: 0, worldIncarnation: 0];

RemoteSED: TYPE = RECORD
                    [world: World _ ,
                     worldIncarnation: LONG CARDINAL ,
                     sed: UNSPECIFIED ];
 nilRemoteSED: RemoteSED = [world: NIL, sed: 0, worldIncarnation: 0];
```

4.2 How to acquire a TypedVariable from a Cedar value

(a) From a local Cedar REF (RTTBridge)

```
TVForReferent: PROC[ref: REF ANY, status: Status _ mutable]
    RETURNS[TypedVariable];
```

This yields a TypedVariable that represents the Cedar value that is the referent of **ref**.

```
TVForReadOnlyReferent: PROC[ref: REF READONLY ANY]
    RETURNS[TypedVariable];
```

This yields a TypedVariable, status **readOnly**, that represents the Cedar value that is the referent of **ref**.

(b) From a remote Cedar REF (RTTRemoteBridge)

```
TVForRemoteReferent: PROC[remoteRef: RTTRemoteBridge.RemoteRef,
                          status: Status _ mutable]
    RETURNS[TypedVariable];
```

This yields a TypedVariable that represents the Cedar value that is the referent of **ref**.

(c) From other local Cedar values (RTTBridge)

```
TVForATOM: PROC[atom: ATOM]
    RETURNS[TypedVariable--atom--];
```

```
TVForROPE: PROC[rope: Rope.ROPE]
    RETURNS[TypedVariable--rope--];
      -- use TVToName to get the ROPE back
```

```
TVForProc: PROC[proc: PROC ANY RETURNS ANY]
    RETURNS[TypedVariable--procedure--];
```

```
TVForSignal: PROC[signal: ERROR ANY RETURNS ANY]
    RETURNS[TypedVariable--signal, error--];
```

These yield TypedVariables that describe the indicated procedure or signal descriptor. The Class of such a TypedVariable is either procedure, signal, or error.

(NOTE: The operations below and the TypedVariables they produce are unsafe)

```
TVForPointerReferent:
  PROC[ptr: LONG POINTER, type: Type, status: Status _ mutable]
    RETURNS[TypedVariable];
```

This yields a TypedVariable that represents the referent of the specified pointer. The specified Type is accepted as the variable's Type.

```
TVForFHReferent: PROC[fh: PrincOps.FrameHandle,
                      evalStack: WordSequence _ NIL]
    RETURNS[TypedVariable];
```

This yields a TypedVariable that represents the indicated local frame. The structure of nested blocks and their local variables will be determined from the PC that is stored in the frame. BEWARE. This is UNSAFE. Reference to this TypedVariable after the specified frame is no longer valid will lead to manifestations of strange and wondrous phenomena (RTTypes functions that deal with frame values validate them wherever possible, but accidents can still happen.)

```
TVForGFHReferent: PROC[gfh: PrincOps.GlobalFrameHandle]
```

RETURNS[TypedVariable];

This yields a TypedVariable that represents the indicated global frame.

(d) From other remote Cedar values (RTTRemoteBridge)

TVForRemotePointerReferent: PROC[remotePointer: RemotePointer,
type: Type,
status: Status _ mutable]
RETURNS[TypedVariable];

TVForRemoteFHReferent: PROC[remoteFrameHandle: RemoteFrameHandle,
evalStack: WordSequence _ NIL]
RETURNS[TypedVariable];

This yields a TypedVariable that represents the indicated local frame. The structure of nested blocks and their local variables will be determined from the PC that is stored in the frame. BEWARE. This is UNSAFE. Reference to this TypedVariable after the specified frame is no longer valid will lead to manifestations of strange and wondrous phenomena (RTTypes functions that deal with frame values validate them wherever possible, but accidents can still happen.)

TVForRemoteGFHReferent:
PROC[remoteGlobalFrameHandle: RemoteGlobalFrameHandle]
RETURNS[TypedVariable];

TVForRemotePD: PROC[remotePD: RemotePD]
RETURNS[TypedVariable--procedure--];

TVForRemoteSED: PROC[remoteSED: RemoteSED]
RETURNS[TypedVariable--signal, error--];

4.3 How to acquire a local Cedar value from a TypedVariable (RTTBridge)

RefFromTV: PROC[tv: TypedVariable] RETURNS[REF ANY];

If possible, this returns the REF that points to the value represented by **tv**. This raises InternalTV if **tv** is embedded, NotMutable if TVStatus[tv] # mutable.

SomeRefFromTV: PROC[tv: TypedVariable] RETURNS[REF ANY];

Like RefFromTV, but creates a copy of the specified value in collectible storage and returns a REF to the copy instead of raising an error.

ReadOnlyRefFromTV: PROC[tv: TypedVariable]
RETURNS[REF READONLY ANY];

If possible, this returns a READONLY REF that points to the value represented by **tv**. This raises InternalTV if **tv** is embedded.

PointerFromTV: PROC[tv: TypedVariable] RETURNS[LONG POINTER];

TVToATOM: PROC[tv: TypedVariable] RETURNS[ATOM];

TVToProc: PROC[tv: TypedVariable]
RETURNS[PROC ANY RETURNS ANY];

TVToSignal: PROC[tv: TypedVariable]
RETURNS[ERROR ANY RETURNS ANY];

These return the value represented by the specified TypedVariable. If tv is NIL, they

return null signal or procedure values.

