Portable library descriptions for Standard ML

Matthias Blume*
Lucent Technologies, Bell Labs
600 Mountain Avenue, 2C-481
Murray Hill, NJ 07974, USA

November 26, 2001

1 Introduction

This text is a proposal for the design of a portable library description format that can be used across different implementations of Standard ML. My hope is that I made it general and flexible enough to meet most people's needs while keeping it conceptually simple so that all implementations should have a good chance of being able to generate and process these library descriptions easily.

> This proposal is **not** meant to be the library mechanism of choice for every implementation of Standard ML.[1] Instead, it is expected that each implementation continues to develop its own native design of such a mechanism. However, each implementation-specific library mechanisms should make provisions for "importing" and "exporting" portable libraries.

The ideas behind my design are very simple. The starting point is the general definition-use graph that connects library elements that contain toplevel definitions with library elements that contain the corresponding uses. From this I derive *library graphs*—a more elaborate version of dependency graphs that consist of *typed* nodes. Each node type represents one of the conceptual operations that an implementation must carry out during library elaboration.

Library graphs have a number of special properties. Most of these properties arise directly from the problem. Some other properties are imposed in an ad-hoc fashion to make life easier for implementers. This text tries to list all the relevant properties.[2]

After discussing graph properties, I present both abstract and concrete syntax for library graph descriptions. The abstract syntax is shown in the form of an SML datatype.[3]

*e-mail: *blume@research.bell-labs.com*

[1]The portable format was designed to be processed by programs, not to be read or written by human programmers.

[2]The reader should not be too alarmed by the length of this listing. None of the properties should be "surprising."

[3]I tested the ideas here using a sample implementation based on SML/NJ's compilation manager CM. This implementation uses precisely the type definitions shown here.

Finally, the concrete syntax is more or less a straightforward rendering of the abstract syntax—but not without a twist: Library descriptions can be parsed by simple special-purpose parsers, but they can also be *interpreted as SML source code*. This gives rise to some clever implementation techniques.

2 The starting point: def-use graphs

A *Standard ML library* is a collection of *source files* ("library elements"), each containing a sequence of toplevel declarations, a collection of *imported libraries*, and an *export specification*.

The most important aspect of a library's structure is its *def-use graph*. The def-use relation describes how the free occurence of an identifier in a source file or within the export specification matches up with its corresponding toplevel definitions in another source file or within one of the library's imports.

For our purposes, def-use graphs will always be acyclic. This means that we can describe them formally as directed labelled multigraphs. The labels on each node designate it as being either a named imported library, a named source file, or the export node. Each edge is labelled with an SML symbol and connects the definition with a use of that symbol. For many purposes, a concrete representation of a library's multigraph together with its source files and its imported libraries is all we need.

One can view the nodes in library graphs as representing concrete operations that will be carried out by implementations that process the library description. For example, a source file node could stand for the operation of compiling that source file. The set of incoming edges characterizes the static environment relative to which the code must be elaborated, the set of outgoing edges describes a static environment that carries toplevel bindings created by definitions in the node's source itself.

This view is useful but has a few shortcomings. Most importantly, graph nodes must be a bit too "multi-talented"as they have to do everything by themselves. There is no separation of concerns such as

- formation of environments,
- compilation,
- distribution of bindings,
- imports from other libraries, and
- exports from this library.

We would also like to be able to extract information such as the set of toplevel names defined by a source file directly from the graph without having to look inside SML source files. A pure def-use graph without additional annotations is not enough for this.

2.1 Example

Consider four source files `a.sml`, `b.sml`, `c.sml`, and `d.sml` with the following contents:

```
(* a.sml *)
structure X = struct
   val baz = ...
   ... Int.+ ...
   ... IntRedBlackMap.find ...
end
structure Y = struct
   val foo = ...
   ...
end
```

```
(* b.sml *)
structure X = struct
   val bar = ...
   ... Y.foo ...
end
```

```
(* c.sml *)
structure Z = struct
   ... X.bar + Y.foo ...
   ... Int.- ...
end
```

```
(* d.sml *)
structure W = struct
   ... X.baz ...
end
```

Let us assume that structure `Int` is being imported from a library called `basis` while structure `IntRedBlackMap` comes from another library called `smlnj-lib`. The def-use graph for a library consisting of the above four source files and exporting bindings for structures `Z` and `W` is shown in figure 1.

