Separate Compilation and Cut-Off Incremental Recompilation

Martin Elsman

May 17, 1997

Abstract

We suggest how separate compilation and cut-off incremental recompilation can be obtained in the TIL compiler. Whereas separate compilation and incremental recompilation are provided by both Moscow ML and Objective Caml, none of these systems provide cut-off incremental recompilation. The system we present also provides type-safe linking.

1 Introduction

Separate compilation makes it possible to compile different units of a software project in isolation. Separate compilation has a cost, however: The programmer is required to write interfaces to other units of the software project, on which the isolated unit depends.

On the other hand, incremental recompilation does not force the programmer to specify interfaces, but requires a unit U of a software project to be compiled prior to a unit U' that depends on U. The incremental recompilation system then uses cached information (from compiling U) when compiling U'.

Separate compilation and incremental recompilation can be combined as done in Moscow ML and Objective Caml. We suggest how these techniques can be implemented for the TIL ML compiler and extended to also allow for cut-off incremental compilation, leading to less recompilation upon change of source code.

The system provides type-safe linking by "pairing" object files with (maybe summaries of) interfaces for imported and exported units. When two object files are linked, imported units of the second object file are (partly) resolved by exported units of the first object file. It is considered a link time type-error if interfaces to be resolved are not equivalent (up to renaming of bound variables). Traditional linking mechanisms are used to link actual object files. These mechanisms, however, require exported names from object files to be globally unique throughout a software project and only occasionally provide operations for redefining or eliminating names in object files. The consequence of lack of restriction and renaming is that unit names can only be used once in an entire software project. In particular, if a software project uses a library that

locally contains a unit with some unit name, then another unit in the project can not be given the same unit name.

The system implements a batch compiler and allows for traditional use of `make` to manage recompilation. As opposed to the Moscow ML system, TIL also supports the ML modules language. Since Standard ML does not allow for functors and signatures to be contained in structures (as Objective Caml does) a name space separate from structure identifiers is used for units.

In the next section we describe (informally) how target files are generated from source files and how separate compilation and incremental cut-off recompilation are combined.

2 Informal Development

Interface files hold interfaces written by the programmer. An interface file *U.int* compiles into a unit interface file *U.ui*. It is possible for the programmer to "include" other units in an interface file, using the construct

$$\texttt{include}\ U_1 \cdots U_n$$

Compilation of an interface file only succeeds if unit interface files for all included units exist. The include construct closes the interface.

An implementation file holds an ML top-level declaration (perhaps including signature declarations). As it is possible to "include" units in interface files, it is possible to "import" units in implementation files, using the construct

$$\texttt{import}\ U_1 \cdots U_n$$

To type check and compile an implementation file *U.sml*, unit interface files must exist for each imported unit. The import construct closes the implementation. First, unit interface files are "loaded" for each imported unit to form a compilation context. In this context the top-level declaration is compiled into intermediate code. At this point it is checked if a file *U.int* exists. If this is the case an interface is loaded from the file *U.ui* (which is assumed to exist) and object code is generated for the intermediate code constrained to the loaded interface, and put into the file *U.uo*.

If on the other hand, no file *U.int* exists, object code is generated from the intermediate code and put in the file *U.uo*. Further, if a file *U.ui* does not exist or the interface in it is not equivalent (up to renaming of bound variables) to the interface resulting from compiling *U.sml*, this interface is written into the file *U.ui*. Avoiding unnecessary changes in modification times of unit interface files is the key to achieve cut-off incremental recompilation.

In the following sections we describe interfaces and implementations and their compiled counterparts. We then proceed to present compilation of interfaces and implementations. Thereafter we show how type-safe linking is obtained and how compilation and recompilation is managed.

3 The External Language

In this section we extend the external language of [HS96] to facilitate separate compilation and incremental recompilation.

3.1 Interfaces

Separate compilation is achieved by allowing the user to write interfaces at the source-level. Interfaces hold top-level specifications (see below), possibly prepended with information about what units are included. Functor identifiers, structure identifiers and signature identifiers, ranged over by *funid*, *strid* and *sigid*, respectively, are defined as in [HS96]. Similarly, the grammars for signature expressions, specifications and signature bindings, ranged over by *sigexp*, *spec* and *sigbind*, are as in [HS96]. Notice that the grammar for specifications allows for functor specifications.