```
FHFromTV: PROC[tv: TypedVariable] RETURNS[ PrincOps. FrameHandle];

GFHFromTV: PROC[tv: TypedVariable]
    RETURNS[ PrincOps. GlobalFrameHandle];
```

These return an appropriate kind of pointer to the value represented by the specified TypedVariable.

```
TVToLC: PROC[tv: TypedVariable] RETURNS[LONG CARDINAL];
TVToInteger: PROC[tv: TypedVariable] RETURNS[INTEGER];
TVToLI: PROC[tv: TypedVariable] RETURNS[LONG INTEGER];
TVToCardinal: PROC[tv: TypedVariable] RETURNS[CARDINAL];
TVToCharacter: PROC[tv: TypedVariable] RETURNS[CHARACTER];
TVToReal: PROC[tv: TypedVariable] RETURNS[REAL];
```

These procedures PUN the specified value into the specified Cedar basic TYPE. They raise RangeFault if the size of the specified TypedVariable is inappropriate.

4.4 How to acquire a remote Cedar value from a TypedVariable (RTTRemoteBridge)

```
RemoteRefFromTV: PROC[tv: TypedVariable] RETURNS[RemoteRef];

RemotePointerFromTV: PROC[tv: TypedVariable] RETURNS[RemotePointer];

RemoteFHFromTV: PROC[tv: TypedVariable]
RETURNS[RemoteFrameHandle];

RemoteGFHFromTV: PROC[tv: TypedVariable]
    RETURNS[RemoteGlobalFrameHandle];

TVToRemotePD: PROC[tv: TypedVariable--procedure, program--]
    RETURNS[RemotePD];

TVToRemoteSED: PROC[tv: TypedVariable] RETURNS[RemoteSED];
```

4.5 Remoteness properties of TypedVariables (RTTRemoteBridge)

```
IsRemote: PROC[tv: TypedVariable] RETURNS[BOOLEAN];

GetWorld: PROC[tv: TypedVariable] RETURNS[World];

GetWorldIncarnation: PROC[tv: TypedVariable] RETURNS[LONG
CARDINAL];
```

4.6 RTTBridge facilities of interest only to Wizards

The following stuff is for use by Wizards. If you need it, chances are good you're doing something wrong

```
WordSequence: TYPE = REF WordSequenceRecord;
WordSequenceRecord:
        TYPE = RECORD[s: SEQUENCE size: NAT OF WORD];

TVToWordSequence: PROC[tv: TypedVariable]
    RETURNS[s: WordSequence];
```

This is useful for TypedVariables with size>Size[CODE[LONG INTEGER]]. It will produce what amounts to an uninterpreted bit string for any TypedVariable.

```
Loophole: PROC[tv: TypedVariable, type: Type, tag: TypedVariable _ NIL]
    RETURNS[TypedVariable];