3 Library graphs

To address the shortcomings of simple def-use graphs, we will use a slightly more elaborate representation of the def-use graph with the following properties:

1. The graph is a directed graph.
2. The graph is acyclic.
3. The graph is not a multigraph.
4. The nodes of the graph are *typed*, and have labels appropriate to their types.
5. Each node type is dedicated to one simple "task."
6. Each node has an associated symbol set. It is useful to think of this set as the domain of a "static environment." (Our library descriptions do not actually talk about static environments but only characterize their domains.)

The idea is that the graph represents the flow of static information (*static environments*) within the library. *Compile nodes* and *import nodes* introduce new environments, *merge nodes* combine existing environments, *filter nodes* are used for thinning, and the graph's *export node* acts as a unique "sink" into which results flow. Import nodes act as "source" nodes.

Merge nodes are the only nodes that have more than one incoming edge. We sidestep the need for an ordering on such incoming edges by requiring the domains of the corresponding environments to be disjoint.

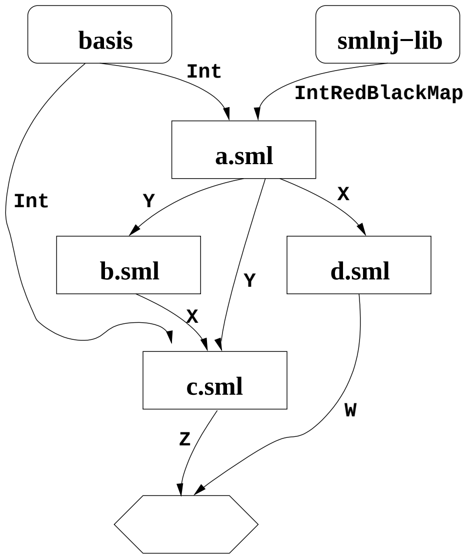

Figure 1: The def-use graph for a simple library

3.1 Node types

Each node of a library graph belongs to precisely one of the following five *node types*:

Import: This node type represents the operation of importing bindings from imported libraries.

eXport: The unique export node represents the operation of exporting bindings to other libraries.

Compile: Nodes of this type represent the operation of compiling an SML source in a given environment, resulting in a new environment that carries the toplevel definitions of the source.

Filter: This is the type of nodes that reduce the domain of a given environment by “thinning” or “filtering” (restricting the domain of the environment to a given symbol set).

Merge: Nodes of this type combine several environments with disjoint domains into a single environment.

I, C, and X have to do with production and consumption of environments (represented by symbol sets), while F and M nodes are “name space administration” nodes, whose purpose is to unambiguously determine the flow of names between the other nodes.

3.2 Formal definition of library graphs

Here is a formal definition of library graphs $G = (V, E)$:

$$\begin{aligned} V &\subset \text{nodes} \\ E &\subset V \times V \end{aligned}$$

For each node v we define sets pred(v) and succ(v) as:

$$\begin{aligned} \text{pred}(v) &= \{u \in V \mid (u, v) \in E\} \\ \text{succ}(v) &= \{u \in V \mid (v, u) \in E\} \end{aligned}$$

The in- and out-degrees $\text{deg}_{\text{in}}(v)$ and $\text{deg}_{\text{out}}(v)$ of a node v are defined as:

$$\begin{aligned} \text{deg}_{\text{in}}(v) &= \mid \text{pred}(v) \mid \\ \text{deg}_{\text{out}}(v) &= \mid \text{succ}(v) \mid \end{aligned}$$

3.2.1 Node types

The typing of nodes partitions V into five disjoint sets I, X, C, F, and M.

$$\begin{aligned} V &= \text{I} \cup \text{X} \cup \text{C} \cup \text{F} \cup \text{M} \\ X, Y \in \{\text{I}, \text{X}, \text{C}, \text{F}, \text{M}\} \wedge X \neq Y &\Rightarrow X \cap Y = \emptyset \end{aligned}$$

3.2.2 Node labels

Every import node $v \in \text{I}$ is labelled with a name $\mathcal{L}_i(v) \in L$ of a library and every compile nodes $w \in \text{C}$ bears the name $\mathcal{L}_c(v) \in F$ of an SML source file.

$$\begin{aligned} v \in \text{I} &\Rightarrow \mathcal{L}_i(v) \in L \\ v \in \text{C} &\Rightarrow \mathcal{L}_c(v) \in F \end{aligned}$$

3.2.3 Symbol sets

Every node has an associated set $s \subset S$ of SML symbols:

$$\mathcal{S}(v) \in 2^S$$

3.2.4 Paths

We define $u \rightsquigarrow v$ as the transitive closure of the edge relation E:

$$\begin{aligned} (u,v) \in E &\quad\Rightarrow\quad u \rightsquigarrow v \\ (u,v) \in E \wedge v \rightsquigarrow W &\quad\Rightarrow\quad u \rightsquigarrow w \end{aligned}$$

If $u \rightsquigarrow v$ then we also say v *depends on* u.