We assume a denumerably infinite set of *unit names*, ranged over by U and V. Top-level specifications and interfaces, ranged over by *topdec* and *int*, respectively, conform to the following grammars:

$$\begin{array}{rcl} \mathit{topspec} & ::= & \mathit{spec}\ \mathit{topspec} \\ & | & \texttt{signature}\ \mathit{sigbind}\ \mathit{topspec} \\ & | & \varepsilon \\ \mathit{int} & ::= & \texttt{include}\ U_1 \cdots U_n\ ;\ \mathit{topspec} \end{array}$$

Notice that also signatures may be specified in interface files. An implementation file for the interface file must provide declarations for each specification in the interface file.[1]

Contexts are defined in [HS96]. A *unit interface* is a context and we use *ui* and Γ to range over unit interfaces and contexts. Interfaces of the external language compile into unit interfaces in the internal language. Equality of contexts is defined up to renaming of variables and renaming of labels that are only accessed by components already in the contexts and that cannot be accessed else how. These labels are those that are not predefined or in the range of $\overline{\cdot}$. Allowing "hidden" labels to α-vary helps achieving cut-off recompilation.

3.2 Implementations

The grammar for structure declarations, ranged over by *strdec*, is as in [HS96]. Top-level declarations and implementations, ranged over by *topdec* and *sml*, respectively, conform to the following grammars:

$$\begin{array}{rcl} \mathit{topdec} & ::= & \mathit{strdec}\ \mathit{topdec} \\ & | & \texttt{signature}\ \mathit{sigbind}\ \mathit{topdec} \end{array}$$

[1] We could also allow unmatched signature specifications, but I don't know if this is a good idea...

$$
\begin{array}{rcl}
 & | & \varepsilon \\
\mathit{sml} & ::= & \texttt{import}\ U_1 \cdots U_n\ ;\ \mathit{topdec}
\end{array}
$$

The grammar for structure declarations allows for declarations of functors. Also, notice that signature declarations are allowed in implementations. Signature declarations are "compiled out" during elaboration and hence do not appear in the internal language.

A *unit environment*, $\mathcal{L}$, is a finite map from unit names to contexts. Further, a *unit object*, *uo*, is an object of the form $\langle\!\langle \mathcal{L} \mid \mathit{sbnds} \mid \mathcal{L}' \rangle\!\rangle$, where $\mathcal{L}$ is an *import* unit environment, $\mathcal{L}'$ is an *export* unit environment and *sbnds* is a sequence of structure bindings in the internal language [HS96]. As we shall see in the following section, implementations compile into unit objects.

4 Compilation

Compilation covers compilation of interfaces and compilation of implementations. Compilation of an implementation is the process of constructing a unit object for the implementation. Compilation of an interface is the process of constructing a unit interface for the interface.

4.1 Basic Inference Rules for Compilation

In this section we present basic inference rules for deriving judgments of the forms

$$\Gamma \vdash \mathit{topspec} \rightsquigarrow \Gamma' \tag{1}$$

$$\Gamma \vdash \mathit{topdec} \rightsquigarrow \mathit{sbnds} : \Gamma' \tag{2}$$

$$\Gamma \vdash \mathit{topdec}\ \texttt{:>}\ \Gamma' \rightsquigarrow \mathit{sbnds} \tag{3}$$

Judgments of the above forms are used in Section 4.3 in inference rules for compiling interface files and implementation files. Judgment (1) is read: In a context, Γ, a top-level specification, *topspec*, compiles into a context Γ'. Further, judgment (2) is read: In a context, Γ, a top-level declaration, *topdec*, is compiled into a sequence of structure bindings, *sbnds*, with resulting context Γ'. Finally, judgment (3) is read: In a context, Γ, a top-level declaration, *topdec*, constrained to the context, Γ', is compiled into a sequence of structure bindings, *sbnds*.

In the inference rules below we rely on judgments for which rules are given in [HS96].