OctalRead: PROC[tv: TypedVariable, offset: INT]
```

RETURNS[CARDINAL];

BytePC: PROC[tv: TypedVariable--localFrame--]
RETURNS[PrincOps.BytePC];

IsStarted: PROC[tv: TypedVariable--globalFrame--] RETURNS[BOOL];

IsCopied: PROC[tv: TypedVariable--globalFrame--] RETURNS[BOOL];

TVHead: PROC[tv: TypedVariable] RETURNS[TVHeadType];

TVHeadType: TYPE =
{constant, remoteConstant, reference, remoteReference,
copiedRemoteObject, pointer, remotePointer, gfh,
remoteGFH, fh, remoteFH, notTVRecord};

SetTVFromLC: PROC[tv: TypedVariable, lc: LONG CARDINAL];

This function can change the value of a mutable non-ref-containing TypedVariable. A RangeFault can occur if the entire long cardinal value will not fit in the field described by tv.

SetTVFromLI: PROC[tv: TypedVariable, li: LONG INTEGER];

This is the same as SetTVFromLC, except that it uses signed operations to determine whether the value is in range.

5.0 Basic operations on TypedVariables from the RTTypes interface

TVType: PROC[tv: TypedVariable] RETURNS[Type];

TVStatus: PROC[tv: TypedVariable] RETURNS[Status];

TVSize: PROC[tv: TypedVariable] RETURNS[words: INT];

This returns the number of words of storage for the value described by **tv**.

Copy: PROC[tv: TypedVariable] RETURNS[TypedVariable];

Assign: PROC[lhs, rhs: TypedVariable];

If lhs is mutable and the types of the TypedVariables are equivalent, the value of rhs becomes the value of lhs.

TVEq: PROC[tv1, tv2: TypedVariable] RETURNS [BOOL];

This is true if both TypedVariables point to exactly the same bits; i.e., if assignment to one would change the value seen by the other.

TVEqual: PROC[tv1, tv2: TypedVariable] RETURNS [BOOL];

This is true if both values, taken as uninterpreted bit strings, are the same length and are equal.

6.0 Basic operations on Types from the RTTypes interface

Size: PROC[type: Type, length: CARDINAL --sequence only, number of elements-- **_ 0]**
RETURNS[words: CARDINAL --max for unions-- **];**

Size returns the number of words of storage for a value of the specified Type. See the record, union, and sequence sections for special interpretations.

New: PROC[type: Type,
status: Status _ mutable,
world: WorldVM.World _ WorldVM.LocalWorld[],
tag: TV _ NIL]

RETURNS[TypedVariable];

New creates and returns a cleared object.

DefaultInitialValue: PROC[type: Type] RETURNS[TypedVariable];

DefaultInitialValue is implemented only for types with constant initialization. At present it is not available for procedure parameter and result record types; use **IndexToDefaultInitialValue** to examine each field.

IndexToDefaultInitialValue: PROC[type: Type -- record or structure-- **, index: Index] RETURNS[TypedVariable];**

Returns the default initial value for the index'th field of type. Obviously not in all cases equivalent to DefaultInitialValue[IndexToType[type, index]] .

N.B. There are at present no operations for obtaining a Type describing a definitions module procedure definition; one instead normally obtains the Type of a specific exported implementation. The default initial values that one obtains from this Type will be those of the implementation, which may differ from those of the imported definition.

TypeClass: PROC[type: Type] RETURNS[Class];

IsPainted: PROC[type: Type] RETURNS [BOOL];

TRUE if type represents a painted type.

7.0 Applicable operations for each Class of Type

This section contains a list for each Class of Type of the operations that are applicable to TypedVariables and Types of that Class. You will note that some operations are applicable to more than one Class. A runtime test is made for each operation to verify that it is applicable to the given TypedVariable or Type. RTTypes.TypeFault is raised if this test fails.

Index: TYPE = NAT;

This is a numeric type that is used for indexing in some of the procedures below. Generally, the valid index range for these is [1..n].

definition

This class appears to confuse new users of the RTTypes interface. A Type has *definition* as its class IFF it was defined as an identifier that is equated to some other Type, as in

