

Practical Use of a Polymorphic Applicative Language

Butler W. Lampson
Eric E. Schmidt

Computer Science Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

Assembling a large system from its component elements is not a simple task. An adequate notation for specifying this task must reflect the system structure, accommodate many configurations of the system and many versions as it develops, and be a suitable input to the many tools that support software development. The language described here applies the ideas of λ -abstraction, hierarchical naming and type-checking to this problem. Some preliminary experience with its use is also given.

1. Introduction

Assembling a large system from its component elements is not a simple task. The subject of this paper is a language for describing how to do this assembly. We begin with a necessarily terse summary of the issues, in order to establish a context for the work described later.

A large system usually exhibits a complex *structure*.

It has many *configurations*, different but related.

Each configuration is composed of many *elements*.

The elements have complex *interconnections* [5].

They form a *hierarchy*: each may itself be a system.

The system *develops* during a long period of design and implementation, and an even longer period of maintenance.

Each element is *changed* many times.

Hence there are many *versions* of each element.

Certain sets of compatible versions form *releases*.

Many *tools* can be useful in this development.

Translators build an executable program.

Accelerators speed up rebuilding after a change.

Debuggers display the state of the program.

Databases collect and present useful information [7].

The System Modelling language (SML for short) is a notation for describing how to compose a set of related system configurations from their elements. A description in SML is called a *model* of the system. The development of a system can be described by a collection of models, one for each stage in the development; certain models define releases. A model contains all the information needed by a development tool; indeed, a tool can be regarded as a useful operator on models, e.g., build system, display state or structure, print source files, cross-reference, etc.

In this paper we present the ideas which underlie SML, define its syntax and semantics, discuss a number of pragmatic issues, and give several examples of its use. We neglect development (changes, versions, releases) and tools; these are the subjects of a companion paper [11]. More information about system modelling can be found in the second author's PhD thesis [15], along with a discussion of related work [1, 2, 3, 8, 16].

1.1 Background

SML is the heart of the program development system being built as part of the Cedar project [6] at Xerox PARC. The SML language, and indeed most of the implementation, does not depend on the languages in which the system elements are written. Most of our experience, however, is with elements written in the Cedar language, which is an outgrowth of Mesa [13]; these elements are called *modules*. Mesa (and hence Cedar) has a very general mechanism for interconnecting modules, and hence is a good test of the SML facilities for interconnection.

The Cedar programmer writes SML programs, called *system models*, to specify the modules in his system and

the interconnections among them. These system models are analyzed by a program called the *system modeller* that automates the program development cycle by tracking changes to modules and controlling the compiling and loading of systems.

A system model is a stable, unambiguous representation for a system. It is easily transferred among programmers and file systems. It has a readable text representation that can be edited by a user at any time. Finally, it is usable by other program utilities such as cross-reference programs, debuggers, and optimizers that analyze inter-module relationships.

1.2 Organization of the paper

The remainder of this introduction contains a summary of the basic ideas behind the design and implementation of the SML language; each idea is explained in detail in its proper place later in the paper. Then comes an example of a model for a small but rather intricate Cedar system (§ 2). The next two sections give the syntax and semantics of SML (§ 3) and explain the many uses of functions and function application (§ 4). Then a number of pragmatic issues are discussed: the treatment of files containing system elements, the implementation of the SML evaluator, and experience with the language (§ 5). A conclusion (§ 6) is followed by an appendix with the model for a significant component of the Cedar system.

1.3 Basic ideas

System modelling is based on a few simple ideas which are applied uniformly. Because the semantics are very simple, it is easy to determine the meaning of a model. Conciseness and expressive power come from lambda abstraction and hierarchical naming.

Models are functional. A system is described by a model, which is a functional program, i.e., a program that is executed only for its result and has no side-effects. The result of executing the model is the desired system.

Everything is a value. The primitive values are inherited from Cedar (or any language in which the elements are programmed). SML has three kinds of composite values:

- Functions.

- Bindings, or sets of [name, type, value] triples.

- Declarations, or sets of [name, type] pairs.

An element may have a composite value. In addition, such values can be constructed in SML.

Abstraction is done with functions. A function can take any kind of arguments and return any kind of results. Any SML expression can be turned into a function by λ -

abstraction; some of the many uses of this mechanism are given in § 4.

Structure is expressed by bindings. Any collection of [name, value] pairs can be aggregated into a single value called a binding. Because the values can themselves be bindings, any amount of detail can be subordinated hierarchically. A binding can provide a context for interpreting a hierarchical name such as *Cedar.Compiler.Parser.NextState*.

Checking is done with types. The type of a function is an arrow type ($T \rightarrow U$), which specifies the types its arguments must have and the types of its results. In each application the arguments are checked to ensure they meet the function's requirements. In addition to providing type-checking for SML programs, this also allows any type-checking in the elements to be extended to their interconnections.

The type of a binding is a declaration. In a λ -expression, the required types of the arguments are given by a declaration. This allows the body to be type-checked only once, before the λ -expression is applied.

A model is complete. Abstractly, it contains the entire text of all the elements of the program. In an implementation, the text may be stored in separate files and referenced by name, but the text associated with such a name must never change, so this separation is strictly an implementation technique and does not affect the semantics. Of course a model with parameters is not complete until it is applied.

Only source text is real. The object files output by translators etc. are an implementation technique for *accelerating* the rebuilding of a system after minor changes.

2. An example

This section gives a small but rather intricate example of a model for a Cedar program. For the sake of concreteness, it explained entirely in Cedar-specific terms, rather than in the more general terms of SML. The interfaces, implementations and instances described here are identical to those in Mesa; these structuring methods have been used for several sizable systems, in the range of 100k to 500k lines of source code [9, 10, 12].

A Cedar system consists of a set of modules. There are two kinds of module: *implementation* (PROGRAM) modules, or *interface* (DEFINITIONS) modules. An interface module contains constants (numbers, types, inline procedures, etc.) and declarations for values to be supplied by an implementation (usually procedures, but also types and other values). A module M_1 that calls a procedure in another module M_2 must IMPORT an *instance Inst* of an interface I that declares this procedure. *Inst* must be EXPORTED by the PROGRAM module M_2 . For example, a procedure *SortList* declared in a module *SortImpl* would also be declared

in an interface *Sort*, and *SortImpl* would EXPORT an instance of *Sort*. A PROGRAM calls *SortList* by IMPORTing this instance of *Sort* and referring to the *SortList* component of the instance. We call the importer of *Sort* the *client* module, and say that *SortImpl* (the exporter) implements *Sort*. Of course *SortImpl* may itself IMPORT and use interfaces that are defined elsewhere.

These interconnections are shown in Figure 1, with filenames for each module shown in bold above the module text. The interface *Sort* defines an *Object* composed of a pair [x, y] of coordinates. The exporter, *SortImpl*, declares a procedure *SortList* that takes a list of these objects and sorts them. *ClientImpl* defines a procedure *Test* that calls *SortList* to sort a list of such objects.

Sort.cedar

```
Sort: DEFINITIONS~{
  Object: TYPE~RECORD[
    x, y: INT ];
  CompareProc: TYPE~PROC
  [a, b: Object] RETURNS [BOOL];
  SortList: PROC
  [LIST OF Object, CompareProc]
  RETURNS[LIST OF Object];
}
```

ClientImpl.cedar

```
DIRECTORY
  Sort;

ClientImpl: PROGRAM
IMPORTS Sort~{
  Test: PROC[l: LIST OF Object]~{
    --Call SortList with this list.
    t←Sort.SortList[l, Compare];
    ...
  };
  Compare: CompareProc~{
    --Compares the two objects,
    --returns less, equal or greater.
    ...
  };
}
```

SortImpl.cedar

```
DIRECTORY
  Sort;

SortImpl: PROGRAM
EXPORTS Sort~{
  SortList: PUBLIC PROC[
    l: LIST OF Object,
    Compare: CompareProc]~{
    RETURNS[n: LIST OF Object]~{
      --Code to sort the list l,
      eliminating duplicates
      ...
    };
  };
}
```

Figure 1: An interface, an implementor and a client

2.1 Several interfaces

Usually there is only one version of a particular interface in a Cedar program. Sometimes, however, it is useful to have several versions of the same interface, especially when the interface defines some constants.

On the left in Figure 2 is the *Sort* module from Figure 1, and on the right a similar module that defines an *Object* to be a string instead of a pair of coordinates. A module that uses *Sort* must be compiled with one of the two versions, since the compiler must know the type of each name in an interface. This is *interface parameterization*, since the types

of items from the interface used by a client (*ClientImpl*) are determined by the specific version of the interface (*SortPoints* or *SortNames*). The actual argument supplied for the *Sort* parameter is determined by the model for the program; see below.

SortPoints.cedar

```
Sort: DEFINITIONS~{
  Object: TYPE~RECORD[
    x, y: INT ];
  CompareProc: TYPE~PROC
  [a, b: Object] RETURNS [BOOL];
  SortList: PROC
  [LIST OF Object, CompareProc]
  RETURNS[LIST OF Object];
}
```

SortNames.cedar

```
Sort: DEFINITIONS~{
  Object: TYPE~RECORD[
    x: STRING ];
  CompareProc: TYPE~PROC
  [a, b: Object] RETURNS [BOOL];
  SortList: PROC
  [LIST OF Object, CompareProc]
  RETURNS[LIST OF Object];
}
```

Figure 2: Two versions of the same interface

The parameterization can be arranged differently, to reflect the fact that most of the *SortPoints* and *SortNames* interfaces are identical. On top in Figure 3 is a single *Sort* interface which takes an *ObjectType* interface as a parameter. Below are two versions of *ObjectType*. *Sort[Points]* is the same as *SortPoints* in Figure 2; *Sort[Names]* is the same as *SortNames*. A model using these modules is given in Figure 7.

