True Separate Compilation for SML♯

Liu Bochao

January 29, 2007

1 Aim

In current days, separate compilation plays a very important role in the practical program development. As the name "separate compilation" indicates, it allows that the program is split into several source files. Each of them is self-contained in the sense it describes the import signature which are assumed to be provided by other code fragment and its own export declarations. Each unit can be compiled independently and then linked together as a whole. [1] proposes such an simple but elegant framework to describe the semantics of separate compilation and linking. Given a code fragment with the explicit import interface called *linkset*, it is compiled to a self-contained entity called *intra-checking* and then compatible modules can be safely linked together called *inter-checking*.

As for the SML language, many SML compilers support the so-called *incremental recompilation* system such as the SML/NJ [2] and MLKit[3]. In these compiler systems, the compilation of the source files must be compiled incrementally. The interface of each source file is inferred automatically by the compiler without the interference of programmer. But a weak point of such strategy is that compiler always needs to know the implementation which imposes a really inconvenient situation in practical development. Imagine the following scenario: A and B cooperates to develop a software system and B depends on A. Without the implementation of A, B cannot do any real development. This is a serious obstacle to apply SML language into large scale software development.

The ML dialect, Ocaml proposes a solution for cut-off separate compilation[4]. In Objective Caml, compilation units are regarded as a basic unit to be independently compiled and linked later. And each can be seen as special cases of structures and signatures, and the relationship between the units can be explained easily in terms of the ML module system. We use the example given in [4] to illustrate. A compilation unit A comprises two files: implementation file A.ml, consisting a sequence of definitions, analogous to the inside of a *struct* ... *end* construct; interface file A.mli, which contains a sequence of specifications, analogous to the inside of a *sig* ... *end* construct. Both files define a structure named A as if the following definition was entered at top-level as follows:

```
module A: sig (* contents of file A.mli *) end =
```

```
struct (* contents of file A.ml *) end;;
```

However since Ocaml does not provide a formalism definition, from the description of the manual we do not know the implementation details.

Compared with the popular Java and C language, Java supports the non-true separate compilation and C supports true separate compilation by explicit declarations with type information often grouped as a header file.

Our aim is to develop a true separate compilation for SML♯ and state it formally, which is a great contribution to SML community.

2 Approach

We refine the notion of linkage unit, propose an interface language to between the linkage units, and give an overall structure of true separate compilation for ML language.

Linkage unit denotes a given file consisting of a sequence of the top level declaration. Interface language is proposed to deal with the explicit import and export information. `import` keyword introduces an interface specification into the current linkage unit. It just plays the role like the signature for functor argument which declares the expected external value identifiers and type constructors. The export type and value components follows the import specification. Since the import specification provides enough type information for the external objects, linkage unit can be compiled in isolation.

The object file consists of the type environment of import specification and object code. In addition, our SML# compiler supports the using of an object file in interactive mode. To be compatible with the interactive compilation strategy, to each export value identifier we allocates a global index which corresponds to the position in a global array in runtime to hold the global values. At separate compilation, dummy global indexes are allocated to the import value identifier and export value identifier. The global index allocation environment is also pickled into the object file.

At linking time, linker takes several object files and merge them into new object file according to some certain strategy. First we specify the dependency order which is decided by the given object files order. In SML#, the linker command is executed as follows.

```
$ smlsharp -l A.smo C.smo
```

Here the linker thinks `C.smo` depends on the implementation of `A.smo`. And then the linker takes the responsibility of type check, type merging and global index reallocation.

Another problem is how to deal with open objects linking. Consider the following example,

```
(* linkage unit A *)
import
   val f : int -> int
```

```
end
val x = f 1

(* linkage unit B *)
import
   val f : bool -> bool
end
val y = f true
```

After linking the object files `A.smo` and `B.smo`, function `f` is still unresolved. But what is the expected type for it? A possible solution is to keep all the specified type for f. Until `f` is exported, the implementation type is type checked with all the specified types. Thus all the type checking complexity is postponed to that point which is a little bit unexpected. So we propose another solution by using the anti-unification algorithm to compute the least generalized type scheme. The original algorithms for anti-unification are given in [5], [6]. In this case, the least generalized type scheme for `f` is `'a -> 'a`. In the practical implementation, our `SML#` compiler does type directed compilation which thus requires that the new linked object should generate polymorphic type coercion code as follows.

```
(* new linkage unit *)
import
   val f : 'a -> 'a
end

val f' : int -> int = f
val f'' : bool -> bool = f
val x = f' 1
val y = f'' true
```

On the other hand [7] points that ML language has no greatest lower bound of the two given signatures, that is, the most abstract signature that is an implementation of those two. [7] also points out the reason that the implemenation may be given by using either module language mechnisms or polymorphism in the core language.

Another task of linker is to reallocate the global index for import and export value identifier. Note that the resolved global indexes may still not be the actual ones, i.e. the bound global index in the new linked object.

In our linking strategy, we do not demand that the export value identifier or type constructor be distinct. We take the convention in functional language community, that is, overriding the existing declared components.

3 Progress

We finished the true separate compilation in the intermediate language level. We complete the new design for kind system extension and interface language

for our separate compilation.

4 Future Direction

Our next goal in the near future will be implementing the separate compilation under the byte code interpreter framework to provide an experiement and experience for our native code oriented separate compilation.

5 Publication

"A Flattening Strategy for SML Module Compilation and Its Implementation", Liu Bochao and Atsushi Ohori, submitted to JSSST, 2006. 12

References

[1] L. Cardelli. "Program fragments, linking, and modularization". ACM Symp. on Principles of Programming Languages 1997.pp. 266-277.

[2] Appel, A.W. and D.B. MacQueen, "Separate compilation for Standard ML". Proc. 1994 ACM Conf. on Programming Language Design and Implementation,(ACM SIGPLAN Notices vol.29, number 6), 13-23, June 1994.

[3] Elsman, M. (1999). "Program modules, separate compilation, and intermodule optimisation". PhD thesis, Department of Computer Science, University of Copenhagen.

[4] "The Objective Caml system release 3.09. Documentation and user's manual". October 26, 2005.

[5] G.Plotkin. "A note on inductive generalization". Mach. Intell.,5,1970

[6] J.C. Reynolds. "Transformation systems and the algebraic structure of atomic formulas". Mach. Intell. 5,1970.

[7] Norman Ramsey, Kathleen Firsher, Paul Govereau. "An Expressive Language of Signatures". ICFP05. Sep. 2005