```
T: TYPE = RECORD[...
```

For example,

```
tv: TV _ TVForReferent[NEW[T]];
...
IF TypeClass[TVType[tv]] # definition THEN ERROR;
...
IF TypeClass[UnderType[TVType[tv]]] # record THEN ERROR;
```

TypeToName: PROC[type: Type,
moduleName: REF Rope.ROPE _ NIL,
fileName: REF Rope.ROPE _ NIL]
RETURNS[Rope.ROPE];

If moduleName and/or fileName are supplied, TypeName attempts to discover the module and file names for the module in which type was defined. If the supplied type has no name (e.g., an anonymous procedure parameter), NIL is returned.

Ground: PROC[type: Type] RETURNS[Type];

This peels off one layer of defintion.

UnderType: PUBLIC PROC[type: Type] RETURNS[Type];

This returns the first underlying non-definition type.

GroundStar: PUBLIC PROC[type: Type] RETURNS[Type];

This finds the first non-definition type, then peels off subrange or relativeRef layers to arrive at the "ground" Type.

cardinal, longCardinal, integer, longInteger, real, character

First, Last: PROC[type: Type] RETURNS[TypedVariable];

Next: PROC[tv: TypedVariable] RETURNS[TypedVariable];

Coerce: PROC[tv: TypedVariable, targetType: Type] RETURNS[TypedVariable];

See the *enumerated* section for a complete description. Coercions between *integer*s and *cardinal*s and their subranges are supported. No coercions currently exist between *real*s or *character*s and any other types, nor are coercions to/from long numeric types supported, although that would be easy.

list, ref

Range: PROC[type: Type] RETURNS[Type];

This returns the referent type of the specified ref type.

Referent: PROC[ref: TypedVariable] RETURNS[TypedVariable];

This is the dereferencing operation.

ReferentStatus: PROC[type: Type] RETURNS[Status];

This will return readOnly if the ref type is REF READONLY..., mutable otherwise.

Coerce: PROC[tv: TypedVariable, targetType: Type] RETURNS[TypedVariable];

Coerce will produce a TypedVariable, status readOnly, if appropriate type and range tests are successful. It will produce a TypedVariable for a REF ANY from a TypedVariable for some explicit REF type (i.e., Widen). It can also produce a TypedVariable for an explicit REF type from a TypedVariable for a REF ANY whose current referent is of an acceptable type (i.e., Narrow).

pointer, longPointer

Range: PROC[type: Type] RETURNS[Type];

This returns the referent type of the specified pointer type.

Referent: PROC[ref: TypedVariable] RETURNS[TypedVariable];

This is the dereferencing operation.

ReferentStatus: PROC[type: Type] RETURNS[Status];

This will return readOnly if the pointer type is ... POINTER TO READONLY..., mutable otherwise.

descriptor, longDescriptor

Range: PROC[type: Type] RETURNS[Type];

This returns the referent type of the specified descriptor type.

ReferentStatus: PROC[type: Type] RETURNS[Status];

This will return readOnly if the descriptor type is ... DESCRIPTOR FOR READONLY..., mutable otherwise.

Apply: PROC[tv: TypedVariable, arg: TypedVariable]
RETURNS[TypedVariable];

This returns a TypedVariable for the specified element of the specified array. NOTE: Apply for descriptors is not implemented yet.

procedure

Domain: PROC[type: Type] RETURNS[Type];
Range: PROC[type: Type] RETURNS[Type];
TVToName: PROC[tv: TypedVariable] RETURNS[ans: Rope.ROPE];
StaticParent: PROC[tv: TypedVariable]
RETURNS[TypedVariable --proc-- **];** -- maybe NIL
GlobalParent: PROC[tv: TypedVariable]
RETURNS[TypedVariable --globalFrame-- **];**

signal

Domain: PROC[type: Type] RETURNS[Type];
Range: PROC[type: Type] RETURNS[Type];
TVToName: PROC[tv: TypedVariable] RETURNS[ans: Rope.ROPE];
GlobalParent: PROC[tv: TypedVariable]
RETURNS[TypedVariable --globalFrame-- **];**

error

Domain: PROC[type: Type] RETURNS[Type];
TVToName: PROC[tv: TypedVariable] RETURNS[ans: Rope.ROPE];
GlobalParent: PROC[tv: TypedVariable]
RETURNS[TypedVariable --globalFrame-- **];**

program

Domain: PROC[type: Type] RETURNS[Type];
Range: PROC[type: Type] RETURNS[Type];
TVToName: PROC[tv: TypedVariable] RETURNS[ans: Rope.ROPE];

port No special operations (yet).

enumerated

NameToIndex: PROC[type: Type, name: Rope.ROPE] RETURNS[Index];

The range of **index** is [1..NValues[type]] . NameToIndex raises BadName

IndexToName: PROC[type: Type, index: Index] RETURNS[Rope.ROPE];

Raises BadIndex.

TVToName: PROC[tv: TypedVariable] RETURNS[ans: Rope.ROPE];

First, Last: PROC[type: Type] RETURNS[TypedVariable];

Next: PROC[tv: TypedVariable] RETURNS[TypedVariable];

Next returns NIL if there is no successor to **tv**.

NValues: PROC[type: Type] RETURNS[Index];

Value: PROC[type: Type, index: Index] RETURNS[TypedVariable];

This returns a TypedVariable for the **index**'th element of the enumerated type. The

range of **index** is [1..NValues[type]] .

Coerce: PROC[tv: TypedVariable, targetType: Type] RETURNS[TypedVariable];

Coerce determines whether targetType is equivalent to TVType[tv] or one if its subranges, and whether the current value of tv is in targetType's range. If so, it returns a readOnly TypedVariable for a copy of tv's value, coerced to the specified type. If not, it raises TypeFault or RangeFault.