3.3 Well-formed library graphs

A well-formed library graph $G = (V, E)$ has the following properties:

3.3.1 General properties

ACYCLIC The $\rightsquigarrow$ relation is irreflexive:

$$\forall v \in V . v \not\rightsquigarrow v$$

ONE_X The graph has precisely one export node:

$$\mid X \mid = 1$$

CONNECTED The unique export node depends on all other nodes in the graph:

$$v \in \mathrm{X} \Rightarrow \forall u \in V . (u \neq v \rightarrow u \rightsquigarrow v)$$

3.3.2 Node-specific properties

I_NOIN Import nodes have no incoming edges:

$$v \in \mathrm{I} \Rightarrow \mathrm{deg}_{\mathrm{in}}(v) = 0$$

X_NOOUT The unique export node has no outgoing edges:

$$v \in \mathrm{X} \Rightarrow \mathrm{deg}_{\mathrm{out}}(v) = 0$$

(This follows from ACYCLIC and CONNECTED.)

X_OUT The export node's symbol set is the same as that of its predecessor:

$$v \in X \wedge u \in \mathrm{pred}(v) \Rightarrow \mathcal{S}(v) = \mathcal{S}(u)$$

CFX_ONEIN Nodes of type C, F, or X have precisely one incoming edge:

$$v \in \mathrm{C} \cup \mathrm{F} \cup \mathrm{X} \Rightarrow \mathrm{deg}_{\mathrm{in}}(v) = 1$$

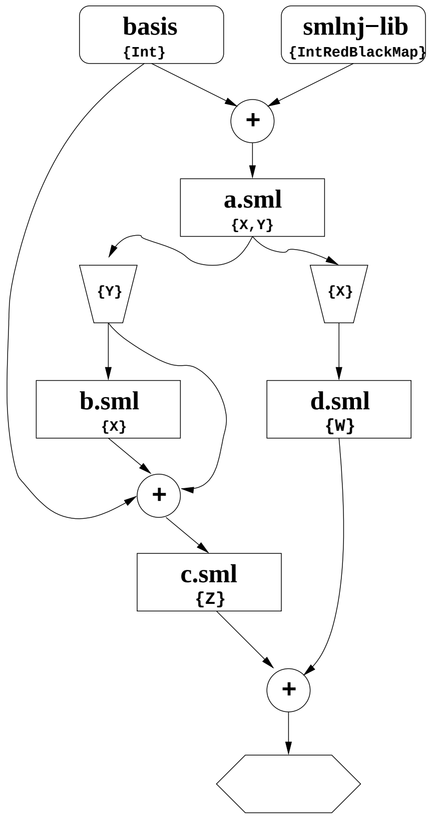

Figure 2: The library graph for a simple library

F_OUT The symbol set of a filter node is a subset of the symbol set of its predecessor:

$$v \in \mathrm{F} \wedge u \in \mathrm{pred}(v) \Rightarrow \mathcal{S}(v) \subset \mathcal{S}(u)$$

M_DISJOINT The symbol sets of a merge node's predecessors are disjont:

$$v \in \mathrm{M} \wedge \{u_1, u_2\} \subset \mathrm{pred}(v) \wedge u_1 \neq u_2 \Rightarrow \mathcal{S}(u_1) \cap \mathcal{S}(u_2) = \emptyset$$

M_UNION The symbol set of a merge node is the union of the symbol sets of its predecessors:

$$v \in \mathrm{M} \Rightarrow \mathcal{S}(v) = \bigcup_{u \in \mathrm{pred}(v)} \mathcal{S}(u)$$

Note: The in-degree of merge nodes $v \in \mathrm{M}$ is not restricted.

3.4 Example

The library graph for our example (see section 2.1) would look as shown in the figure 2. Square boxes are nodes of type C, square boxes with rounded corners are nodes of type I, circles with a plus sign are nodes of type M, trapezoids are nodes of type F, and the hexagonal box is the export node (type X). The symbol sets for M- and X-nodes can be inferred (X_OUT, M_UNION) and, thus, do not need to be given explicitly.

4 Abstract syntax

We write down a library graph as a list of named node specifications. Each node name can also be seen as naming all of the node's outgoing edges. Individual edges do not need to be mentioned at all: each node specification simply lists the node's type, its label, and the name(s) of the source node(s) of its incoming edge(s). The only node that does not need to be named is the export node; all one needs to know about it is its incoming edge.