Top-level Specifications $\boxed{\Gamma \vdash \mathit{topspec} \rightsquigarrow \Gamma'}$

$$\frac{\Gamma \vdash \mathit{spec} \rightsquigarrow [\mathit{sdecs}] : \texttt{Sig} \qquad \Gamma, \mathit{sdecs} \vdash \mathit{topspec} \rightsquigarrow \Gamma'}{\Gamma \vdash \mathit{spec}\ \ \mathit{topspec} \rightsquigarrow \mathit{sdecs}, \Gamma'} \tag{4}$$

$$\frac{\Gamma \vdash \mathit{sigbind} \rightsquigarrow \Gamma' \qquad \Gamma, \Gamma' \vdash \mathit{topspec} \rightsquigarrow \Gamma''}{\Gamma \vdash \texttt{signature}\ \mathit{sigbind}\ \mathit{topspec} \rightsquigarrow \Gamma', \Gamma''} \tag{5}$$

$$\frac{}{\Gamma \vdash \varepsilon \rightsquigarrow \cdot} \tag{6}$$

Top-level Declarations $\boxed{\Gamma \vdash \mathit{topdec} \rightsquigarrow \mathit{sbnds} : \Gamma'}$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{strdec} \rightsquigarrow \mathit{mod} : \mathit{sig} \qquad \mathit{var} \notin \mathrm{BV}(\Gamma) \\ \Gamma,\ \mathit{lbl}^* \triangleright \mathit{var} : \mathit{sig} \vdash \mathit{topdec} \rightsquigarrow \mathit{sbnds} : \Gamma'\end{array}}{\Gamma \vdash \mathit{strdec}\ \mathit{topdec} \rightsquigarrow \mathit{lbl}^* \triangleright \mathit{var} = \mathit{mod},\ \mathit{sbnds} : \mathit{lbl}^* \triangleright \mathit{var} : \mathit{sig},\ \Gamma'} \tag{7}$$

$$\frac{\Gamma \vdash \mathit{sigbind} \rightsquigarrow \Gamma' \qquad \Gamma, \Gamma' \vdash \mathit{topdec} \rightsquigarrow \mathit{sbnds} : \Gamma''}{\Gamma \vdash \texttt{signature}\ \mathit{sigbind}\ \mathit{topdec} \rightsquigarrow \mathit{sbnds} : \Gamma', \Gamma''} \tag{8}$$

$$\frac{}{\Gamma \vdash \varepsilon \rightsquigarrow \cdot : \cdot} \tag{9}$$

$\boxed{\Gamma \vdash \mathit{topdec}\ \texttt{:>}\Gamma' \rightsquigarrow \mathit{sbnds}}$

$$\frac{\Gamma \vdash \mathit{topdec} \rightsquigarrow \mathit{sbnds} : \Gamma'' \qquad \Gamma \vdash_{\mathrm{subtop}} \Gamma'' \preceq \Gamma' \rightsquigarrow \mathit{sbnds}'}{\Gamma \vdash \mathit{topdec}\ \texttt{:>}\Gamma' \rightsquigarrow \mathit{sbnds},\ \mathit{sbnds}'} \tag{10}$$

Signature Bindings $\boxed{\Gamma \vdash \mathit{sigbind} \rightsquigarrow \Gamma'}$

$$\frac{\Gamma \vdash \mathit{sigexp} \rightsquigarrow \mathit{sig} : \texttt{Sig} \qquad \mathit{var} \notin \mathrm{BV}(\Gamma\langle,\ \Gamma'\rangle) \qquad \langle \Gamma \vdash \mathit{sigbind} \rightsquigarrow \Gamma' \rangle}{\Gamma \vdash \mathit{sigid} = \mathit{sigexp}\ \langle \texttt{and}\ \mathit{sigbind} \rangle \rightsquigarrow \overline{\mathit{sigid}} \triangleright \mathit{var} : \texttt{Sig} = \mathit{sig}\langle,\ \Gamma' \rangle} \tag{11}$$

Structure Binding Coercions[2] $\boxed{\Gamma \vdash_{\mathrm{subtop}} \Gamma' \preceq \Gamma'' \rightsquigarrow \mathit{sbnds}}$

$$\frac{\begin{array}{c}\Gamma' \vdash_{\mathrm{ctx}} \mathit{lbl} \rightsquigarrow \mathit{sig}' : \texttt{Sig} \\ \Gamma \vdash \mathit{sig} \equiv \mathit{sig}' : \texttt{Sig} \qquad \Gamma \vdash_{\mathrm{subtop}} \Gamma' \preceq \Gamma'' \rightsquigarrow \mathit{sbnds}\end{array}}{\Gamma \vdash_{\mathrm{subtop}} \Gamma' \preceq \mathit{lbl} \triangleright \mathit{var} : \texttt{Sig} = \mathit{sig},\ \Gamma'' \rightsquigarrow \mathit{sbnds}} \tag{12}$$

[2]The rule for instantiating a polymorphic value component to a monomorphic value component has been left out.