Sort.cedar

```
DIRECTORY
  ObjectType USING [ObjectType];

Sort: DEFINITIONS~{
  CompareProc: TYPE~PROC
  [a, b: Object] RETURNS [BOOL];
  SortList: PROC
  [LIST OF Object, CompareProc]
  RETURNS[LIST OF Object];
}
```

Points.cedar

```
ObjectType: DEFINITIONS~{
  Object: TYPE~RECORD[
    x, y: INT ];
}
```

Names.cedar

```
ObjectType: DEFINITIONS~{
  Object: TYPE~RECORD[
    x: STRING ];
}
```

Figure 3: A parameterized interface

It might be argued that this example should be done with instance parameters, like the ones in § 2.2 below, and that interface parameters are needed only because part of the implementation (namely the actual value of the type *Object*) was improperly put in the interface. If the interface were a pure declaration, with all values supplied by an implementation, there would be only one *Sort* interface as in Figure 4. Cedar does allow type declarations (rather than actual values such as RECORD[. . .]) to appear in interfaces, and the *Sort* of Figure 4 would actually be fine, because the client does not depend on the type value. In general,

however, constant values in interfaces cannot be avoided in Cedar, because the only way to supply an argument value (such as a type) at compile-time is to put it in an interface (Ada goes half-way by allowing it to be in the **private** part of the package specification). This is probably a deficiency in the Cedar implementation. From the point of view of this paper, however, it illustrates the ability of system modelling to handle an imperfect programming language.

Sort.cedar

```
Sort: DEFINITIONS~{
  Object: TYPE;
  CompareProc: TYPE~PROC
  [a, b: Object] RETURNS [BOOL];
  SortList: PROC
  [LIST OF Object, CompareProc]
  RETURNS[LIST OF Object];
}
```

Figure 4: A single *Sort* interface with a type declaration

Quicksort.cedar

```
DIRECTORY
  Sort;

Quicksort: PROGRAM
  EXPORTS Sort~{
    SortList: PUBLIC PROC[
      l: LIST OF Object,
      Compare: CompareProc]
  RETURNS[nl: LIST OF Object]~{
    --Code to sort the list l,
    eliminating duplicates.
    --Uses Quicksort
    ...
  };
}
```

Heapsort.cedar

```
DIRECTORY
  Sort;

Heapsort: PROGRAM
  EXPORTS Sort~{
    SortList: PUBLIC PROC[
      l: LIST OF Object,
      Compare: CompareProc]
  RETURNS[nl: LIST OF Object]~{
    --Code to sort the list l,
    eliminating duplicates
    --Uses Heapsort
    ...
  };
}
```

ClientImpl.cedar

```
DIRECTORY
  Sort;

ClientImpl: PROGRAM
  IMPORTS Quicksort: Sort, Heapsort: Sort~{
    Test: PROC[l: LIST OF Object]~{
      --Quicksort the list.
      quickL: LIST OF Object ~ Quicksort.SortList[l, Compare];
      --Now try Heapsort.
      heapL: LIST OF Object ~ Heapsort.SortList[l, Compare];
      IF NOT ListEqual[quickL, heapL] THEN ...
      ...
    };

    Compare: CompareProc~{
      --Compares the two objects; returns less, equal or greater.
      ...
    };
  };
}
```

Figure 5: Two implementations of an interface

2.2 Several instances

When there is only one interface, it may have several implementations. For example, a system that uses the left version of the *Sort* interface in Figure 2 might use two different versions of the module that EXPORTS *Sort*, one using the Quicksort algorithm and the other using the Heapsort algorithm to do the sort. Such a system includes both implementors of *Sort*, and must specify which *SortList* routine the clients get when they call *Sort.SortList*[]. In Cedar a client module can IMPORT both versions, as shown in Figure 5.

In Figure 5, *Quicksort* and *Heapsort* each EXPORT a *SortList* procedure to the *Sort* interface: *Quicksort.SortList* uses Quicksort to sort the list; *Heapsort.SortList* uses Heapsort. *ClientImpl* imports each version under a different name, in this case the same names (*Quicksort* and *Heapsort*) used for the implementation modules. The client procedure *Test* calls each *SortList* in turn by specifying the name of the interface and the name of the procedure (e.g. *Quicksort.SortList*[. . .]). This client has three parameters: an interface *Sort* and two instances of *Sort* (*Quicksort* and *Heapsort*). A reasonable application in a model which uses this client might be something like

```
ClientImpl[Sort, Quicksort, Heapsort]
```

ClientImpl.cedar

```
DIRECTORY
  SortPoints: INTERFACE Sort,
  SortNames: INTERFACE Sort;

ClientImpl: PROGRAM IMPORTS
  QuicksortP: SortPoints, HeapsortP: SortPoints
  QuicksortN: SortNames, HeapsortN: SortNames~{

  LP: TYPE~LIST OF SortPoints.Object;
  LN: TYPE~LIST OF SortNames.Object;

  Test: PROC[lp: LP, ln: LN]~{
    --Quicksort the list of points.
    quickLP: LP~QuicksortP.SortList[lp, CompareP];
    --Now try Heapsort.
    heapLP: LP~HeapsortP.SortList[lp, CompareP];
    IF NOT ListEqual[quickLP, heapLP] THEN ...
    ...
    --Now do it for names.
    quickLN: LN~QuicksortP.SortList[ln, CompareN];
    heapLN: LN~HeapsortP.SortList[ln, CompareN];
    IF NOT ListEqual[quickLN, heapLN] THEN ...
    ... };

  CompareP: SortPoints.CompareProc~{
    --Compares the two objects; returns less, equal or greater.
    ...
  };

  CompareN: SortNames.CompareProc~{
    ... };
}
```

Figure 6: A client with two interfaces and four instances

This would not be legal in SML because of the rules for binding function parameters (see § 3.4), but the model for a slightly more complicated system than this one is given at the end of this section.

We can put the examples of Figures 3 and 5 together to get the somewhat contrived client program in Figure 6, which has two interface parameters and four instance parameters. It sorts both lists of *Points* and lists of *Coords*, using both Quicksort and Heapsort. Thus there are two interfaces, for the two types of lists being sorted, and four instances, one for each combination of type and sorting method.

2.3 Parameterization in SML

We are now in a position to understand the model within which the interface modules of Figure 3, the *Sort* implementations of Figure 5, and the client of Figure 6 can be embedded; it is given in Figure 7.

Client.model

```
SortTest ~ [
-- Interfaces
Points: INTERFACE ObjectType ~ @Points.cedar[];
Names: INTERFACE ObjectType ~ @Names.cedar[];
Sort: [INTERFACE ObjectType]→[INTERFACE Sort] ~ @Sort.cedar,
SortP: INTERFACE Sort ~ Sort[Points];
SortN: INTERFACE Sort ~ Sort[Names];
-- Implementations
Quicksort: [I: INTERFACE Sort]→[Inst: I] ~ @Quicksort.cedar,
Heapsort: [I: INTERFACE Sort]→[Inst: I] ~ @Heapsort.cedar,
-- Now the instances and the client.
Client: CONTROL ~ @ClientImpl.cedar[
SortPoints~SortP, SortNames~SortN,
QuicksortP~Quicksort[SortP], QuicksortN~Quicksort[SortN],
HeapsortP~Heapsort[SortP], HeapsortN~Heapsort[SortN] ]
]
```

Figure 7: A system model with several of everything

This model is a binding, which gives values to eight names: *Points*, *Names*, *Sort*, *SortP*, *SortN*, *Quicksort*, *Heapsort*, and *Client*. The first three are bound to the corresponding Cedar modules (the module stored on file *F* is referred to by the expression *@F*, as in *@Points.cedar*). Note that *Sort* is a function, since it takes a parameter (the type of the objects to be sorted) and returns a *Sort* interface. The type INTERFACE *N* is the type of the value returned by a Cedar module which begins *N*: DEFINITIONS; it acts as the bridge between the SML type system and the Cedar type system, which SML does not understand. Then *SortP* and *SortN* are bound to the *Sort* interfaces for points and names, obtained by applying the function *Sort* to the interfaces *Points* and *Names*. They have the same type (INTERFACE *Sort*).

Next the two *Sort* implementations are bound to *Quicksort*

and *Heapsort*. Both of these are functions, since they depend on the *Sort* interface: note that they return (EXPORT) an interface whose type is given by their argument. Finally, the client can be applied to its six arguments. This model does not give names to the four instances of *Quicksort* and *Heapsort*, although it could have done so and then used those names for the arguments to *Client*. Alternatively, we could have refrained from naming *Points* and *Names*, writing instead

```
SortP: INTERFACE Sort ~ Sort[@Points.cedar[]];
SortN: INTERFACE Sort ~ Sort[@Names.cedar[]].
```

We used a binding for the arguments of *Client* to make the correspondence of arguments and parameters clear. If all the arguments have unique types, the parameter names can be omitted in an application; this is the usual case, but this example is different. In addition, any argument can be defaulted to the name of the corresponding parameter; defaulting is specified by writing a * between the function and the arguments. This model would normally be written

```
[...
SortPoints: INTERFACE Sort ~ Sort[Points];
SortNames: INTERFACE Sort ~ Sort[Names];
...
Client: CONTROL ~ @ClientImpl.cedar*[
QuicksortP~Quicksort[SortPoints], QuicksortN~Quicksort[SortNames],
HeapsortP~Heapsort[SortPoints], HeapsortN~Heapsort[SortNames] ]
]
```

with the interface parameters to *Client* defaulted. If we named the instances of *Sort* with the names used in the domain declaration of *ClientImpl*, we could default those parameters also, getting simply

```
Client: CONTROL ~ @ClientImpl.cedar*[]
```

for the client.

The kinds of values in SML follow naturally from the objects being represented.

The value of *@Points.cedar[]* is the file for the interface module *Points.cedar*. When *SortTest* is built, this module will be compiled, and it is actually the resulting object file that is passed as the *Points* argument when *Sort* is compiled; it is possible to think of this object file as the value of *@Points.cedar[]*.

The value of *@Quicksort.cedar* can only be the source file, since the Cedar implementation requires all the interface arguments to be supplied at compile time.

The value of *Quicksort[SortP]* is the instance of *SortP* returned by *Quicksort*. To run the program, we need a representation of this instance which is a record of procedure descriptors produced when the object file gotten by compiling *Quicksort[SortP]* is loaded. Note there are two object files for *Quicksort* in this example; one corresponds to *Quicksort[SortP]* and exports *SortP*, and the other corresponds to *Quicksort[SortN]* and exports *SortN*.

It is instructive to distinguish the two kinds of arguments

by the difference in implementation. An interface argument is supplied to the compiler, which checks the types of the various objects and procedures. An instance argument is supplied when a module is loaded and the imports of other modules are resolved.

The reader who is frustrated by this rather unrealistic example may wish to examine the more realistic one in § 4.3, and the real system model in the Appendix.