Nodes are named by *varnames*—simple strings drawn from a countable supply of fresh names.[4] A varname identifies both the node itself and all of its outgoing edges. In some sense the constructors of the `rhs` type represent node types as discussed in section 3. However, this is not the only purpose of `rhs`: We have "flattened" the representation and included the possibility of describing symbols and symbol sets. As a result, our abstract syntax of library graphs is a list of varname *definitions*, each such definition either specifying a graph node, a symbol, or a symbol set.

The following SML code defines the abstract syntax:

```
structure PortableGraph = struct
    type varname = string

    datatype namespace = SGN | STR | FCT
```

[4]We call these names *varnames* because they will be used as actual SML variable names later.

```
    datatype rhs =
        SYM of namespace * string
      | SYMS of varname list
      | IMPORT of { lib: varname, syms: varname }
      | COMPILE of { src: string * bool, env: varname, syms: varname }
      | FILTER of { env: varname, syms: varname }
      | MERGE of varname list

    datatype def = DEF of { lhs: varname, rhs: rhs }

    datatype graph =
        GRAPH of { imports: varname list,
                   defs: def list,
                   export: varname }
end
```

Notice that there is no constructor representing the X node type since the export node is implicitly described by the `export` field of the `GRAPH` constructor.

The `lib` field of `IMPORT` for some import node v corresponds to $\mathcal{L}_i(v)$ and `src` of `COMPILE` corresponds to the label $\mathcal{L}_c(u)$ for a compile node u. For a discussion of the boolean component of `src` which is used to select either *portable* or *native* source file naming, see section 7.

4.1 Well-formed abstract syntax

For a graph description to be well-formed, all of the following must be true:

VARNAME Varnames must be non-empty strings that start with a lowercase letter and contain only lowercase letters and decimal digits. Moreover, varnames must not coincide with any Standard ML keywords or with any of the following: `sgn`, `str`, `fct`, `syms`, `import`, `compile`, `ncompile`, `merge`, `filter`, `export`, `c`. (This is a simple way of guaranteeing that each varname can be used as an ML variable name and does not clash with other names.)

VARNAME_ONCE Each varname can appear in at most one `lhs` or once in `imports` (but not both).[5]

TOPOLOGICAL Any occurrence of a varname in an `rhs` other than `lib` in `IMPORT` must be the `lhs` of a definition that appears closer to the front of `defs`.[6]

EXPORTLAST The `lhs` of the last element of `defs` must coincide with `export`.

SRC_ONCE There can be no more than one `COMPILE` with `src` fields that refer to the same source file (see section 7).

CSE "Common subexpressions" must be eliminated, i.e., the right-hand sides (`rhs`) in `defs` must be pairwise distinct.

[5]This rule is an easy way of guaranteeing freedom from name clashes when a library description is rendered in concrete syntax and interpreted as SML code (see section 5). The extra freedom that one would get from using a weaker and still sufficient restriction does not seem to be worth the extra complication.

[6]We require topological order to guarantee a 1-1 match of abstract and concrete syntax (see section 5) without having to explicitly "sort" the definitions.

LIB_TYPE Varnames used for `lib` in `IMPORT` must appear in `imports`.

SYM_TYPE The varnames in `SYMS` must be defined by `SYM`.

SYMS_TYPE The varname used in the `syms` field of `IMPORT`, `COMPILE`, and `FILTER` must be defined using `SYMS`.

ENV_TYPE The varnames used for `export`, by `MERGE`, and by the `env`-fields of `COMPILE` and `FILTER` must have been defined by either `IMPORT` or `COMPILE` or `FILTER` or `MERGE`.