$$\frac{\begin{array}{c}\Gamma' \vdash_{\text{ctx}} lbl \rightsquigarrow path : sig_0 \\ \Gamma \vdash_{\text{sub}} sig_0 \preceq sig \rightsquigarrow \lambda var_0 : sig_0.mod : (var_0 : sig_0) \to sig' \\ \Gamma,\ lbl \triangleright var : sig \vdash_{\text{subtop}} \Gamma' \preceq \Gamma'' \rightsquigarrow sbnds\end{array}}{\begin{array}{c}\Gamma \vdash_{\text{subtop}} \Gamma' \preceq lbl \triangleright var : sig,\ \Gamma'' \rightsquigarrow \\ lbl \triangleright var' = ((\lambda var_0 : sig_0.mod)\ path) : sig,\ sbnds\end{array}} \tag{13}$$

$$\frac{\begin{array}{c}\Gamma' \vdash_{\text{ctx}} lbl \rightsquigarrow path : con' \quad \Gamma \vdash con \equiv con' : \Omega \\ \Gamma,\ lbl \triangleright var' : con \vdash_{\text{subtop}} \Gamma' \preceq \Gamma'' \rightsquigarrow sbnds \quad var' \notin \text{BV}(\Gamma)\end{array}}{\Gamma \vdash_{\text{subtop}} \Gamma' \preceq lbl \triangleright var : con,\ \Gamma'' \rightsquigarrow lbl \triangleright var' = path,\ sbnds} \tag{14}$$

$$\frac{\begin{array}{c}\Gamma' \vdash_{\text{ctx}} lbl \rightsquigarrow path : knd' \quad \langle \Gamma \vdash path \equiv con : knd \rangle \\ \Gamma,\ lbl \triangleright var' : knd\langle = con\rangle \vdash_{\text{subtop}} \Gamma' \preceq \Gamma'' \rightsquigarrow sbnds \quad var' \notin \text{BV}(\Gamma)\end{array}}{\Gamma \vdash_{\text{subtop}} \Gamma' \preceq lbl \triangleright var : knd\langle = con\rangle,\ \Gamma'' \rightsquigarrow lbl \triangleright var' = path,\ sbnds} \tag{15}$$

$$\frac{}{\Gamma \vdash_{\text{subtop}} \Gamma' \preceq \cdot \rightsquigarrow \cdot} \tag{16}$$

4.2 Projects

A *file* is a pair of a unit name and an object being either an *implementation*, *sml*, an *interface*, *int*, a *unit object*, *uo* or a *unit interface*, *ui*. In the following we use A and B to range over files.

A *project*, p, is defined by the following grammar:

$$p \quad ::= \quad \bullet \ \mid\ p,\ U.sml \ \mid\ p,\ U.int \ \mid\ p,\ U.uo \ \mid\ p,\ U.ui$$

When p is a project and p' is a project of the form $\bullet, A_1, \cdots, A_n,\ n \geq 0$, we write p, p' to mean the project $p, A_1, \cdots, A_n$. The order in which files occur in a project models the order of creation (or modification) of files. When p is a project we say that B *is newer than* A *in* p, written $A <_p B$, if there exist projects p_1, p_2 and p_3 such that $p = p_1, A, p_2, B, p_3$. When the order in which files occur in a project is of no importance, we sometimes treat projects as sets of files. Updating a project, p, with a file, A, written $p \oplus A$, is defined as follows:

$$p \oplus U.sml \ = \ \begin{cases} p_1,\ p_2,\ U.sml & \text{if } p = p_1,\ U.sml',\ p_2 \ \text{ for some } p_1, p_2, sml' \\ p,\ U.sml & \text{otherwise} \end{cases}$$

$$p \oplus U.int \ = \ \begin{cases} p_1,\ p_2,\ U.int & \text{if } p = p_1,\ U.int',\ p_2 \ \text{ for some } p_1, p_2, int' \\ p,\ U.int & \text{otherwise} \end{cases}$$

$$p \oplus U.uo \ = \ \begin{cases} p_1,\ p_2,\ U.uo & \text{if } p = p_1,\ U.uo',\ p_2 \ \text{ for some } p_1, p_2, uo' \\ p,\ U.uo & \text{otherwise} \end{cases}$$

$$p \oplus U.ui \ = \ \begin{cases} p_1,\ p_2,\ U.ui & \text{if } p = p_1,\ U.ui',\ p_2 \ \text{ for some } p_1, p_2, ui' \\ p,\ U.ui & \text{otherwise} \end{cases}$$

4.3 Inference Rules for Compilation

Inference rules for compilation includes inferences among sentences of the forms

$$p \vdash U.sml \Rightarrow p' \tag{17}$$
$$p \vdash U.int \Rightarrow p' \tag{18}$$

where p and p' are projects, $U.sml$ an implementation file and $U.int$ an interface file.