3. The SML language

In this section we describe the polymorphic applicative language SML which was illustrated in the previous section. SML was devised to serve two purposes:

It is a notation for describing the composition of a system from its elements.

It is roughly the applicative subset of the Cedar Kernel language.

The Kernel is a small, precisely defined and intuitively simple language, with the property that any Cedar program can be straightforwardly rewritten as a Kernel program. The Kernel program may invoke some procedures from a fixed library; e.g., $i+j$ is rewritten as `INTEGER.PLUS[i, j]`. Eventually, the Kernel will be a subset of Cedar. When this happens, Cedar programmers will write programs and describe systems in the same underlying language.

Thus SML is both independent of Cedar, in the sense that it can be used to compose elements written in any language, and very closely related to it, in the sense that it is a subset of an evolved Cedar language. This paper views SML primarily as a subset of Cedar, but its general utility is discussed in § 3.2 and § 5.

3.1 Concepts

The SML language is built on four concepts:

Application of functions: Any expression can be made into a function by λ -abstraction. Application is the basic method of computing. There are no side-effects; hence an expression can be replaced by its value without changing the meaning of a program.

Values: Everything is a value, including types and functions.

Bindings: [Name, value] pairs can be grouped into sets called bindings; the values are identified by their names. Every name is interpreted by looking it up in some binding.

Types: Every name has a type, and strong type-checking ensures that the value of the name has that type. Function bodies are checked independently of applications.

3.1.1 Application

The basic method of computation in SML (as in Lisp) is by applying a function to argument values. A function is a mapping from argument values to result values.

A function is implemented either by a primitive supplied by the language (whose inner workings are not open to inspection) or by a *closure*. A closure is the value of a λ -expression whose body in turn consists of applications of functions to arguments, e.g. $\lambda[x: \text{INT}] \text{IN } x+y+3$ where x : INT is a (typed) parameter, y is a free name, and $x+y+3$ is the body. A closure is a triple:

parameters: a declaration;

environment: a binding equal to the current environment of the λ -expression;

body: an expression, the body of the λ -expression.

It is helpful to think of a closure as an expression together with values for all its free names except the parameters; hence the term closure. Evaluation always terminates in SML because there are no conditionals; thus an equivalent way to evaluate a λ -expression is to replace all the free names in the body with their current values, and make a closure with an empty environment.

A λ -expression that doesn't return values is useless, since there are no side effects. Application is denoted in programs by expressions of the form $f[\text{arg}, \text{arg}, \dots]$.

3.1.2 Values

An SML program manipulates values. Anything that can be denoted by a name or expression in the program is a value. Thus strings, functions, interfaces, and types are all values. In the SML language, all values are treated uniformly, in the sense that any can be

passed as an argument,

bound to a name, or

returned as a result.

These operations must work on all values so that application can be used as the basis for computation and λ -expressions as the basis for abstraction. In addition, each particular type of value may have its own primitive functions; e.g., equality for most types, plus for integers, etc. None of these operations, however, is fundamental to the language.

SML has very few types of values, since it is not intended for general-purpose computing. The primitive types are `STRING`, `TYPE`, and `INTERFACE n` for any name n . Strings are useful for compiler options and as components of file names. SML also has string literals. E.g., the binding

[x : `STRING ~ "lit"`,

y : `STRING ~ x`]

gives x and y the string value "lit". `TYPE` is the type of a

type, i.e. of STRING, TYPE, INTERFACE n , of a declaration; it is seldom needed in models.

INTERFACE n is the type of the value returned by a Cedar interface module named n ; thus INTERFACE *Sort* is the type of the *Sort* interface returned by the *Sort* modules in § 2. Every value has to have *some* type, and there isn't anything more specific to say about the type of the *Sort* interface. It would be slightly simpler to have a single type INTERFACE which is the type of every interface value. In practice, however, an interface is usually identified by its name, and the more specific type provides a useful check. This point is treated in more detail in § 5.

In addition to values of these primitive types, there are two kinds of composite values:

functions, whose types are *arrow* types, e.g., $T \rightarrow U$;

bindings, whose types are *declarations*.

3.1.3 Bindings and scope

A binding is an ordered set of [name, type, value] triples, often denoted by a constructor like this:

```
[x: STRING ~ "s", y: STRING ~ "t"]
```

or simply

```
[x ~ "s", y ~ "t"].
```

Individual values can be selected from a binding using the "." operation, which is like Pascal record selection: if b is the binding above, then $b.x$ denotes "s" and has the type STRING.

Since the values in a binding can also be bindings, it is possible to have a single binding which is a multi-level, hierarchically named set of values. This is exactly the same power provided by a hierarchical file system, e.g., the one in Unix. Any amount of information can be denoted by a single name. For example, in the Appendix *Cedar* is a binding which contains a large set of Cedar system interfaces.

There are two useful operators for combining bindings. If b_1 and b_2 are bindings with no names in common, then $b_1 + b_2$ is the union; if there are names in common, this expression is an error. This also works for declarations. The expression b_1 THEN b_2 , on the other hand, is never an error; if n appears in both bindings, it has the value $b_1.n$ in the result. Some other, more esoteric operators on bindings are described in § 4.1.

A scope is a region of the program in which the value bound to a name does not change. For each scope there is a binding that determines these values, called the *current environment*. A new scope is introduced by IN following a LET or λ , or by a [. . .] constructor for a binding. LET adds the names in a binding to the current environment:

```
LET  $b$  IN  $exp$ 
```

changes the current environment for exp to b THEN ENV, where ENV is the current environment for the LET expres-

sion. Thus it makes the names in b accessible in exp without qualification. A λ does the same thing when it is applied, using the arguments as the first binding:

```
( $\lambda d$  IN  $exp$ ){ $args$ }
```

is equivalent to

```
LET  $d \sim args$  IN  $exp$ .
```

In a binding constructor b , the current environment is b THEN ENV; thus expressions in the constructor see all the names being bound.

LET expressions are useful for giving names to cumbersome expressions. For example, a set of standard Cedar interfaces can be defined in the file *Cedar.model*:

```
[Rope: INTERFACE Rope ~ @Rope.cedar*[],
```

```
IO: INTERFACE IO ~ @IO.cedar*[],
```

```
Space: INTERFACE Space ~ @Space.cedar*[]]
```

Then a LET expression like

```
LET  $C \sim @Cedar.Model$  IN [ . . .  $C.Rope$  . . . ]
```

is equivalent to

```
[ . . . (@Cedar.Model).Rope . . . ]
```

which in turn is equivalent to

```
[ . . . @Rope.cedar . . . ]
```

It is also possible to make a prepackaged set of definitions directly accessible. For instance, the previous expressions are also equivalent to

```
LET @Cedar.Model IN [ . . . Rope . . . ]
```

(provided *Rope* is not bound in the [. . .]). After the IN the identifiers *Rope*, *IO*, and *Scope* can be used without qualification.

A declaration is the type of a binding. It is an ordered set of [name, type] pairs, often denoted

```
[x: STRING, y: STRING].
```

If d is a declaration, a binding b has type d if it has the same names, and for each name n the value $b.n$ has the type $d.n$.

3.1.4 Types

A type in SML is a *predicate* on values: a function which maps a value into a Boolean. If T is a type and $T[x]$ is true, we say that x has type T . A function f has an *arrow* type $D \rightarrow U$, where D is a declaration. When f is applied to an argument x , we require that x have type D ; thus an argument must be a binding. This is called a *type check*; if it fails, there is a *type error*. The type of $f[x]$ is U . For a function in which the result type depends on the arguments (e.g., an implementation module, which maps an interface I into an instance of type I), U is itself a function, which must be applied to the arguments to yield the result type. Thus in this case the type of $f[x]$ is $U[x]$. Rather than writing the function explicitly, we infer its presence whenever U has a free name declared in D ; thus

```
[x:  $T$ ]  $\rightarrow U$ 
```

is shorthand for

```
[x:  $T$ ]  $\rightarrow (\lambda [x:  $T$ ]  $\rightarrow TYPE$  IN  $U$ ).$ 
```

If f is primitive (e.g., if it is the value of a Cedar module, rather than of a λ -expression in SML) the type check en-

sures that the primitive is getting the kind of arguments it expects. For example, when a module imports an instance of the *SortPoints* interface (see Figure 2), the type check ensures that it will actually get an instance of *SortPoints*, rather than an instance of *SortNames*, or of *BesselFunctions*. Since Cedar itself is strongly typed, it chooses the types of its modules so that this check is sufficient to ensure that the individual procedures and other values in the instance have the proper types. Thus the type-checking of SML extends the type-checking of Cedar to the composition of an entire program.

Within SML itself the main role of types is in the declarations in a λ -expression. In the expression

$$\lambda d_1 \rightarrow d_2 \text{ IN } \textit{exp}$$

the types of the parameters in d_1 are a pre-condition for applying the function: the arguments must have those types. The types of the results in d_2 are a post-condition: the results are guaranteed to have those types. Thus:

The caller must establish the pre-condition (supply arguments of the right types) and may assume the post-condition (count on the types of the results).

Symmetrically, the λ -body may assume the pre-condition (count on the types of the parameters), and must establish the post-condition (supply results of the right types).

Like any pre-conditions and post-conditions, the purpose of the declarations is to allow the body and the application to be checked independently of each other. Once the body has been checked (assuming the declared types for the parameters) it is certain that if an application doesn't cause a type error, then the expression which results from substituting the arguments for the parameters will not cause a type error.

A secondary use of types in SML is in bindings. The binding

$$n: T \sim e$$

is a type error unless e has type T ; if it checks, n has type T in the binding. This is a form of redundancy which is often useful, but it is entirely optional. If T is omitted, then n has the type of e .

To do the type-checking, it is necessary to be able to compute the type of each expression. This is done by the usual induction on the structure of expressions. Every name has a type, because the name is introduced either in a binding or in a declaration. Every literal has a type derived from its syntax; e.g., the type of "abc" is `STRING`. Every Cedar module has a type, whose derivation is discussed in the next section.

The type of an application is computed as described above. The type of a λ -expression $\lambda d_1 \Rightarrow d_2 \text{ IN } e$ is $d_1 \rightarrow d_2$. The type of a binding is the corresponding declaration; the type of a declaration is `TYPE`.

3.2 Values of Cedar modules

SML allows the text of system elements written in any language to be included in a model, as long as there are procedures to:

Turn this text into an SML value.

If it is a function value, apply it to arguments.