4.2 Example

The abstract syntax for the library graph shown in sections 2.1 and 3.4 could be the following:

```
let open PortableGraph in
GRAPH {
  imports = ["basis", "smlnjlib"],
  defs =
    [DEF { lhs = "v1", rhs = SYM (STR, "Int") },
     DEF { lhs = "v2", rhs = SYM (STR, "IntRedBlackMap") },
     DEF { lhs = "v3", rhs = SYM (STR, "X") },
     DEF { lhs = "v4", rhs = SYM (STR, "Y") },
     DEF { lhs = "v5", rhs = SYM (STR, "Z") },
     DEF { lhs = "v6", rhs = SYM (STR, "W") },
     DEF { lhs = "v7", rhs = SYMS ["v1"] },
     DEF { lhs = "v8", rhs = SYMS ["v2"] },
     DEF { lhs = "v9", rhs = SYMS ["v3"] },
     DEF { lhs = "v10", rhs = SYMS ["v4"] },
     DEF { lhs = "v11", rhs = SYMS ["v5"] },
     DEF { lhs = "v12", rhs = SYMS ["v6"] },
     DEF { lhs = "v13", rhs = SYMS ["v3", "v4"] },
     DEF { lhs = "v14", rhs = IMPORT { lib = "basis", syms = "v7" } },
     DEF { lhs = "v15", rhs = IMPORT { lib = "smlnjlib", syms = "v8" } },
     DEF { lhs = "v16", rhs = MERGE ["v14", "v15"] },
     DEF { lhs = "v17", rhs =
         COMPILE { src = ("a.sml", false), env = "v16", syms = "v13" } },
     DEF { lhs = "v18", rhs = FILTER { env = "v17", syms = "v9" } },
     DEF { lhs = "v19", rhs = FILTER { env = "v17", syms = "v10" } },
     DEF { lhs = "v20", rhs =
         COMPILE { src = ("b.sml", false), env = "v19", syms = "v9" } },
     DEF { lhs = "v21", rhs =
         COMPILE { src = ("d.sml", false), env = "v18", syms = "v12" } },
     DEF { lhs = "v22", rhs = MERGE ["v14", "v20", "v19"] },
     DEF { lhs = "v23", rhs =
         COMPILE { src = ("c.sml", false), env = "v22", syms = "v11" } },
     DEF { lhs = "v24", rhs = MERGE ["v22", "v23"] }],
  export = "v24" }
end
```

5 Concrete syntax

SML code such as shown in section 4.2 could be parsed, compiled, and executed. The result would be a data structure containing the abstract syntax of a library graph. Thus, the code itself could serve as a form of concrete syntax. However, the abstract syntax was chosen in such a way that it could be formatted as SML code more directly: each *varname* becomes an actual SML variable instead of a string and each node type becomes a function application. The library as a whole then describes a function, parameterized by "context," that maps "imported libraries" to its "exports."

Nevertheless, the interpretation of library descriptions as SML programs is not the only one that should be supported. Thus, the concrete syntax was chosen so that it is easy to be parsed by very simple-minded parsers, too.

When a library description is read as a Standard ML *topdec* and assuming that there are appropriately-typed definitions for `sym`, `syms`, `import`, `compile`, `ncompile`, `filter`, `merge`, and `export` in scope, it should type-check successfully according to Standard ML typing rules.

The format of a graph description *descr* is line-oriented, so in the following EBNF we use the token ↩ to refer to linebreaks.

descr	→	*header* ↩
		impspec ↩
		{ *def* ↩ }
		footer ↩
header	→	**val** `thelibrary` `=` **fn** `c` `=>` `(`
impspec	→	**fn** `[` [*imp* { `,` *imp* }] `]` `=>` **let open** `PGOps`
footer	→	**in** ↩
		`export` `c` *export* ↩
		end ↩
		`_` `=>` **raise** `Fail` *errormsg* `)`
export	→	*varname*
def	→	**val** `(` `c` `,` *sym* `)` `=` *namespace* `c` *symname*
	|	**val** `(` `c` `,` *symset* `)` `=` `syms` `c` `[` [*sym* { `,` *sym* }] `]`
	|	**val** `(` `c` `,` *env* `)` `=` `import` `c` *lib symset*
	|	**val** `(` `c` `,` *env* `)` `=` `compile` `c` *srcname env symset*
	|	**val** `(` `c` `,` *env* `)` `=` `ncompile` `c` *srcname env symset*
	|	**val** `(` `c` `,` *env* `)` `=` `filter` `c` *env symset*
	|	**val** `(` `c` `,` *env* `)` `=` `merge` `c` `[` [*env* { `,` *env* }] `]`
namespace	→	`sgn` | `str` | `fct`
symname	→	*string*
srcname	→	*string*
lib	→	*varname*
symset	→	*varname*
env	→	*varname*
sym	→	*varname*
imp	→	*varname*

Notice that both `compile` and `ncompile` of the concrete syntax correspond to the `COMPILE` constructor of the abstract syntax. The choice between `compile` and `ncompile` encodes the value of the boolean component of the `src` field (see section 7).

The syntactic category *varname* describes the subset of Standard ML identifiers that start with some lower-case letter and contain nothing but decimal digits and lowercase letters. Moreover, to avoid clashes with other names that occur in library descriptions, the following names cannot be used as a *varname*: `sgn`, `str`, `fct`, `syms`, `import`, `compile`, `ncompile`, `filter`, `merge`, `export`, and `c`.