Implementations $\boxed{p \vdash U.sml \Rightarrow p'}$

$$\frac{\begin{array}{c} U.sml \in p \quad sml = \texttt{import}\ U_1 \cdots U_n\ ;\ topdec \\ \mathcal{L} = \{U_1 \mapsto ui_1, \cdots, U_n \mapsto ui_n\} \quad \{U_1.ui_1, \cdots, U_n.ui_n\} \subseteq p \\ \nexists int\,.\, U.int \in p \quad ui_1, \cdots, ui_n \vdash topdec \rightsquigarrow sbnds : ui \quad U.ui \notin p \end{array}}{p \vdash U.sml \Rightarrow p \oplus U.\langle\!\langle \mathcal{L} \mid sbnds \mid \{U \mapsto ui\} \rangle\!\rangle \oplus U.ui} \tag{19}$$

$$\frac{\begin{array}{c} U.sml \in p \quad sml = \texttt{import}\ U_1 \cdots U_n\ ;\ topdec \\ \mathcal{L} = \{U_1 \mapsto ui_1, \cdots, U_n \mapsto ui_n\} \quad \{U_1.ui_1, \cdots, U_n.ui_n\} \subseteq p \\ \nexists int\,.\, U.int \in p \quad ui_1, \cdots, ui_n \vdash topdec \rightsquigarrow sbnds : ui \quad U.ui \in p \end{array}}{p \vdash U.sml \Rightarrow p \oplus U.\langle\!\langle \mathcal{L} \mid sbnds \mid \{U \mapsto ui\} \rangle\!\rangle} \tag{20}$$

$$\frac{\begin{array}{c} U.sml \in p \quad sml = \texttt{import}\ U_1 \cdots U_n\ ;\ topdec \\ \mathcal{L} = \{U_1 \mapsto ui_1, \cdots, U_n \mapsto ui_n\} \quad \{U_1.ui_1, \cdots, U_n.ui_n\} \subseteq p \\ U.ui \in p \quad U.int \in p \quad ui_1, \cdots, ui_n \vdash topdec\ \texttt{:>}\ ui \rightsquigarrow sbnds \end{array}}{p \vdash U.sml \Rightarrow p \oplus U.\langle\!\langle \mathcal{L} \mid sbnds \mid \{U \mapsto ui\} \rangle\!\rangle} \tag{21}$$

Interfaces $\boxed{p \vdash U.int \Rightarrow p'}$

$$\frac{\begin{array}{c} U.int \in p \quad int = \texttt{include}\ U_1 \cdots U_n\ ;\ topspec \\ \{U_1.ui_1, \cdots, U_n.ui_n\} \subseteq p \quad ui_1, \cdots, ui_n \vdash topspec \rightsquigarrow ui \end{array}}{p \vdash U.int \Rightarrow p \oplus U.ui} \tag{22}$$

The rules (19) and (20) implement incremental recompilation and cut-off incremental recompilation, respectively. The rules (21) and (22) implement separate compilation.

5 Type-Safe Linking

Linking is the process of combining two (or more) unit object files into one unit object file. When $\mathcal{L}$ and $\mathcal{L}'$ are unit environments we define the *restricted addition* of $\mathcal{L}$ and $\mathcal{L}'$, written $\mathcal{L} \uplus \mathcal{L}'$ as follows:

$$\mathcal{L} \uplus \mathcal{L}' \quad = \quad \begin{cases} \mathcal{L} + \mathcal{L}' & \text{if } \mathrm{dom}(\mathcal{L}(U)) \cap \mathrm{dom}(\mathcal{L}'(V)) = \emptyset, \\ & \quad \text{for all } U \in \mathrm{dom}(\mathcal{L}) \text{ and } V \in \mathrm{dom}(\mathcal{L}') \\ & \quad \text{such that } U \neq V \\ \text{undefined} & \text{otherwise} \end{cases} \tag{23}$$

Linking is then defined as follows:

Linking $\boxed{\mathrm{Link}(U_1.uo_1, U_2.uo_2) = U.uo}$

$$\frac{\begin{array}{c} \mathrm{dom}(\mathcal{L}_1') \cap \mathrm{dom}(\mathcal{L}_2') = \emptyset \quad U \notin \mathrm{dom}(\mathcal{L}_1') \cup \mathrm{dom}(\mathcal{L}_2') \\ \mathcal{L}_1'(V) = \mathcal{L}_2(V) \text{ for all } V \in \mathrm{dom}(\mathcal{L}_1') \cap \mathrm{dom}(\mathcal{L}_2) \\ \mathcal{L}_1(V) = \mathcal{L}_2(V) \text{ for all } V \in \mathrm{dom}(\mathcal{L}_1) \cap \mathrm{dom}(\mathcal{L}_2) \\ uo = \langle\!\langle \mathcal{L}_1 \uplus (\mathcal{L}_2 \downarrow (\mathrm{dom}(\mathcal{L}_2) \setminus \mathrm{dom}(\mathcal{L}_1'))) \mid sbnds_1,\ sbnds_2 \mid \mathcal{L}_1' \uplus \mathcal{L}_2' \rangle\!\rangle \end{array}}{\mathrm{Link}(U_1.\langle\!\langle \mathcal{L}_1 \mid sbnds_1 \mid \mathcal{L}_1' \rangle\!\rangle, U_2.\langle\!\langle \mathcal{L}_2 \mid sbnds_2 \mid \mathcal{L}_2' \rangle\!\rangle) = U.uo} \tag{24}$$

The restricted addition relation can be implemented by prepending all labels generated during compilation of an implementation with its unit name. In this way we can assure that no labels "clash."

To build an executable from a unit object we require the import unit environment to be empty.

5.1 Probabilistic Type-Safe Linking

Instead of storing unit interfaces in unit objects, one can do with storing summaries (a 20-byte checksum, say) of the string representation of unit interfaces. The only operation required on unit interfaces in unit objects is equality and this operation can be approximated by equality of summaries of interfaces. The Moscow ML system takes this approach to type-safe linking.

Before computing the 20-byte checksum of a string representation of a unit interface, the version of the compiler can be appended to the string. This ensures that code generated by one version of a compiler is not - by mistake - considered compatible with code generated by another version of the compiler.

6 Managing Recompilation

To manage compilation and recompilation of a software project the UNIX utility `make` can be used. This utility is based on dependency information among files of a project and the exact order object files should be linked.

Dependencies among files of a project can be derived solely from the source files of the project. A program `tilmkdep` derives dependency information and inserts this information into a `Makefile` to be processed by `make`.

When A and B are files and p is a project, we write $A \leftarrow_p B$ to mean "A depends on B in p". The dependency relation can be derived from the rules in

the previous section. The following two dependencies can be derived from the rule concerning compilation of interfaces:

$$U.ui \leftarrow_p U.int \quad \text{if } U.int \in p \tag{25}$$
$$U.ui \leftarrow_p V.ui \quad \text{if } U.int \in p \text{ and } U.int \text{ includes } V \tag{26}$$

From the rules concerning compilation of implementations the following dependencies can be derived:

$$U.uo \leftarrow_p U.sml \quad \text{if } U.sml \in p \tag{27}$$
$$U.uo \leftarrow_p V.ui \quad \text{if } U.sml \in p \text{ and } U.sml \text{ imports } V \tag{28}$$
$$U.uo \leftarrow_p U.ui \quad \text{if } U.int \in p \tag{29}$$