These procedures are specific to the language in which the element is written. This language must be identified somehow. A conceptually straightforward way of doing this is to treat the text as a string, and provide a primitive function for each language which converts the string into an SML value. With this convention the expression

```

CEDAR["
  Sort: DEFINITIONS~{
    ...}.
"]

```

is the way to include *Sort.cedar* from Figure 1 in a model. In fact, SML includes text from other languages by naming a file containing the text. The previous example would be written

```
@Sort.cedar
```

where *Sort.cedar* has the contents given in Figure 1. The language is identified by the last component of the file name. This is logically identical to the previous mechanism; its pragmatics are discussed in § 5.

The SML value of a module is always a function. When this function is applied, the result is one of:

another function, with an arrow type;

an interface, with `INTERFACE n` as its type for some n ;

a binding whose values are instances, each with some interface as its type.

Interfaces and instances are opaque in SML; i.e., there is nothing to do with them except to hand them around as uninterpreted values, or pass them as arguments to some function derived from a Cedar module. The functions are also opaque, in the sense that the implementation of application is outside the province of SML, and depends on the implementation of Cedar. The only requirements imposed by SML are that there be no side-effects visible in the model, and that the types supplied for Cedar modules correctly express the Cedar rules for what arguments can properly be given to each function. The operations of compiling and loading modules and establishing linkages are the result of applying these opaque functions to interface and instance arguments.

In order to make the interface between SML and Cedar clearer, we will describe in some detail exactly how opaque SML values are derived from Cedar modules, and how Cedar implements application of opaque functions. When a Cedar module M is compiled, any interface I needed by M (as an argument for a parameter declared in its `DIRECTORY` statement) must be compiled first, and the

compiler must have access to the object file for *I*. All the compile-time external dependencies of a module are specified in this way. When a module is loaded, any instance needed by *M* (as an argument for a parameter declared in its IMPORTS statement) must be satisfied by filling in links in the compiled code with procedure descriptors exported by other modules. All the load-time external dependencies of a module are specified in this way. These relations are expressed in SML by passing as arguments to *M* the values corresponding to an interface (for compilation) or an instance with procedure descriptors (for loading).

Interface parameters

Consider an interface that depends on no other interfaces, i.e. its DIRECTORY statement is empty, and hence it can be compiled without reference to any other modules. SML treats the module containing the interface as a function value. Its type is an arrow type with no parameters and one result, e.g.

```
[] → [INTERFACE Sort]
```

where *Sort* is the name of the interface, as in Figure 1. The application of this function (to no arguments) will result in an object of type INTERFACE *Sort*.

```
SortInterface: INTERFACE Sort ~ @SortModule.cedar[]
```

declares a name *SortInterface* that can be used to specify the dependency of other modules on this interface. An interface *BTree* defined in the module *BTree.cedar* that depends on *Sort* would have a type like

```
[SortParameter: INTERFACE Sort] → [INTERFACE BTree]
```

To express this dependency in the model, we apply the *BTree* module value to the *Sort* interface value:

```
BTreeInterface: INTERFACE BTree ~
@BTreeModule.cedar[SortParameter~SortInterface];
```

In this example we used different names for various entities having to do with the *Sort* and *BTree* interfaces:

SortInterface and *BTreeInterface* for the SML names to which the interface values are bound;

Sort and *BTree* for the names of the interface types: INTERFACE *Sort* and INTERFACE *BTree*;

SortModule.cedar and *BTreeModule.cedar* for the names of the (files containing the) Cedar modules which define the interfaces;

SortParameter for the name of the parameter to *BTreeModule* which has type INTERFACE *Sort*.

Normally the same name is used for all these purposes, so the two bindings in the last paragraph would be

```
Sort: INTERFACE Sort ~ @Sort.cedar[]
BTree: INTERFACE BTree ~ @BTree.cedar[Sort~Sort];
```

Instance parameters

An instance of an interface that is EXPORTed is represented as a record that contains procedure descriptors, etc. These procedure names are declared in the inter-

face being exported and in the exporting PROGRAM module. We can think of the interface module as a declaration for this record. Consider the implementation module *SortImpl* in Figure 1. *SortImpl* exports an instance of the *Sort* interface and calls no procedures in other modules (i.e. has no imports). This module has the arrow type

```
[Sort: INTERFACE Sort] → [SortInst: Sort]
```

and can be used as follows:

```
Sort: INTERFACE Sort ~ @Sort.cedar[];
SortInst: Sort ~ @SortImpl.cedar[Sort~Sort];
```

which declares first a name *Sort* of type INTERFACE *Sort* whose value is the interface defined by *Sort.cedar*, and then a name *SortInst* of type *Sort*, whose value is the instance exported by *SortImpl.cedar*. If *SortImpl* imported an instance for *BTree*, then the type would be

```
[Sort: INTERFACE Sort, BTree: INTERFACE BTree, BTreeInst: BTree] →
[SortInst: Sort]
```

and the exported instance would be computed by

```
SortInst: Sort ~ @SortImpl.cedar[Sort, BTree, BTreeInst].
```

Here in the argument [*Sort*, *BTree*, *BTreeInst*] we have omitted the parameter names; see § 4.1 for the semantics of this.

3.3 Syntax

SML is described by the BNF grammar below. Whenever "x, ..." appears, it refers to 0 or more occurrences of *x* separated by commas. "|" separates different productions for the same non-terminal. Words in which all letters are capitalized are terminal symbols which are reserved words in the language; punctuation symbols other than ::=, | and ... are also terminals. Words that are all lower case are non-terminals; the definitions for the following non-terminals are omitted:

- name, which stands for an name,
- string, which stands for a string literal in quotes, and
- filename, which stands for a string of characters that are legal in a file name.

Subscripts are used to identify specific non-terminals, so they can be referenced without ambiguity in the accompanying explanation.

```
exp ::= λ exp1 ⇒ exp2 IN exp3
      | LET exp1 IN exp2
      | exp1 exp2
      | exp1 * exp2
      | exp1 infixOp exp2
      | exp1 . name
      | [ exp1, ... ]
      | [ decl ]
      | [ binding ]
      | name
      | string
      | ENV
      | INTERFACE name
      | STRING | TYPE
```

```

      | exp1 → exp2
      | @ filename
decl ::= declElement, ...
declElement ::= name : exp
binding ::= bindElement, ...
bindElement ::= [ decl ] ~ exp2
               | name : exp1 ~ exp2
               | name ~ exp2
               | [ name, ... ] ~ exp2
infixOp ::= + | - | / | \ | ↑ | THEN

```

3.4 Semantics

The value of an SML expression is defined by induction on the syntax. The evaluation rules are those of the λ -calculus, together with definitions of the primitives for handling types, declarations and bindings. In this section ENV stands for the current environment.

Lambda

```
exp ::=  $\lambda$  exp1  $\Rightarrow$  exp2 IN exp3
```

The exp_1 is evaluated and must yield a declaration d . The value of exp is a closure, consisting of:

- the *parameters*, which are the declaration d ;
- the *environment* binding, which is ENV;
- the *body*, which is the expression exp_3 .

The type is $exp_1 \rightarrow exp_2$. The exp type-checks if in every environment P THEN ENV, where P is a binding of type d ,

- exp_2 and exp_3 type-check, and
- exp_3 has the type exp_2 .

In other words, the result and body must type-check for *any* arguments which satisfy the parameter declaration, and the body must have the type specified as the result type of the λ -expression. This check is implemented by constructing a P with each name bound to a value different from any other value.

LET

```
exp ::= LET exp1 IN exp2
```

The exp_1 is evaluated and must yield a binding b . The type and value of exp are the type and value of exp_2 in the environment b THEN ENV.

Application

```
exp ::= exp1 exp2
      | exp1*exp2
```

The exp_1 and exp_2 are evaluated to yield a function f of type $D \rightarrow U$ and a value x ; usually exp_2 has the form [binding]. Then for the first alternative, the expression $D \sim x$ is evaluated to yield a binding $args$; of course this expression

must type-check. For the second alternative, the expression $x + (ENV \uparrow (D - x))$ is evaluated to yield a binding $args$; see § 4.1 for the meaning of these operators. The type of exp is U , or $U[args]$ if U is a function (see § 3.1.4 for a discussion of this case).

For the value there are two cases to consider:

1. The function f is a closure, i.e., the value of an SML λ -expression. If the closure has environment E and body $body$, then the value of exp is the value of
 $LET (args \text{ THEN } E) \text{ IN } body$
2. The function is a primitive, either "built-in" or more likely, derived from a Cedar module (see § 3.2). The value is whatever result the primitive computes from the argument $args$; a primitive is responsible for ensuring that the result has the proper type.

Infix operators

These are discussed in § 4.1.

Dot, group, declaration and binding

```
exp ::= exp1.name
```

The exp_1 must evaluate to a binding b , whose type is a declaration d . The type of exp is the type of the name in d , and the value is the value of the name in b .

```
exp ::= [ exp1, ... ]
```

This is a constructor for a *group*, which is a binding with anonymous names for the elements.

```
exp ::= [ decl ]
decl ::= declElement, ...
declElement ::= name : exp1
```

The type of exp is TYPE. Its value is the set of [name, type] pairs obtained by evaluating the exp_1 expressions in the declElements, and pairing them with the corresponding names.

If a binding is used where a declaration is required (e.g., in a λ , or after a colon), then it is coerced to a declaration in the obvious way. Thus $[x \sim T, y \sim U]$ is coerced to $[x: T, y: U]$. Of course, this fails unless each component in the binding is a type.

```
exp ::= [ binding ]
binding ::= bindElement, ...
bindElement ::= [ decl ] ~ exp2
                | name : exp1 ~ exp2
                | name ~ exp2
                | [ name, ... ] ~ exp2
```

In the first alternative for bindElement, decl is evaluated to a declaration d and exp_2 must evaluate to a binding or group b with type d ; the bindElement binds each name n in d to the value of n in b . If n does not appear in b there is a type error. If b is a group its elements must all have

different types, the elements of d must all have different types, and the `bindElement` binds each name n in d to the value in b with the same type; if there is no such value, there is a type error.

The second alternative allows the brackets around a single-component declaration and group to be omitted. If only a name appears before the \sim , as in the third alternative, the type is inferred from that of exp_2 . The fourth alternative is the same, except that exp_2 must evaluate to a binding or group with the same number of components as there are names in the brackets, much like the first alternative.