When a library description is rendered in concrete syntax, then each *varname* assumes the rôle of an SML variable name. However, these variable names are not to be confused with those SML variable names that appear in the actual source code (contained in library elements). Instead, occurences of *varname* are interpreted at the "meta"-level where they refer to nodes of the library graph.

The category *string* stands for Standard ML string literals. When a string literal is used for *srcname* (see `compile` and `ncompile`), then it will be interpreted as the name of a Standard ML source file (see section 7).

When *string* is used as *symname*, then the contents of the string spells out the name of a Standard ML identifier. These identifiers are interpreted at the level of the source code contained in individual library elements represented by nodes of type C.

5.1 Well-formed concrete syntax

A concrete library description is well-formed iff all of the following is true:

- No *varname* occurs more than once between **val** and `=` or within the *impspec*.
- The *varname* that occurs as *export* is the same as the one that occurs last between **val** and `=`.
- The entire description must be a valid Standard ML definition if it is elaborated in the scope of a structure `PGOps` with the following signature:

```
structure PGOps : sig
   type context
   type lib
   type env
   type sym
   type symset
   type export

   val sgn : context -> string -> context * sym
   val str : context -> string -> context * sym
   val fct : context -> string -> context * sym
   val syms : context -> sym list -> context * symset
   val import : context -> lib -> symset -> context * env
   val compile : context -> string -> env -> symset -> context * env
   val ncompile : context -> string -> env -> symset -> context * env
   val filter : context -> env -> symset -> context * env
   val merge : context -> env list -> context * env
   val export : context -> env -> export
end
```

  Moreover, the type of `thelibrary` shall be inferred as

```
        val thelibrary : context -> lib list -> export
```

The main purpose of the context argument `c` is to provide information on how to interpret relative pathnames that are given to `compile` and `ncompile`. However, by threading `c` through all operations, we gain a lot more expressive flexibility. Appendices C and D show examples on how to exploit this.

We deliberately leave types such as `env`, `context`, or `export` abstract here although it may be a useful mental model to think of, for example, `env` as the type of the respective implementation's static environments.

5.2 Example

The graph description for our example (sections 2.1 and 3.4 and 4.2) in concrete syntax looks like this:

```
val thelibrary = fn c => (
fn [basis, smlnjlib] => let open PGOps
       val (c, v1) = str c "Int"
       val (c, v2) = str c "IntRedBlackMap"
       val (c, v3) = str c "X"
       val (c, v4) = str c "Y"
       val (c, v5) = str c "Z"
       val (c, v6) = str c "W"
       val (c, v7) = syms c [v1]
       val (c, v8) = syms c [v2]
       val (c, v9) = syms c [v3]
       val (c, v10) = syms c [v4]
       val (c, v11) = syms c [v5]
       val (c, v12) = syms c [v6]
       val (c, v13) = syms c [v3, v4]
       val (c, v14) = import c basis v7
       val (c, v15) = import c smlnjlib v8
       val (c, v16) = merge c [v14, v15]
       val (c, v17) = compile c "a.sml" v16 v13
       val (c, v18) = filter c v17 v9
       val (c, v19) = filter c v17 v10
       val (c, v20) = compile c "b.sml" v19 v9
       val (c, v21) = compile c "d.sml" v18 v12
       val (c, v22) = merge c [v14, v20, v19]
       val (c, v23) = compile c "c.sml" v22 v11
       val (c, v24) = merge c [v22, v23]
   in
       export c v24
   end
 | _ => raise Fail "wrong number of input libraries")
```

6 Imported libraries

The proposal treats names of imported libraries as free variables and then uses λ-abstraction to formally close over them. No claims are made here about how to bind library variables to actual external libraries. (Of course, each implementation must provie a mechanims for this.)

Within portable descriptions, an external library is simply referred to by a *varname* (i.e., an SML variable under the description-as-SML-code interpretation). As an example of how library binding *could* be implemented, here is the outline of a realization based on the description-as-SML-code interpretation:

Suppose we already have primitive operations `compile`, `ncompile`, `filter`, and `merge` at our disposal. We then define:

```
type export = env
fun export c e = e
type lib = env
val import = filter
```

We could now store libraries in toplevel-bound variables and then use the following pattern:

```
val lib1 = ...          (* library defined earlier *)
val lib2 = ...          (* library defined earlier *)
val lib3 = ...          (* library defined earlier *)
```

```
val thelibrary = fn c => (
   fn [ l1, l2, l3 ] => let open PGOps
      ... (* library description here *)
      val (e721, c) = ...
   in
      export c e721
   end
 | _ => raise Fail "import libraries")
```

```
val context = ... (* context for thelibrary *)
val lib4 = thelibrary context [lib1, lib2, lib3]
```

If library descriptions are stored in separate files, then those lines that are boxed would have to be provided by `use`, e.g.:

```
 use "library-description.sml";
```

7 Source names

Both `compile` and `ncompile` definitions represent C nodes of the graph. The only difference is in how they interpret the source name string.