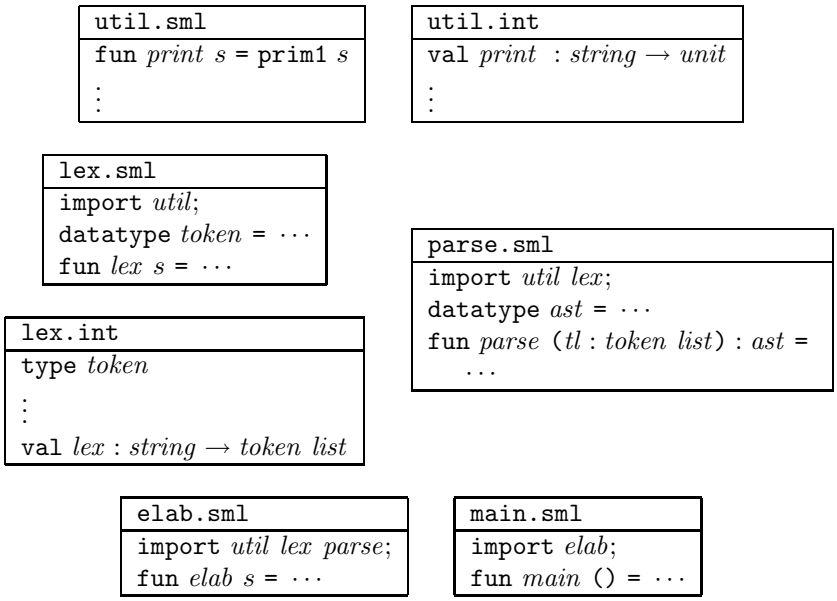
If A depends on B in p, possibly transitively, and B is modified, then A must be recompiled, since the assumptions on which it is built may have changed. Consistency of a project is thus defined as follows:

Definition 6.1 (Consistency) A project p *is consistent* if for all files A and B, if $A \leftarrow_p B$ then $B <_p A$. ■

Dependencies of the forms (25) and (27) are modeled in the `Makefile` for a project by pattern rules (see manual page for GNU `make` and the example below). Dependencies of the forms (26), (28) and (29) are derived by the program `tilmkdep` and inserted in the `Makefile` for the project. The user needs only run `tilmkdep` initially and when dependencies change.

6.1 An Example: The Diamond

Consider a project consisting of the following source files:



```
objects=util.uo lex.uo parse.uo elab.uo main.uo

all: $(objects)
        tilc $(objects)

%.ui : %.int
        tilc -c $<

%.uo : %.sml
        tilc -c $<

depend:
        tilmkdep Makefile

clean:
        rm -f *.ui *.uo Makefile.bak

### DO NOT DELETE THIS LINE
util.uo: util.ui
lex.uo: lex.ui util.ui
elab.uo: parse.ui lex.ui util.ui
parse.uo: util.ui lex.ui
main.uo: elab.ui
```

Figure 1: Sample `Makefile` for managing separate compilation in TIL under UNIX.

A `Makefile` for the example is given in Figure 1. All dependencies under the line

```
### DO NOT DELETE THIS LINE
```

have been inferred by the `tilmkdep` program as described above. Notice that the implementation file `parse.sml` has no corresponding interface file.

In the unit interface files for the units `parse` and `lex`, the label associated with the type constructor *token* will be `lex_token`, hence the types will be compatible when compiling the implementation `elab.sml`.

If the programmer modifies the source file `parse.sml` in such a way that the resulting unit interface is equivalent to the unit interface in the unit interface file `parse.ui`, then recompilation of the units `elab` and `main` is avoided. Without cut-off incremental recompilation both of these units are recompiled.

7 Related Work

Many of these ideas are taken from the Objective Caml user's manual and the Moscow ML Owner's Manual:

```
http://pauillac.inria.fr/ocaml/htmlman/
http://www.dina.kvl.dk/~sestoft/mosml.html
```

The Objective Caml compiler uses `open` instead of `import` and allows the programmer to use qualified names. We distinguish `import` from `open`, since Standard ML does not allow signatures and functors in structures, and the `import` construct may also import such objects. Whereas `open` is provided at the modules level, `import` is provided at the compilation unit level.

8 Conclusion

I have implemented a small "pseudo compiler" that updates files and checks for equality of interfaces. The "pseudo compiler" worked perfectly well together with `make` and the sample `Makefile` given in this document; upon change of source code only the expected units were "recompiled".

To summarize, the system requires the following support from the compiler:

- Compilation functions;
- Context read and write from files. This we can do by blast-in and blast-out provided by SML/NJ. This puts requirement on bootstrapping, however;
- Context equality (α-equivalence up to renaming of bound variables and hidden labels; hidden labels for a unit are not accessed by other units.

Unit object files are implemented as follows (unit environments map unit identifiers to checksums of interface unit files):

Import unit environment;
Export unit environment;
Target object code.

Keeping import and export environments in the unit object files makes it close to impossible for the user to "fool" the system.

References

[HS96] Robert Harper and Chris Stone. A type-theoretic account of Standard ML 1996 (version 2). Carnegie Mellon University, Technical Report CMU-CS-96-136R. Also appears as Fox Memorandum CMU-CS-FOX-96-02R, September 1996.