Note that because of the definition of application, these rules are also used for binding arguments to function parameters.

Names and literals

`exp ::= name`

The type and value of exp are the type and value of `ENV.name`.

`exp ::= string`

A string literal like "abc" is a primitive value.

`exp ::= INTERFACE name`

This literal primitive type is discussed in § 3.2. Note that *name* here is part of the literal, and is *not* looked up in `ENV`.

`exp ::= STRING | TYPE`

Literals denoting primitive types.

`exp ::= exp1 → exp2`

The exp_1 must evaluate to a declaration d . The value of exp is a function type T . If f has type T , it takes values of type exp_1 , and the type of $f[x]$ is $(\lambda d \rightsquigarrow \text{TYPE IN } \text{exp}_2)[x]$; see § 3.1.4. If g has type $T \rightarrow U$, then `DOMAIN[g]` is T and `RANGE[g]` is U .

`exp ::= @ filename`

This expression is shorthand for the text stored in file *filename*. If the file contains a model, then exp can be replaced by the contents of the file. If it contains a Cedar module, then the type and value of exp are derived from that module as described in § 3.2.

4. Functions and arguments

In addition to providing the basic abstraction mechanism of SML, functions and function application play a number of important roles in the practical use of models:

They allow the *interconnections* among modules to be expressed, even when there are multiple versions of interfaces and instances; see § 2 for examples.

They allow the relation between a model and the *environment* it depends on to be expressed: a model

with free names can easily be converted by λ -abstraction into one with no free names, in which all dependence on the environment is explicit in the parameter declaration, and the nature of the model's value is explicit in the result declaration. An application of the function makes the choice of environment explicit. See the *BTree* models later in this section, and the Appendix, for examples.

They allow different *configurations* of a system to be produced. For instance, by parameterizing a model with a *DiskDriver* instance, different configurations of the system which use different disk drives can be produced by applying the model to different *DiskDriver* instances.

They allow the choice of translator for a system element to be made explicit, and make it easy to specify parameters of the translation (e.g., target machine, optimization level, etc.). See § 5 for further discussion of this point.

In this section we present some SML facilities for manipulating functions and their applications which are not generally needed for ordinary computation, but are useful in writing models.

We begin by reviewing the rules for binding arguments to parameters given in § 3.4. When the argument is a binding, its elements are matched by name with the parameters. Thus if P is bound by

$$P \sim \lambda[x: \text{STRING}, y: \text{INTERFACE } Y] \Rightarrow [z: \text{INTERFACE } Z] \text{ IN } [\dots]$$

then P takes two arguments. Suppose we also have y bound by

$$y: \text{INTERFACE } Y \sim @Y.cedar[],$$

The arguments to P may be specified as a binding:

$$z: \text{INTERFACE } Z \sim P["lit", y \sim y]$$

Alternatively, since the types are all distinct (and this is the normal case; multiple versions of interfaces or instances are not too common), the arguments may be specified as a group:

$$z: \text{INTERFACE } Z \sim P["lit", y]$$

and they are matched by type to the parameters. In both cases the elements of the argument binding or group are matched to the elements of the parameter declaration based on some distinguishing property; the order of "lit" and y does not matter in either example. The reason for the absence of binding by position in the binding is that argument lists are often rather long.

4.1 Defaulting

Another feature of SML which is motivated by long argument lists is the *defaulted* form of application: $\text{exp}_1^* \text{exp}_2$. Defaulting allows the programmer to omit many parameters; a missing parameter named n is supplied as `ENV.n`, where `ENV` is the current environment. Thus with the binding for y above, and

$$x: \text{STRING} \sim "lit"$$

in ENV, the expression $P^*[]$ is equivalent to $P[x, y]$. Of course, if there is no binding for n in ENV, or if it has the wrong type, there is an error, since this is just a shorthand. The model at the end of this section gives other examples of defaulting.

Note that in a model consisting of a single large binding with all the elements as its values, defaulting all the arguments corresponds to the interconnection rule used by most linkers, which connects all the external references to a name n with the single definition of n .

To define the defaulting rule precisely, we need to introduce a few operators on declarations and bindings. The *union* operator $+$ was defined in § 3.1.3; it combines two declarations or bindings with no names in common. The *restriction* operator \uparrow takes a binding as the first operand and removes all the names which do *not* appear in the declaration which is the second operand. Thus

$[x: \text{STRING} \sim \text{"lit"}, y: \text{INTERFACE } Y \sim \text{Def}] \uparrow [x: \text{STRING}]$
is equal to
 $[x: \text{STRING} \sim \text{"lit"}]$.

There is an error unless the resulting binding $b \uparrow d$ has type d , i.e., unless each name in d actually appears in b with the corresponding type.

The same operator also works with a declaration as the first operand, removing all the names which do *not* appear in the binding which is the second operand. Thus

$[x: \text{STRING}, y: \text{INTERFACE } Y] \uparrow [x: \text{STRING} \sim \text{"lit"}]$
is equal to
 $[x: \text{STRING}]$.

In this form b must have type $d \uparrow b$; i.e., each name n in b must actually appear in d , and b, n must have the type declared for n in d .

The *exclusion* operator $-$ takes a declaration as the first operand and removes all the names which *do* occur in the declaration or binding which is the second operand. Thus

$[x: \text{STRING}, y: \text{INTERFACE } Y] - [x: \text{STRING}]$
is equal to
 $[y: \text{INTERFACE } Y]$

and so is

$[x: \text{STRING}, y: \text{INTERFACE } Y] - [x: \text{STRING} \sim \text{"lit"}]$.

Like $d \uparrow b$, $d - b$ is an error unless b has type $d \uparrow b$, or equivalently

$$d = (d - b) + (d \uparrow b)$$

Similarly, $d_1 - d_2$ is an error unless

$$d_1 = (d_1 - d_2) + d_2.$$

Now we can define application with defaulting in terms of ordinary application: f^*b is equal to

$$f(b + (\text{ENV} \uparrow (\text{DOMAIN}[f] - b)))$$

where $\text{DOMAIN}[f]$ is the domain declaration for the function f (see § 3.4).

4.2 Splitting

SML also provides two operators \backslash and $/$ which *split* a function that takes several arguments into one that takes some of the arguments and returns a function that takes the others. Thus

$P: [x: \text{STRING}, y: \text{INTERFACE } Y] \rightarrow [z: \text{INTERFACE } Z] \sim \dots$
can be split to yield

$PSplit: [x: \text{STRING}] \rightarrow ([y: \text{INTERFACE } Y] \rightarrow [z: \text{INTERFACE } Z]) \sim$
 $P \backslash [x: \text{STRING}]$

The split function can then be applied once, leaving a *curried* function which takes fewer arguments than the original one, because some of the argument values have been fixed. For example, with the bindings for y and $PSplit$ given above,

$PI: [y: \text{INTERFACE } Y] \rightarrow [z: \text{INTERFACE } Z] \sim PSplit[\text{"lit"}],$
 $z: \text{INTERFACE } Z \sim PI[y]$

binds z to the same value as before, but does it in one extra step.

The main application of splitting and subsequent currying is to fix the interface arguments of a Cedar module without fixing the instance arguments. This makes for a clearer model when modules are applied to several implementations of the same interface. It also reflects the realities of the implementation, in which compilation automatically fixes the interface arguments, but instances are bound only on loading.

There are two splitting operators. $f \backslash d$ is a function of type

$d \rightarrow ((\text{DOMAIN}[f] - d) \rightarrow \text{RANGE}[f]);$
it leaves d "on top". Symmetrically, f / d has type

$(\text{DOMAIN}[f] - d) \rightarrow (d \rightarrow \text{RANGE}[f]);$
it leaves d "underneath," and can be pronounced "*f* keeping d ." Both require that $\text{DOMAIN}[f] - d$ be legal, i.e. that the splitting declaration be a subset of the function's domain. The precise definition of f / d is

$$\lambda \text{DOMAIN}[f] - d \Rightarrow (d \rightarrow \text{RANGE}[f]) \text{ IN } (\lambda d \Rightarrow \text{RANGE}[f] \text{ IN } f^*[])$$

The defaulted application collects the argument values for f from the argument bindings of the two nested λ -expressions. There is a similar definition for $f \backslash d$, but more enlightening is this one:

$$f / (\text{DOMAIN}[f] - d)$$

Splitting and defaulting can be used together, of course, so that the last example with PI is equivalent to:

$PI: [y: \text{INTERFACE } Y] \rightarrow [z: \text{INTERFACE } Z] \sim (P / [y: \text{INTERFACE } Y])^*[],$
 $z: \text{INTERFACE } Z \sim PI^*[]$

In fact, the declaration for PI is unnecessary: we can write this as:

$PI \sim (P / [y: \text{INTERFACE } Y])^*[],$
 $z: \text{INTERFACE } Z \sim PI^*[]$

The $/$ operator makes explicit the parameters remaining for PI ; the other way of writing it is less clear without the declaration:

$$PI \sim (P \backslash [x: \text{STRING}])^*[].$$

4.3 A more realistic example

The B-tree package presented in this section is a small system, but one which displays most of the features of larger ones. It consists of an implementation module in the file *BTreeImpl.cedar* and an interface *BTree* that *BTreeImpl* exports. There is no client of *BTree* in the example; the model returns a function which, when given suitable arguments, returns the *BTree* interface and an instance which implements it. A client model would have a reference to this model and a client for the interface.

The *BTree* interface uses some constants found in *Ascii*, which contains names for the ASCII character set. The *BTreeImpl* module depends on the *BTree* interface (since it exports it), and it uses three standard Cedar interfaces: *Rope* defines procedures to operate on immutable, garbage collected strings. *IO* defines procedures to read and write formatted data to a stream. *Space* defines procedures to allocate Cedar virtual memory for large objects, in this case the B-tree pages. Figure 8 is a first version of the package.