A portable graph description should use `compile` which accepts a *relative Unix-style* pathname. This means that forward slashes `/` separate arcs, a single dot `.` is the *current arc*, and two dots `..` denote the *parent arc*. Relative pathnames are interpreted in some implementation-specific *context* (e.g., relative to the directory that contains the library description file).

Graph descriptions can refer to source files in the host operating system's pathname syntax by using `ncompile` instead of `compile`. Of course, the use of `ncompile` renders a library description non-portable.

8 The Standard ML Basis library and “pervasive” bindings

All symbols used in symbol sets are names of modules: structures, signatures, or functors. This means that libraries do not export non-modular bindings.

To be able to treat the Standard ML Basis Library as a library in the sense of this proposal, we split it into two parts:

1. The first part is a very small *pervasive environment* that contains just those value and type bindings that are not within structures, signatures, or functors. This environment is implicitly passed to every compilation unit in every library and does not appear in library descriptions.
2. The other (and far larger) part consists of all other (module-) bindings of the Basis. This part could itself be described using the facilities proposed here. Other libraries refer to it just like they refer to any other library via `import`.

9 Order of processing

The only constraints on the order in which library elements are processed are given by the def-use relationship between them. In particular, if processing involves “linking” (which means executing top-level code), then the only requirement imposed by this proposal is that if $u \leadsto v$, then u must be linked before v. Conversely, it is a mistake to expect an effect in u to occur before some other effect in v unless $u \leadsto v$.

In terms of a library’s static semantics (i.e., compilation), it does not matter which order one chooses in processing individual library elements—as long as this order is consistent with the $\leadsto$ relation of the library graph. In particular, there are no issues of order of environment composition, since environments can only be combined at M nodes. These require disjoint input environments, which will in general have to be enforced by F nodes. So any post-order traversal of the graph will do.

Notice that library descriptions must necessarily choose one of the possible post-order traversals to lay out their list of definitions. Thus, an implementation could take advantage of the description-as-SML-code interpretation and SML’s order of evaluation by implementing `compile` as a function that really compiles and links the given source file. However, it would also be consistent if `compile` would act “lazily.”

10 (Non-)Extensibility

Individual implementations should refrain from generating “extended” library descriptions since they would not be “portable.” For example, a compiler that supports more namespaces in its own native separate compilation system (e.g., **funsig** in SML/NJ) should reject attempts of producing portable descriptions for those libraries that would require carrying the extensions into the description.

11 Acknowledgments

I thank Lal George, Dave MacQueen, and John Reppy for valuable feedback.

A Conversion from abstract syntax to concrete syntax

Source code given (or pointed-to) in this and subsequent sections is meant to

- demonstrate the power and flexibility of the library description format
- clarify any issues that had not been made clear so far
- serve as a possible starting point for individual implementations

We begin with Standard ML code defining a function `FormatPortable.output` that writes concrete syntax for a given `PortableGraph.graph` to a text output stream. The definition of such a function is shown in

http://cm.bell-labs.com/cm/cs/who/blume/pgraph/format.sml

.

B Conversion from concrete syntax to abstract syntax I: an explicit parser

The code shown in

http://cm.bell-labs.com/cm/cs/who/blume/pgraph/scan.sml

defines an explicit parser for concrete graph descriptions. It implements a function `ScanPortable.input` which will read a concrete description from a text input stream and produce the corresponding PortableGraph.graph. (The code does error-checking poorly and is more forgiving than the actual spec. For example, it does not enforce or rely on line-orientation within the body of the function.)