IsMachineDependent: PROC[type: Type] RETURNS[BOOL];

subrange

Ground: PROC[type: Type] RETURNS[Type];

This peels off one layer of subrange.

GroundStar: PUBLIC PROC[type: Type] RETURNS[Type]; -- peels 'em all off

This finds the first non-subrange type.

InRange: PROC[type: Type, groundTV: TypedVariable] RETURNS[BOOL];

First, Last: PROC[type: Type] RETURNS[TypedVariable];

Raises RangeFault if the specified subrange is empty.

Next: PROC[tv: TypedVariable] RETURNS[TypedVariable];

Next returns NIL if there is no successor to **tv**.

Coerce: PROC[tv: TypedVariable, targetType: Type] RETURNS[TypedVariable];

See the *enumerated* section for a complete description. Coercions between *integer*s and *cardinal*s and their subranges are supported. No coercions currently exist between *real*s or *character*s and any other types, nor are coercions to/from long numeric types supported, although that would be easy.

union

IsMachineDependent: PROC[type: Type] RETURNS[BOOL];

IndexToType: PROC[type: Type, index: Index] RETURNS[Type];

An upper bound on the range of **index** is [1..NComponents[type]] (there may be fewer variants than elements of the tag type). IndexToType raises BadIndex. It returns the Type of the specified variant. The resulting record type appears to contain all of the common fields of the original union-record followed by all of the fields of the selected variant; it omits the tag field. Size[IndexToType[...]] returns the size of this particular variant.

NameToIndex: PROC[type: Type, name: Rope.ROPE] RETURNS[Index];

This returns the name of the selected variant arm. Raises BadName.

IndexToName: PROC[type: Type, index: Index] RETURNS[Rope.ROPE];

Raises BadIndex.

Tag: PROC[tv: TypedVariable] RETURNS[TypedVariable --enumerated-- **];**

Variant: PROC[tv: TypedVariable] RETURNS[TypedVariable --record-- **];**

This returns a TypedVariable for the currently selected variant, as indicated by the tag. The resulting record contains all of the common fields of the original union-record

(variant record) followed by all of the fields of this variant; it omits the tag field. TVSize[Variant[tv]] returns the actual size of this variant.

IsMachineDependent: PROC[type: Type] RETURNS[BOOL];

Domain: PROC[type: Type] RETURNS[Type --enumerated-- **];**

sequence

IsMachineDependent: PROC[type: Type] RETURNS[BOOL];

IsPacked: PROC[type: Type] RETURNS[BOOL];

Domain: PROC[type: Type] RETURNS[Type];

This returns the index Type that is associated with the specified sequence Type.

TagType: PROC[type: Type] RETURNS[Type];

Equivalent to Domain[type] for sequence types.

Range: PROC[type: Type] RETURNS[Type];

This returns the element Type that is associated with the specified sequence Type.

Apply: PROC[tv: TypedVariable, arg: TypedVariable]
RETURNS[TypedVariable];

This returns a TypedVariable for the specified element of the specified sequence. The element's value must be in the range First[TagType[TVType[tv]]]..Last[...]].

Tag: PROC[tv: TypedVariable] RETURNS[tagTV: TypedVariable];

This returns a TypedVariable for the index (tag) field of the sequence. The current implementation permints one to Assign to tagTV, but you shouldn't do it.

Length: PROC[tv: TypedVariable] RETURNS[TypedVariable];

This returns a readOnly TypedVariable for the length of (number of elements in) the specified sequence, represented as a long integer. The values returned by Tag and Length will be different if the TagType has a non-zero lower bound.

record, structure

Records differ from structures only in that record types are unique. No two record types are equivalent. Record types are said to be *painted*. Two structure types are equivalent if their element types are equivalent, in the order of declaration. In current Cedar, the only structure types are procedure (signal, error, program) argument and result types, implicit LIST component types, and RECORD types that are defined in program modules.

A variant record will be referred to here as a union-record: i.e., a record whose last component is of a union type. A record whose last component is of a sequence type will be referred to as a sequence-record. Some of the operations on these records apply to these record types and their TypedVariables; others apply directly to the sequence or union types and thier TypedVariables, as described below.

IsMachineDependent: PROC[type: Type] RETURNS[BOOL];

NComponents: PROC[type: Type] RETURNS[Index];

IndexToTV: PROC[tv: TypedVariable, index: Index]
RETURNS[TypedVariable];

The range of **index** is [1..NComponents[TVType[tv]]].
IndexToTV returns the indicated component of the record or structure as a TypedVariable.

IndexToType: PROC[type: Type, index: Index] RETURNS[Type];

Here **index** is used to select one of the types of the components of a record or structure.