BTree1.model

```
LET [
  Ascii: INTERFACE Ascii ~ @Ascii.cedar[],
  Rope: INTERFACE Rope ~ @Rope.cedar*[],
  IO: INTERFACE IO ~ @IO.cedar*[],
  Space: INTERFACE Space ~ @Space.cedar*[] ] IN
BTreeProc ~
λ [RopeInst: Rope, IOInst: IO, SpaceInst: Space]
⇒ [BTree: INTERFACE BTree, BTreeInst: BTree] IN [
  BTree: INTERFACE BTree ~ @BTree.cedar[Ascii],
  BTreeInst: BTree ~ @BTreeImpl.cedar
  [BTree, Rope, IO, Space, RopeInst, IOInst, SpaceInst]
]
```

Figure 8: The fully-expanded B-tree model

This model, stored in the file *BTree1.model*, describes a B-tree system composed of one interface *BTree* and a single implementation module for it. The first four lines declare four names used later. *Ascii* just defines some constants, and needs no arguments; the arguments to the other interface modules are defaulted to reduce clutter. Note that the types are optional; these lines could read:

```
Ascii ~ @Ascii.cedar[],
Rope ~ @Rope.cedar*[],
IO ~ @IO.cedar*[],
Space ~ @Space.cedar*[]
```

since the types can be determined from the values.

Next we bind a name *BTreeProc* to a function with three instances as parameters. If those are supplied, the function will return an interface for the B-tree package, and an instance of that interface. Within the body of the λ -expression which defines this function, there are bindings for the identifiers *BTree* and *BTreeInst*. Here again the types could be omitted.

The value of *BTree1.model* is a binding of *BTreeProc* to a function. Another model might refer to the B-tree package by

```
[BTree, BTreeInst] ~
(@BTree1.model).BTreeProc[RopeInst, IOInst, SpaceInst]
```

The individual treatment of *Ascii*, *Rope*, *IO*, and *Space* is clumsy, and it would be even more clumsy if there were twenty such interfaces instead of four. To make this neater, we can construct a binding for these names, and refer to it in *BTree.model*. Figure 9 shows this, together with defaulting of the interface arguments to *BTreeImpl*.

Cedar.model

```
[ Ascii: INTERFACE Ascii ~ @Ascii.cedar[],
  Rope: INTERFACE Rope ~ @Rope.cedar*[],
  IO: INTERFACE IO ~ @IO.cedar*[],
  Space: INTERFACE Space ~ @Space.cedar*[] ]
```

BTree2.model

```
LET @Cedar.model IN
BTreeProc ~
λ [RopeInst: Rope, IOInst: IO, SpaceInst: Space]
⇒ [BTree: INTERFACE BTree, BTreeInst: BTree] IN [
  BTree: INTERFACE BTree ~ @BTree.cedar[Ascii],
  BTreeInst: BTree ~ @BTreeImpl.cedar[RopeInst, IOInst, SpaceInst] ]
```

Figure 9: The B-tree model with interfaces separated

The prefix of *BTree1* is split into a separate file called *Cedar.model*. Now *BTree2.model* contains a LET statement that makes the values in *Cedar* accessible in *BTree*. Dividing *BTree1* into two models like this makes it possible to establish standard naming environments, such as a binding that names the commonly-used Cedar interfaces. The Appendix has a bigger example. Programmers are free to redefine these bindings in their models; the operators on bindings defined in § 4.1 make this easy.

The idea can be carried further by defining another binding with the standard implementations of the interfaces. Figure 10 shows how this is done. It also replaces the inclusion of the standard models in *BTree* with parameters, so that the dependence of *BTree* on the environment is made explicit. *BTree3* has in its text (including text incorporated by the @ construct) only the B-tree package itself. We can apply *BTree3* to get a *BTree* interface and instance:

```
LET [Interfaces~@Cedar.model] IN
@BTree2.model[Interfaces, @CedarInsts.model[Interfaces] ]
```

This is still clumsy in two ways: the four-component type for the *Interfaces* parameter of *BTreeProc*, and the three separate instance parameters to *BTreeImpl*; both of these would be much longer in a larger system.

The first problem cannot be solved without giving up the idea of type-checking a λ -expression independently of its applications. If we write a more general type for *Interfaces*,

Cedar.model

```
[ Interfaces ~ [  
  Ascii: INTERFACE Ascii ~ @Ascii.cedar[],  
  Rope: INTERFACE Rope ~ @Rope.cedar*[],  
  IO: INTERFACE IO ~ @IO.cedar*[],  
  Space: INTERFACE Space ~ @Space.cedar*[] ] ]
```

CedarInsts.model

```
λ [Interfaces: [Ascii: INTERFACE Ascii, Rope: INTERFACE Rope,  
  IO: INTERFACE IO, Space: INTERFACE Space] ⇒ Interfaces IN  
  [Ascii, Rope, IO, Space] ~ LET Interfaces IN [  
    @AsciiImpl.cedar[],  
    @RopeImpl.cedar*[],  
    @IOImpl.cedar*[],  
    @SpaceImpl.cedar*[] ]
```

BTree3.model

```
[ BTreeProc ~  
  λ [ Interfaces: [Ascii~INTERFACE Ascii, Rope~INTERFACE Rope,  
    IO~INTERFACE IO, Space~INTERFACE Space],  
    Inst: Interfaces]  
  ⇒ [BTree: INTERFACE BTree ~ @BTree.cedar[Ascii],  
    BTreeInst: BTree ~ @BTreeImpl.cedar*[Inst.Rope, Inst.IO,  
    Inst.Space] ] ]
```

Figure 10: Standard interfaces and instances as parameters

such as DECLARATION, then there is no way to check an expression like *Interfaces.Ascii* without the argument. This problem also arises in Cedar itself, where a vague type (at the Cedar level) like `INTERFACE Rope` prevents type-checking of a module like *BTreeImpl* until the argument is supplied, e.g., as `@Rope.cedar*[]`. This is the main reason that compilation, which includes type-checking, requires access to all the interfaces used by a module. Thus Cedar in effect has two kinds of λ -expression:

- the ordinary kind, written as an ordinary procedure body, or as the IMPORTS statement of a module;

- an unchecked kind, written as the DIRECTORY statement of a module.

Currently SML does not have an unchecked λ -expression.

The second problem cannot be solved by prefixing `LET Inst` and defaulting the instance arguments, since they have the same names in *Inst* as the interfaces; if the elements of *Inst* are given different names it won't have *Interfaces* as its type. An attractive solution is to move down into Cedar modules the notion of collecting interface and instance parameters into bindings. Thus instead of a *BTreeImpl* with seven parameters (*BTree*, *Rope*, *IO*, *Space*, *RopeInst*, *IOInst* and *SpaceInst*), we would have one with three parameters (*BTree*, *Interfaces*, and *Instances*). To make it clear which parts of these large bindings are actually used, we can modify the DIRECTORY statement according to this example:

```
DIRECTORY Interfaces USING [Rope, IO, Space]
```

Cedar already has this facility for specifying which names in a particular interface are used, so this is a natural extension.

5. Pragmatics

This section discusses a number of pragmatic issues in the use and implementation of SML, and summarizes our experience with a preliminary version of the Modeller.

5.1 Files

We take the view that the software of a system is completely described by a single unit of text. An appropriate analogy is the way a card deck was used to run a program on a bare computer or under an operating system like FMS that had no file system. Everything is said explicitly in such a system description; there is no operator intervention to supply compiler switches or loader options after the GO button is pressed, and no dependence on a changing environment. In such a description there is no question about when to recompile something, and version control is handled by distributing copies of the deck with a version number written on the top of each copy, and a diagonal stripe of marker which makes it easy to tell whether the deck has been changed.

The monolithic nature of a card deck makes it unsuitable for a large system. In 1982 a system is specified by text which is stored in files. This provides modularity in the physical representation: a file can name other files instead of literally including their text. In Cedar, these files hold the text of Cedar modules or system models. This representation is convenient for users to manipulate; it allows sharing of identical objects, and facilitates separate compilation. Unless care is taken, however, the integrity of the system will be lost, since the contents of the named file may change.

To prevent this, we abstract files into *objects*, which are simply pieces of text. We require that names be *unique* and objects be *immutable*. By this we mean that:

- Each object has a unique name, never used for any other object. The name is stored as part of the object, so there is no doubt about whether a particular collection of bits is the object with a given name. A name is made unique by appending a *unique identifier* to a human-sensible string.

- The contents of an object never change once the object is created. The object may be erased, in which case the contents are no longer accessible. If the file system does not guarantee immutability, it can be ensured by using a suitable checksum as the unique identifier of the object.

These rules ensure that a name can be used instead of the text of an object without any loss of integrity, in the sense that either the entire text of a system will be correctly assembled, or the lack of some object will be detected.

With these conventions, a model can incorporate the text of an object by using the name of the object. This is done in SML by writing an object name preceded by an @. The meaning of an SML expression containing an @-expression is defined to be the meaning of an expression that replaces the @ expression by its contents. For example, if the file *inner.model* contains

```
"lit"
which is an SML expression, the binding
```

```
[x: STRING ~ @inner.sm,
```

```
y: STRING ~ "lit"]
```

has identical values for *x* and *y*.

As discussed in § 3.2, if the object *O* is not an SML expression but a Cedar module, or an element written in some other language, it is turned into an SML expression by conceptually surrounding it with a text-to-SML value conversion function, e.g., CEDAR["*O*"].

It is not essential that the text of a system element be source text; all that is needed is a way to turn it into an SML value. For a Cedar source module, this is done by parsing the DIRECTORY, IMPORTS and EXPORTS statements at the start of the module. But it can also be done for a Cedar object module, which is the output of the compiler and has all its interface parameters bound; object modules have enough information (originally for the benefit of the loader) to allow an SML INTERFACE or function value to be derived. This is sometimes convenient when dealing with a system in which some elements come from an outside organization in object form only.

5.2 Miscellaneous problems

SML provides straightforward solutions to a number of problems which have arisen in constructing Cedar systems.

Translators

Cedar programmers may use a number of programs that analyze a source program written in some language, and produce new source programs. For example, an LALR(1) parser generator called PGS takes a Cedar source file with a grammar embedded in it as stylized comments, and produces:

Tables, a Cedar source file for an interface which defines the structure of the parsing tables for this grammar.

TablesImpl, a Cedar object file containing parsing tables that can be loaded.

Actions, a Cedar source file which is a modification of the input, containing:

code for the semantic actions, which is copied from the input;

code to call the parser, supplied by PGS to supplement replace the comments, which contained the grammar.

Another example is a remote procedure call stub generator [14] that takes the source for a Cedar interface, and produces four source files that must all be compiled. In each of these cases the output files depend on the input file, and if the input file were modified, the preprocessor would have to be run again.