C A "generic" set of operations

The following code implements a generic set of operations which—if they are in scope—will let any Standard ML compiler successfully typecheck graph descriptions. Moreover, the resulting function value should still be extremely flexible since it does nothing more than lifting the individual operations out of the body of the function and into the context itself.

```
signature PG_OPS = sig

    type ('lib, 'env, 'sym, 'syms, 'export, 'misc) context

    val sgn : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
              -> string
              -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'sym
    val str : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
              -> string
              -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'sym
    val fct : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
              -> string
              -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'sym
    val import : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
                 -> 'lib -> 'syms
                 -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'env
    val compile : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
```

```
                 -> string -> 'env -> 'syms
                 -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'env
    val ncompile : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
                  -> string -> 'env -> 'syms
                  -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'env
    val merge : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
               -> 'env list
               -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'env
    val filter : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
                -> 'env -> 'syms
                -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'env
    val syms : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
              -> 'sym list
              -> ('lib, 'env, 'sym, 'syms, 'export, 'misc) context * 'syms
    val export : ('lib, 'env, 'sym, 'syms, 'export, 'misc) context
                -> 'env
                -> 'export
end

structure PGOps : PG_OPS = struct

    type ('lib, 'env, 'sym, 'syms, 'export, 'misc) context =
         { Ops : { Sgn: 'misc -> string -> 'misc * 'sym,
                   Str: 'misc -> string -> 'misc * 'sym,
                   Fct: 'misc -> string -> 'misc * 'sym,
                   Imp: 'misc -> 'lib * 'syms -> 'misc * 'env,
                   Com: 'misc -> string * 'env * 'syms * bool -> 'misc * 'env,
                   Mer: 'misc -> 'env list -> 'misc * 'env,
                   Fil: 'misc -> 'env * 'syms -> 'misc * 'env,
                   Syms: 'misc -> 'sym list -> 'misc * 'syms,
                   Exp: 'misc -> 'env -> 'export },
           Misc: 'misc }

    local
        fun generic { Ops = Ops as { Sgn, Str, Fct,
                                     Imp, Com, Mer, Fil, Syms, Exp },
                      Misc }
                    sel args =
            let val (Misc', res) = sel Ops Misc args
            in ({ Ops = Ops, Misc = Misc' }, res)
            end
    in
    fun sgn c s = generic c #Sgn s
    fun str c s = generic c #Str s
    fun fct c s = generic c #Fct s
    fun import c l ss = generic c #Imp (l, ss)
    fun compile c s e ss = generic c #Com (s, e, ss, false)
    fun ncompile c s e ss = generic c #Com (s, e, ss, true)
    fun merge c el = generic c #Mer el
    fun filter c e ss = generic c #Fil (e, ss)
    fun syms c sl = generic c #Syms sl
    fun export { Ops = { Sgn, Str, Fct, Imp, Com, Mer, Fil, Syms, Exp },
                 Misc } e = Exp Misc e
    end
end
```

D Conversion from concrete syntax to abstract syntax II: using SML

The following code will take the value bound to `thelibrary` after a graph description has been compiled and executed in the context of the definitions given in appendix C and produce an equivalent value of type `PortableGraph.graph`. (The resulting graph is isomorphic but not identical to the original because concrete *varnames* will not be recovered.)

Function `PGRecon.reconstruct` takes the value of `thelibrary` together with an integer and produces a PortableGraph.graph. The second argument must coincide with the length of the import list given by the first clause in the definiton of `thelibrary`.

```
local structure P = PortableGraph in
structure PGRecon :> sig

    type lib type env type sym type syms type misc
    type graph = P.graph
    type context = (lib, env, sym, syms, graph, misc) PGOps.context

    val reconstruct : (context -> lib list -> graph) * int -> graph
end = struct

    type lib = P.varname
    type env = P.varname
    type sym = P.varname
    type syms = P.varname
    type misc = int * P.def list
    type graph = P.graph
    type context = (lib, env, sym, syms, graph, misc) PGOps.context

    fun reconstruct (gt, nlibs) = let

        fun varname i = "v" ^ Int.toString i

        fun Bind (r, (i, d)) = let
            val (v, i') = (varname i, i + 1)
            val d' = P.DEF { lhs = v, rhs = r } :: d
        in ((i', d'), v)
        end

        fun Sgn m s = Bind (P.SYM (P.SGN, s), m)
        fun Str m s = Bind (P.SYM (P.STR, s), m)
        fun Fct m s = Bind (P.SYM (P.FCT, s), m)
        fun Syms m sl = Bind (P.SYMS sl, m)
        fun Imp m (l, ss) = Bind (P.IMPORT { lib = l, syms = ss }, m)
        fun Com m (s, e, ss, n) =
            Bind (P.COMPILE { src = (s, n), env = e, syms = ss }, m)
        fun Fil m (e, ss) = Bind (P.FILTER { env = e, syms = ss }, m)
        fun Mer m el = Bind (P.MERGE el, m)

        val im = List.tabulate (nlibs, varname)

        fun Exp (i, d) e = P.GRAPH { imports = im, defs = rev d, export = e }
    in
        gt { Ops = { Sgn = Sgn, Str = Str, Fct = Fct,
                     Exp = Exp, Syms = Syms,
                     Imp = Imp, Com = Com, Fil = Fil, Mer = Mer },
             Misc = (nlibs, []) } im
    end
end
end
```