IndexToName: PROC[type: Type, index: Index] RETURNS[Rope.ROPE];

Here **index** is used to select one of the names of the components of a record or structure.

NameToIndex: PROC[type: Type, name: Rope.ROPE] RETURNS[Index];

NameToIndex returns the index of the component in the record or structure type that has the specified name. NameToIndex raises BadName if the given type has no component with the specified name.

VariableType: PROC[type: Type] RETURNS [v: Type, c: Class];

If type represents:

A union-record: **c** is *union*, and **v** is the type of the union component.

A sequence-record: **c** is *sequence*, and **v** is the type of the sequence component.

Anything else: **c** is *nil*, and **v** is undefined.

VariableType makes it possible to determine whether records of this type contain any elements with variable structure.

If **c** is union or sequence, **n** is NComponents[type], and **tv** is a TypedVariable such that TVType[tv]=type, then IndexToType[type,n] yields **v**, and IndexToTV[tv] yields a TypedVariable describing the union or sequence, for use in the functions described in the sections below.

Size: PROC[type: Type, length: CARDINAL_ 0] RETURNS[CARDINAL];

If VariableType[type].class= *union*, Size returns the size of the largest variant.

If VariableType[type].class= *sequence*, the caller must supply a proposed length, and Size returns the size of a sequence-record whose sequence contains length elements.

array

IsPacked: PROC[type: Type] RETURNS[BOOL];

Domain: PROC[type: Type] RETURNS[Type];

This returns the index Type that is associated with the specified array Type.

Range: PROC[type: Type] RETURNS[Type];

This returns the element Type that is associated with the specified array Type.

Apply: PROC[tv: TypedVariable, arg: TypedVariable]
RETURNS[TypedVariable];

This returns a TypedVariable for the specified element of the specified array.

countedZone No special operations (yet).

uncountedZone No special operations (yet).

nil This is TypeClass[RTBasic.nullType]. No special operations (yet).

unspecified This is TypeClass[Range[CODE[POINTER TO UNSPECIFIED]]]. No special operations (yet).

process No special operations (yet).

type TypedVariables having Types in this Class may be found in global frame TypedVariables. No special operations (yet).

opaque No special operations (yet).

any This is TypeClass[Range[CODE[REF ANY]]]. No special operations (yet).

globalFrame

TVToName: PROC[tv: TypedVariable] RETURNS[ans: Rope.ROPE];
Globals: PROC[tv: TypedVariable] RETURNS[TypedVariable --record-- **];**

localFrame

Procedure: PROC[tv: TypedVariable]
RETURNS[TypedVariable --procedure-- **];**

This returns a TypedVariable for the procedure for **tv** this is a local frame. Raises TypeFault if the frame is a catch frame.

Signal: PROC[tv: TypedVariable] RETURNS[TypedVariable --signal-- **];**

This returns a TypedVariable for the signal for which **tv** is a catch frame. Raises TypeFault if the frame is not a catch frame.

Argument: PROC[tv: TypedVariable, index: Index]
RETURNS[TypedVariable];

This returns a TypedVariable for the specified argument of the procedure call represented by **tv**.

Result: PROC[tv: TypedVariable, index: Index]
RETURNS[TypedVariable];

This returns a TypedVariable for the specified result of the procedure call represented by **tv**.

EnclosingBody: PROC[tv: TypedVariable]
RETURNS[TypedVariable --localFrame-- **];**

TVForFHReferent examines the local frame's program counter (PC) to produce a TypedVariable describing the innermost block surrounding the PC. Each call on EnclosingBody produces a TypedVariable representing the same local frame, but at the next outer level of block nesting. Then use Locals to obtain a TypedVariable describing the local variables at each level. If **tv** already represents the outer block, EnclosingBody returns NIL.

If EnclosingBody is applied to a **tv** representing a catch frame, it may return a tv describing the locals surrounding the catch phrase rather than NIL; this is due to deficiencies in the symbol table, and it should eventually be corrected.

Locals: PROC[tv: TypedVariable] RETURNS[TypedVariable --record-- **];**

This returns a TypedVariable for a structure describing the locals in the local frame and at the block level represented by **tv**.

GlobalParent: PROC[tv: TypedVariable]
RETURNS[TypedVariable --globalFrame-- **];**

This returns a TypedVariable for the global frame corresponding to **tv**'s local frame.

DynamicParent: PROC[tv: TypedVariable]
RETURNS[TypedVariable --normally a localFrame-- **];**

This returns a TypedVariable for the frame of the caller of the frame represented by **tv**. This will normally be a localFrame TypedVariable, but strange things happen at the root of processes and when PORTs are used. Normally, the root localFrame of a process is a program TypedVariable. More can be said about all of this, but this is not the place.

8.0 ERRORs