For each language in which a system element is written, we need a way to derive an SML value from an object in the language. For example, a *pgs* object is a function with the type

```
[]→[Tables: []→[T: INTERFACE PGSTables],
TablesImpl: Tables,
Actions: ActionsInterface+ OtherParameters
+ [ T: INTERFACE PGSTables, TImpl: T,
Parser: INTERFACE Parser, ParserImpl: Parser ]
→[AI: ActionsInterface]]
```

Here *ActionsInterface* is exported by the *Actions* module, and *OtherParameters* is a declaration for any other parameters of that module. *Actions* also has the tables and the parser itself as parameters.

Such a function might be applied like this:

```
[ PascalTables: []→[T: INTERFACE PGSTables],
PascalTablesImpl: PascalTables,
PascalActions: [PascalParser: INTERFACE PP, X: INTERFACE X, XI: X,
T: INTERFACE PGSTables, TImpl: T,
Parser: INTERFACE Parser, ParserImpl: Parser ]
→[PascalParserImpl: PascalParser ]]~
@PascalGrammar.pgs[]
```

Here the types are redundant and included for clarity; we could have written

```
[ PascalTables, PascalTablesImpl, PascalActions ]~@PascalGrammar.pgs[]
```

with the same effect. In either case, the model can now proceed to apply *PascalActions*. Supposing we have a suitable binding for *PascalParser*, the interface implemented by this parser, and for *Parser*, *ParserImpl*, *X* and *XI*, we can default these and write

```
PascalParserInstance: PascalParser ~
PascalActions*[T~PascalTables[]], TImpl~PascalTablesImpl]
```

which leaves us with an instance *PascalParserInstance* of the *PascalParser*, which can be returned from the model, or passed to another component.

The code that derives SML values from PGS objects gets control when *@PascalGrammar.pgs* is applied. It is responsible for invoking the PGS preprocessor and deriving SML values from the files that PGS produces. Since these are ordinary Cedar files, code to derive SML objects from them already exists.

In some cases, when a single object can be translated in several ways, it may be better to apply a translation function to it explicitly in the model. This is the case for the RPC stub generator, since it processes an ordinary Cedar interface module, which might also be treated in the usual way. So we might write

```
RPCSubGenerator{@UpdateFiles.string}
```

Compiler Options

Certain aspects of the Cedar compiler's execution can be controlled by specifying compiler options. When the compiler is run from the operating system's command processor, these options are given as command line *switches* consisting of a single letter. For example, "j" instructs the compiler to perform a cross-jumping optimization on the code it generates, "b" instructs it to check for bounds faults, etc. Since the behavior of a system depends on these options, they are treated like any other parameters. The function type of a Cedar module includes a STRING parameter *options* which can be specified explicitly; e.g.,

```
QuicksortPoints: SortPoints ~
```

```
@Quicksort.cedar{options~"j".Sort~SortPoints}
```

If *options* is missing, it is defaulted automatically, in a slight departure from normal SML semantics. The model can supply a binding; if it does not, there is a global binding for this name.

Multiple Exports

We have described systems where there is one exporter of an interface and one or more importers. It is possible to split the implementation of an interface among several modules, and merge the exported instances together. This often happens when the implementation becomes very large and is split by the programmer. Because instances are actually bindings in Cedar, with essentially the same semantics as SML bindings, it is convenient to extend the + and THEN operators (§ 3.1.3) to them. Usually the instances export disjoint names, so that + is the proper operator:

```
BTreeImpl: BTree ~ @BTreeImplA.cedar*[] + @BTreeImplB.cedar*[]
```

5.3 Implementation

The implementation of the Modeller has three quite distinct parts:

The *language* implementation: parsing, prettyprinting and evaluation of SML expressions.

The *bridges* to the programming languages for elements. The Cedar bridge, for example, derives an SML value from a Cedar module, and when this value is a function knows how to apply it by invoking the Cedar compiler or loader.

The *administrator*, which retrieves the value of an object from the file system, manages the cache of object

files, notices changes to elements and updates the models accordingly, etc.

We will discuss only the language implementation here.

Since SML has no iteration constructs and no recursively-defined functions, the evaluator can expand each application of a closure by β -reduction, replacing it by the closure body with formals replaced by actuals. Similarly, an @ reference to a sub-model can be replaced by the text of the referent. This process of substitution must be applied recursively, as the expansion of a λ -expression may involve expansion of inner λ -expressions. The evaluator does this expansion by copying the body of the λ -expression, and then applying itself recursively after adding the argument binding for the application to ENV.

ENV is maintained as a tree of bindings in which each level corresponds to a [. . .] binding constructor, a binding added by a LET statement, or an argument binding. Bindings are represented as lists of triples of [name, type, value]. A closure is represented as a quadruple [parameter declaration, result declaration, body, environment]. As explained in § 3.4, in an application the body is evaluated with ENV equal to *args*+*E*, where *args* is the argument binding and *E* is the environment from the closure. An @-expression is represented by a pointer to the disk file named, together with its type and, for a function, a procedure for applying it. A interface value is represented as a pair [module name, pointer to module file], and an instance value as a pair [pointer to procedure descriptors, pointer to loaded module].

The substitution property of Russell [4] guarantees that variable-free expressions can be replaced by their values without altering the semantics of Russell programs. Since SML programs have no variables and allow no recursion, the substitution property holds for SML programs as well. This implies that the type-equivalence algorithm for SML programs always terminates, since the value of each type can always be determined statically.

5.4 Experience

The SML language, in a somewhat different form, has been used by about five programmers in the past year, and supports the development of systems ranging from 1k to 50k lines of code. Some of these systems, and in particular the Cedar compiler, exist in numerous versions.

The implementation and use of this old SML language uncovered a number of problems. The language has been redesigned and the evaluator is being rewritten to take advantage of the more solid foundations of the language described in this paper. The largest improvements have been in the uniform treatment of declarations and bindings as first-class values, the systematic derivation of SML

values from elements, and the use of β -reduction for evaluation.

During the next year we expect to use SML and the Modeller to control the development of the entire Cedar system, which is now about 500k lines of source code.

6. Conclusion

SML is used to describe a system assembly and module interconnection scheme in which polymorphism occurs naturally. SML consists of the applicative subset of the Cedar Kernel language, with values that correspond to types, declarations and bindings, as well as the interfaces and instances which characterize Cedar modules. SML is based on the λ -calculus; it uses Algol scope rules.

The most common value is a Cedar interface or instance. Each Cedar interface defines a single INTERFACE value in SML; each implementation has a function type that depends on the interfaces it uses and implements. The interconnections among modules are expressed by treating each module as a function which returns instances of the interfaces it implements, and passing each interface or instance as an argument to the modules that use it. The arguments also include any other information needed to run the module in the system, such as character strings that specify the compiler options.

A model logically includes the entire text of the system it describes. In fact, however, the text of a module is stored in a file which must be immutable, and is referenced from the model by a file name followed by a unique identifier for the particular version of the module. The filename is used as a hint since the unique-id identifies the module unambiguously. An object file is a source file that has been compiled with interface types filled in. A module is recompiled only when one of its interfaces changes.

A system model is thus a stable, unambiguous representation for a system. It is easily transferred among programmers and file systems. It has a readable text representation that can be edited by a user at any time. Finally, it is usable by other program utilities such as cross-reference programs, debuggers, and optimizers that analyze inter-module relationships.

Acknowledgements

System modelling began with ideas developed jointly with Charles Simonyi, and grew out of the discussions of a working group which included Bob Ayers, Phil Karlton, Tom Malloy, Ed Satterthwaite and John Wick. Many conversations with Rod Burstall clarified the notions of binding and declaration. Ed Satterthwaite has given us a lot of helpful comments and advice.

References

- [1] Avakian, A. *et al.*, The design of an integrated support software system. *Proc. SIGPLAN '82 Symp. Compiler Construction*, June 1982, 308-317.
- [2] Coopridge, L.W., *The Representation of Families of Software Systems*. PhD Thesis, CMU-CS-79-116, Computer Science Dept., CMU, April 1979.
- [3] Cristoforo, E. *et al.*, Source control + tools = stable systems. *Proc. 4th Computer Software and Applications Conf.*, Oct., 1980, 527-532.
- [4] Demers, A. and Donahue, J., Data types, parameters, and type checking. *Proc. 7th Symp. Principles of Programming Languages*, Las Vegas, 1980, 12-23.
- [5] DeRemer, F. and Kron, H., Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Software Eng.* SE-2, 2, June 1976, 80-86.
- [6] Deutsch, L.P. and Taft, E.A., *Requirements for an Experimental Programming Environment*. CSL-80-10, Xerox PARC, 1980.
- [7] Goldstein, I.P. and Bobrow, D.G., Descriptions for a programming environment. *Proc. 1st Ann. Conf. Natl. Assoc. Artificial Intelligence*, Stanford, Aug. 1980.
- [8] Habermann, A.N. *et al.*, *The Second Compendium of Gandalf Documentation*. Computer Science Dept., CMU, May 1982.
- [9] Harslem, E. and Nelson, L.E., A retrospective on the development of Star. *Proc. 6th Intl. Conf. Software Eng.*, Tokyo, Sept. 1982.
- [10] Horsley, T.R. and Lynch, W.C., Pilot: A software engineering case study. *Proc. 4th Intl. Conf. Software Eng.*, Munich, 1979, 94-99.
- [11] Lamson, B.W. and Schmidt, E., Organizing software in a distributed environment. In preparation.
- [12] Lauer, H.C. and Satterthwaite, E.H., The impact of Mesa on system design. *Proc. 4th Intl. Conf. Software Eng.*, Munich, 1979, 174-182.
- [13] Mitchell, J.G. *et al.*, *Mesa Language Manual*. CSL-79-3, Xerox PARC, April 1979.
- [14] Nelson, B.J., *Remote Procedure Call*. CSL-81-9, Xerox PARC, May 1981.
- [15] Schmidt, E., *Controlling Large Software Development in a Distributed Environment*. PhD Thesis, EECS Dept., Univ. of Calif. Berkeley, Dec. 82.
- [16] Tichy, W.F., Design, implementation, and evaluation of a revision control system. *Proc. 6th Intl. Conf. Software Eng.*, Tokyo, Sept. 1982.

Appendix: A real example

This model describes the BringOver program, which is a substantial component in the Cedar system. First, we present the model with its environment aggregated into separate models, and with defaults for all the parameters. Then we give a fully expanded version, to show the entire dependency structure.

There are seven implementation modules within this model (*CWFImpl*, *ComParseImpl*, *SubrImpl*, *STPSubrImpl*, *DFSubrImpl*, *DFParserImpl*, *BringOverImpl*). All the rest are interfaces.