```
Error: ERROR
         [reason: ErrorReason,
            msg: ROPE _ NIL,
            type: Type _ RTBasic.nullType,  -- used with TypeFault, IncompatibleTypes
            otherType: Type _ RTBasic.nullType     -- used with IncompatibleTypes
           ];
ErrorReason: TYPE =
                  {noSymbols,            -- msg has moduleName
                   notImplemented,       -- mostly DefaultInitialValue cases
                   incompatibleTypes,    -- raised by Assign
                   rangeFault,           -- e.g. empty subrange, Apply bounds check
                   notMutable,           -- raised by (e.g.) Assign
                   internalTV,           -- raised by (e.g.) RefFromTV
                   badName,              -- raised by NameToIndex
                   badIndex,             -- raised by (e.g.) IndexToType
                   typeFault             -- general applicability violation
                  };
```

9.0 Examples

The example below comes directly from one of my test programs, and has several strange properties and dependencies, none of which are relevant to its value here as a source of examples.

```
PrintTV: PROC [tv: RTTypes.TV, depth: CARDINAL _ 0] =
BEGIN OPEN RTTypes;
  type: Type = TVType[tv];

  SELECT TypeClass[type] FROM
    union =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteString["(a union) Tag = "];
        PrintTV[Tag[tv]];
        THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteString["Variant = "];
        PrintTV[Variant[tv], depth + 1];
      };
    definition =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteString["(a definition) = "];
        WriteString[TypeToName[type]];
        WriteLine[""];
        PrintTV[Coerce[tv, UnderType[type]], depth + 1];
      };
    localFrame =>
      { pt: TV = Procedure[tv];
        THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
```

```
        WriteString["(a local frame for "];
         IF TypeClass[TVType[pt]] = program THEN
            { WriteString["the program named "]; WriteString[TVToName[pt]]; WriteLine[")"];
RETURN };
        WriteString[TVToName[pt]];
        WriteLine["): "];
        IF EnclosingBody[tv] = NIL THEN
         { domainType: Type = Domain[TVType[pt]];
           rangeType: Type = Range[TVType[pt]];
           IF domainType # RTTypesBasic.nullType
                 THEN {THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP ;
                WriteLine["Arguments:"];
                FOR i: CARDINAL IN [1..NComponents[domainType]]
                       DO
                   THROUGH [0..depth] DO WriteChar[Ascii.SP] ENDLOOP ;
                   WriteString[IndexToName[domainType, i]];
                   WriteLine[": "];
                   PrintTV[Argument[tv, i], depth + 2];
                  ENDLOOP };
           IF rangeType # RTTypesBasic.nullType
                 THEN {THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP ;
                WriteLine["Results:"];
                FOR i: CARDINAL IN [1..NComponents[rangeType]]
                       DO
                   THROUGH [0..depth] DO WriteChar[Ascii.SP] ENDLOOP ;
                   WriteString[IndexToName[rangeType, i]];
                   WriteLine[": "];
                   PrintTV[Result[tv, i], depth + 2];
                  ENDLOOP };
           THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP ;
           WriteLine["Locals:"];
           PrintTV[Locals[tv], depth + 1]}
         ELSE { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP ;
                WriteLine["EnclosingBody:"];
                PrintTV[EnclosingBody[tv], depth + 1]};
       };
     globalFrame =>
       { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP ;
         WriteString["(a global frame for "];
         WriteString[TVToName[tv]];
         WriteLine["): "];
         PrintTV[Globals[tv], depth + 1];
       };
      nil=>
       { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP ;
         WriteLine["(no value) "];
        };
     countedZone =>
       { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP ;
         WriteLine["(a countedZone) "];
        };
     uncountedZone =>
       { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP ;
         WriteLine["(an uncountedZone) "];
```

```
    };
  procedure =>
    { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
      WriteString["(a procedure named "];
      WriteString[TVToName[tv]];
      WriteLine[")"];
      THROUGH [0..depth + 1] DO WriteChar[Ascii.SP] ENDLOOP;
      WriteString["Defined in the module named "];
      WriteLine[TVToName[GlobalParent[tv]]];

      THROUGH [0..depth + 1] DO WriteChar[Ascii.SP] ENDLOOP;
      WriteString["Its Domain..."];
      IF Domain[type] = RTTypesBasic.nullType THEN WriteLine["no value"]
       ELSE IF TypeClass[Domain[type]] = structure THEN WriteLine["a structure value"]
       ELSE ERROR;
      THROUGH [0..depth + 1] DO WriteChar[Ascii.SP] ENDLOOP;
      WriteString["Its Range..."];
      IF Range[type] = RTTypesBasic.nullType THEN WriteLine["no value"]
       ELSE IF TypeClass[Range[type]] = structure THEN WriteLine["a structure value"]
       ELSE ERROR;
    };
  signal =>
    { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
      WriteString["(a signal named "];
      WriteString[TVToName[tv]];
      WriteLine[")"];
      THROUGH [0..depth + 1] DO WriteChar[Ascii.SP] ENDLOOP;
      WriteString["Its Domain..."];
      IF Domain[type] = RTTypesBasic.nullType THEN WriteLine["no value"]
       ELSE IF TypeClass[Domain[type]] = structure THEN WriteLine["a structure value"]
       ELSE ERROR;
      THROUGH [0..depth + 1] DO WriteChar[Ascii.SP] ENDLOOP;
      WriteLine["Its Range..."];
      IF Range[type] = RTTypesBasic.nullType THEN WriteLine["no value"]
       ELSE IF TypeClass[Range[type]] = structure THEN WriteLine["a structure value"]
       ELSE ERROR;
    };
  error =>
    { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
      WriteString["(an error named "];
      WriteString[TVToName[tv]];
      WriteLine[")"];
      THROUGH [0..depth + 1] DO WriteChar[Ascii.SP] ENDLOOP;
      WriteString["Its Domain..."];
      IF Domain[type] = RTTypesBasic.nullType THEN WriteLine["no value"]
       ELSE IF TypeClass[Domain[type]] = structure THEN WriteLine["a structure value"]
       ELSE ERROR;
    };
  structure =>
    { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
      WriteLine["(a structure)"];
      FOR i: CARDINAL IN [1..NComponents[type]] DO
        THROUGH [0..depth] DO WriteChar[Ascii.SP] ENDLOOP;
        WriteString[IndexToName[type, i]];
```