First we define the two environment models. One is a big binding for the Pilot interfaces on which *BringOver* and many other parts of Cedar depend. The other is a declaration for the instances of these interfaces. This

declaration is rather repetitive, but it is needed to provide the proper names for defaulting the instance arguments of the *BringOver* models. §4.3 explains how to avoid this declaration by passing the entire interface binding, and a corresponding binding for the instances, as two big arguments to the client modules. Cedar currently does not permit this, however, and we do not show it here.

Pilot.model

```
[ Ascii ~ @Ascii.cedar*[];
  CIFS ~ @CIFS.cedar*[];
  ConvertUnsafe ~ @ConvertUnsafe.cedar*[];
  Date ~ @Date.cedar*[];
  DCSFileTypes ~ @DCSFileTypes.cedar*[];
  Directory ~ @Directory.cedar*[];
  Environment ~ @Environment.cedar*[];
  Exec ~ @Exec.cedar*[];
  File ~ @File.cedar*[];
  FileStream ~ @FileStream.cedar*[];
  Heap ~ @Heap.cedar*[];
  Inline ~ @Inline.cedar*[];
  KernelFile ~ @KernelFile.cedar*[];
  LongString ~ @LongString.cedar*[];
  NameAndPasswordOps ~ @NameAndPasswordOps.cedar*[];
  Process ~ @Process.cedar*[];
  Rope ~ @Rope.cedar*[];
  RopeInline ~ @RopeInline.cedar*[];
  Runtime ~ @Runtime.cedar*[];
  Segments ~ @Segments.cedar*[];
  Space ~ @Space.cedar*[];
  Storage ~ @Storage.cedar*[];
  STP ~ @STP.cedar*[];
  STPOps ~ @STPOps.cedar*[];
  Stream ~ @Stream.cedar*[];
  String ~ @String.cedar*[];
  System ~ @System.cedar*[];
  SystemInternal ~ @SystemInternal.cedar*[];
  Time ~ @Time.cedar*[];
  Transaction ~ @Transaction.cedar*[];
  TTY ~ @TTY.cedar*[];
  UserTerminal ~ @UserTerminal.cedar*[] ]
```

PilotInstancesDecl.model

```
LET @PilotInterfaces.model IN
[ CIFSImpl: CIFS,
  ConvertUnsafeImpl: ConvertUnsafe,
  -- 23 declarations are omitted for brevity--
  TTYImpl: TTY,
  UserTerminalImpl: UserTerminal ]
```

The models above are part of the working environment of a Cedar programmer; they are constructed once, as part of building the Pilot operating system.

Now we can write the model for *BringOver*. It picks up the two Pilot models above, and then gives a single binding of *BringOverProc* to a function which takes the instances as an argument, and returns two interfaces and an instance of each. The body of the function has

one LET to make all the Pilot interface and instance names directly accessible for defaulting;

a second LET to bind all the internal interfaces and instances of *BringOver*;

a binding to construct the two interfaces and two instances which are the result of applying *BringOverProc*.

BringOver.model

```
LET [Interfaces~@Pilot.model, InstancesDecl~@PilotInstancesDecl.model IN
[BringOverProc ~ λ [Instances: InstancesDecl]⇒
  [BringOver: INTERFACE, BringOverImpl: BringOver,
  BringOverCall: INTERFACE, BringOverCallImpl: BringOverCall] IN
  --Make the Pilot interface and instance names accessible
  LET Interfaces+ Instances IN
  LET [ -- These are the internal interfaces and instances
    CWF ~ @CWF.cedar*[];
    CWFImpl ~ @CWFImpl.cedar*[];
    ComParse ~ @ComParse.cedar*[];
    ComParseImpl ~ @ComParseImpl.cedar*[];
    Subr ~ @Subr.cedar*[];
    SubrImpl ~ @SubrImpl.cedar*[];
    STPSubr ~ @STPSubr.cedar*[];
    STPSubrImpl ~ @STPSubrImpl.cedar*[];
    DFSSubr ~ @DFSSubr.cedar*[];
    DFUser ~ @DFUser.cedar*[];
    DFSSubrImplA ~ @DFSSubrImpl.cedar*[];
    DFSSubrImplB ~ @DFParserImpl.cedar*[];
    DFSSubrImpl ~ DFSSubrImpl+ DFSSubrImplB ]
  IN [ -- These are the exported interfaces and instances
    BringOver~ @BringOver.cedar*[];
    BringOverCall ~ @BringOverCall.cedar*[];
    [BringOverImpl: BringOver, BringOverCallImpl: BringOverCall] ~
    @BringOverImpl.cedar*[] ] ]
```

To apply this model, we need instances for the Pilot interfaces. We can get them from the following model; its type is *@PilotInstancesDecl.model*.

PilotInstances.model

```
LET Interfaces@Pilot.model IN
[ CIFSImpl: CIFS ~ CIFSImpl.cedar*[],
  ConvertUnsafeImpl: ConvertUnsafe ~ ConvertUnsafeImpl.cedar*[],
  DateImpl: Date ~ DateImpl.cedar*[],
  -- 23 bindings are omitted for brevity--
  TTYImpl: TTY ~ TTYImpl.cedar*[],
  UserTerminalImpl: UserTerminal ~ UserTerminalImpl.cedar*[] ]
```

Using this binding, we can compute the exported interfaces and instances of *BringOver*:

```
[BringOver, BringOverImpl, BringOverCall, BringOverCallImpl] ~
  BringOverProc[@PilotInstances.model]
```

Making the arguments explicit

In the previous version, we defaulted all the arguments, since the modeller can supply for each parameter an actual with the same name. We also omitted the types in bindings. Here is a version with everything written out explicitly.

```

LET [Interfaces~@Pilot.model, InstancesDecl~@PilotInstancesDecl.model IN
[BringOverProc ~ λ [Instances: InstancesDecl]⇒
[BringOver: INTERFACE, BringOverImpl: BringOver,
BringOverCall: INTERFACE, BringOverCallImpl: BringOverCall] IN
LET Interfaces+Instances IN
LET [ -- These are the internal interfaces and instances
CWF: INTERFACE ~ @CWF.cedar*];
CWFImpl: CWF ~ @CWFImpl.cedar*[[HeapImpl, InlineImpl,
LongStringImpl, TimeImpl],
ComParse: INTERFACE ~ @ComParse.cedar, ComParseImpl:
ComParse ~ @ComParseImpl.cedar{Ascii, ComParse, Exec, Storage,
String, TTY, ExecImpl, StorageImpl, StringImpl, TTYImpl},
Subr: INTERFACE ~ @Subr.cedar{File, Space, Stream, TTY},
SubrImpl: Subr ~ @SubrImpl.cedar{Ascii, CWF, DCSFileTypes,
Directory, Environment, Exec, File, FileStream, Heap, Inline,
LongString, NameAndPasswordOps, Runtime, Segments, Space,
Stream, Subr, System, TTY, CWFImpl, DirectoryImpl, ExecImpl,
FileImpl, FileStreamImpl, HeapImpl, InlineImpl, LongStringImpl,
NameAndPasswordOpsImpl, RuntimeImpl, SegmentsImpl, SpaceImpl,
StreamImpl, TTYImpl},
STPSubr: INTERFACE ~ @STPSubr.cedar{File, STP, Stream,
System, TTY},
STPSubrImpl: STPSubr ~ @STPSubrImpl.cedar{CIFS,
ConvertUnsafe, CWF, Date, DCSFileTypes, Directory, Environment,
Exec, File, FileStream, Inline, LongString, NameAndPasswordOps,
Process, Space, Storage, STP, STPOps, STPSubr, STPSubrExtras,
Stream, String, Subr, TTY, UserTerminal, CIFSImpl,
ConvertUnsafeImpl, CWFImpl, DateImpl, DirectoryImpl, ExecImpl,
FileImpl, FileStreamImpl, InlineImpl, LongStringImpl,
NameAndPasswordOpsImpl, ProcessImpl, SpaceImpl, STPImpl,
STPOpsImpl, StorageImpl, StreamImpl, StringImpl, SubrImpl,
UserTerminalImpl},
DFSubr: INTERFACE ~ @DFSubr.cedar{File, Stream, TTY},
DFUser: INTERFACE ~ @DFUser.cedar{DFSubr, TTY},
DFSubrImplA: DFSubr ~ @DFSubrImpl.cedar{CWF, DFSubr,
DFUser, Directory, Environment, Exec, Heap, Inline, LongString,
Space, STPSubr, Stream, String, Subr, SystemInternal, TTY,
CWFImpl, DFSubrImpl, DirectoryImpl, ExecImpl, HeapImpl,
InlineImpl, LongStringImpl, SpaceImpl, STPSubrImpl, StreamImpl,
StringImpl, SubrImpl, TTYImpl},
DFSubrImplB: DFSubr ~ @DFParserImpl.cedar{CWF, Date,
DFSubr, Exec, LongString, Stream, String, Subr, Time, CWFImpl,
DateImpl, DFSubrImpl, ExecImpl, LongStringImpl, StreamImpl,
StringImpl, SubrImpl, TimeImpl},
DFSubrImpl: DFSubr ~ (DFSubrImplA) + (DFSubrImplB) ]
IN [ -- These are the exported interfaces and instances
BringOver: INTERFACE ~ @BringOver.cedar,
BringOverCall: INTERFACE ~ @BringOverCall.cedar{Rope, TTY},
[BringOverImpl: BringOver, BringOverCallImpl: BringOverCall] ~
@BringOverImpl.cedar{BringOverCall, BringOverInterface, CIFS,
ComParse, CWF, Date, DFSubr, Directory, Exec, File, FileStream,
KernelFile, LongString, Rope, RopeInline, Runtime, Space, Storage,
STP, STPSubr, STPSubrExtras, Stream, String, Subr, Time, TTY,
CIFSImpl, ComParseImpl, CWFImpl, DateImpl, DFSubrImpl,
DirectoryImpl, ExecImpl, FileStreamImpl, KernelFileImpl,
LongStringImpl, RuntimeImpl, RopeImpl, RopeInlineImpl, SpaceImpl,
StorageImpl, STPImpl, STPSubrImpl, STPSubrExtrasImpl,
StreamImpl, StringImpl, SubrImpl, TimeImpl, TTYImpl]
]
]

```