```
        WriteLine[":  "];
        PrintTV[IndexToTV[tv,  i], depth  +  2];
      ENDLOOP ;
    };
  record  =>
    { THROUGH  [0..depth)  DO WriteChar[Ascii. SP] ENDLOOP ;
      WriteLine["(a  record)"];
      FOR i: CARDINAL  IN [1..NComponents[type]]  DO
        THROUGH  [0..depth]  DO WriteChar[Ascii. SP] ENDLOOP ;
        WriteString[IndexToName[type,  i]];
        WriteLine[":  "];
        PrintTV[IndexToTV[tv,  i], depth  +  2];
      ENDLOOP ;
    };
  ref  =>
    { lc: LONG  CARDINAL  =  TVToLC[tv];
      THROUGH  [0..depth)  DO WriteChar[Ascii. SP] ENDLOOP ;
      WriteString["(a  ref)  =  "];
      IF lc =  0 THEN WriteString["NIL"]  ELSE WriteLongOctal[lc];
      WriteLine[""];
--      IF  lc #  0 THEN  PrintTV[Referent[tv],  depth  +  1];
    };
  list  =>
    { lc: LONG  CARDINAL  =  TVToLC[tv];
      THROUGH  [0..depth)  DO WriteChar[Ascii. SP] ENDLOOP ;
      WriteString["(a  list)  =  "];
      IF lc =  0 THEN WriteString["NIL"]  ELSE WriteLongOctal[lc];
      WriteLine[""];
      IF lc #  0 THEN PrintTV[Referent[tv],  depth  +  1];
    };
  pointer  =>
    { lc: LONG  CARDINAL  =  TVToLC[tv];
      THROUGH  [0..depth)  DO WriteChar[Ascii. SP] ENDLOOP ;
      WriteString["(a  POINTER)  =  "];
      IF lc =  0 THEN WriteString["NIL"]  ELSE WriteLongOctal[lc];
      WriteLine[""];
    };
  longPointer  =>
    { lc: LONG  CARDINAL  =  TVToLC[tv];
      THROUGH  [0..depth)  DO WriteChar[Ascii. SP] ENDLOOP ;
      WriteString["(a  LONG  POINTER)  =  "];
      IF lc =  0 THEN WriteString["NIL"]  ELSE WriteLongOctal[lc];
      WriteLine[""];
    };
  character  =>
    { THROUGH  [0..depth)  DO WriteChar[Ascii. SP] ENDLOOP ;
      WriteString["(a  CHAR)  =  '"];
      WriteChar[TVToCharacter[tv]];
      WriteLine[""]
    };
  cardinal  =>
    { THROUGH  [0..depth)  DO WriteChar[Ascii. SP] ENDLOOP ;
      WriteString["(a  CARDINAL)  =  "];
      WriteLongOctal[TVToLC[tv]];
```

```
        WriteLine[""]
      };
    longCardinal =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteString["(a LONG CARDINAL) = "];
        WriteLongOctal[TVToLC[tv]];
        WriteLine[""]
      };
    unspecified =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteString["(an UNSPECIFIED) = "];
        WriteLongOctal[TVToLC[tv]];
        WriteLine[""]
      };
    type =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteLine["(A TYPE)"]
      };
    integer =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteString["(an INTEGER, written as octal) = "];
        WriteLongOctal[TVToLC[tv]];
        WriteLine[""]
      };
    enumerated =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteString["(an enumeration value) = "];
        WriteString[TVToName[tv]];
        WriteLine[""]
      };
    array =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteLine["(an array)"];
        FOR subscript: TV _ First[Domain[type]], Next[subscript] UNTIL subscript = NIL DO
          PrintTV[subscript, depth + 1];
          PrintTV[Apply[tv, subscript], depth + 2];
        ENDLOOP;
      };
    sequence =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteLine["(a sequence). Length = "];
        PrintTV[Length[tv], depth + 2];
--        FOR subscript: TV _ First[Domain[type]], Next[subscript]
--            UNTIL TVToLC[subscript] = TVToLC[Length[tv]] DO
--          PrintTV[subscript, depth + 1];
--          PrintTV[Apply[tv, subscript], depth + 2];
--        ENDLOOP;
      };
    subrange => PrintTV[Coerce[tv, Ground[TVType[tv]]], depth];
    opaque =>
      { THROUGH [0..depth) DO WriteChar[Ascii.SP] ENDLOOP;
        WriteLine["(opaque)"];
      };
  ENDCASE => ERROR;
```

```
END ;
```