A Framework for Cut-Off Incremental Recompilation and Inter-Module Optimization

Martin Elsman
IT University of Copenhagen, Denmark
mael@itu.dk

Abstract

In this paper we present a cut-off incremental recompilation framework that supports inter-module optimization. The framework allows arbitrary compile time information to propagate across program unit boundaries, in such a way that it can be determined if compilation assumptions have changed since the program unit was last compiled.

The abstract presentation of the framework makes explicit the assumptions of the approach and specifies exactly the set of operations necessary for each of the translation phases in a compiler (from parsing to abstract syntax, through a series of intermediate languages, and eventually down to machine code). The correctness results shown are non-trivial due to the flexibility of the framework, which allows even open terms (objects containing free occurrences of names) to propagate across program unit boundaries at compile time.

The framework is based on a language for programming in the very large, which allows for expressing precise program unit dependencies. Moreover, framework is batch-based in the sense that only one source file is compiled for each invocation of the compiler. The scheme is applied in the MLKit Standard ML compiler and experiments show that the approach is feasible and that it scales to very large programs and day-to-day program development for programs consisting of more than 250.000 lines of Standard ML.

Categories and Subject Descriptors D.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords Separate compilation, Inter-module optimization, Serialization, Standard ML

1. Introduction

Modularity arose from the need to divide programs into program units for separate compilation. Since then, the concept of modularity has become a key element of modern software engineering, covering name space management and abstraction mechanisms, such as parametric modules, class mechanisms, and abstract types. As software projects get larger, programming language support for modularity becomes more and more important for software development, for software maintenance, and for software reuse. On the other hand, only rarely does modularity come for free; when a program unit is compiled, many programming language systems impose a restriction to what information is available to the compiler about identifiers declared in other program units [11]. Such restrictions may limit what optimizations are performed across program unit boundaries, which again may encourage programmers to avoid modularization and abstraction mechanisms to enable classical optimizations such as function in-lining, function specialization and constant propagation [2], and newer optimizations and analyses, such as function fusion [24] and region inference [30, 33].

Most programming language systems provide a way of programming in the very large, either through the use of `make` [19] or through a more system specific `make`-like facility. The framework we present here is based on ML Basis files [12], a language used in the whole-program optimizing Standard ML compiler MLton [12] and in the MLKit [32, 31] for expressing dependencies between source files.

The framework exists independently of the source language and imposes little restrictions on what information may be propagated across program unit boundaries. We call the kind of recompilation provided by the framework for *cut-off incremental recompilation* because program units managed by the framework are compiled incrementally (i.e., a program unit may be compiled only when the program units on which it depends have been compiled) and because a program unit need not necessarily be recompiled if other program units, on which it depends, are modified. The framework is based both on properties of the source language and on properties of each translation step in the compiler. An important property of the framework is that it may coexist with a module language; thus, the framework does not compromise software engineering principles.

1.1 Contributions

The main contributions of this paper are the following:

- A theoretical foundation for a cut-off incremental recompilation framework that allows propagation of compile time information across program unit boundaries, enabling inter-module optimization. The abstract presentation of the framework specifies exactly the set of operations necessary for each of the translation phases in a compiler, which provides the compiler writer with a clear interface for plugging in additional compilation phases and optimizations. Correctness of the approach is non-trivial due to the flexibility of the framework, which allows even open terms (objects containing free occurrences of names) to propagate across program unit boundaries.
- A description of how the theoretical foundations are used in a practical (re)compilation framework for the MLKit Standard ML compiler.

Copyright © 2008, Martin Elsman, IT University of Copenhagen, Denmark. All rights reserved. Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

IT University of Copenhagen Technical Report. April 2008.

- Measurements showing that the (re)compilation framework is feasible in practice and that it enables optimizations that are not possible with traditional recompilation systems.

1.2 Outline

In Section 1.3, we first give an overview of related work. Then in Section 2, we give a short introduction to ML Basis (MLB) compilation management and describe how it works well together with inter-module optimization and cut-off incremental recompilation.

In Section 3, we develop the foundations for cut-off incremental recompilation management. Based on a notion of a translation phase and properties that must hold for each translation phase in a compiler, we define the concept of compilation as the composition of a series of translation phases. Moreover, we demonstrate some important properties of compilation based on the properties of each compilation phase.

We develop the concept of MLB (re)compilation management in two steps, based on the notion of compilation. First, in Section 4, we present a set of MLB compilation inference rules that closely resembles the MLB static semantics developed in [12] for Standard ML.

Second, in Section 5, we introduce the concepts of repository and dependency maps for presenting a set of inference rules for MLB recompilation management. We show that the compilation and recompilation semantics are related in a particular sense and that, with respect to so-called well-formed repositories, recompilation is sound and complete. An important property of the recompilation semantics is that an implementation based on a simple dependency analysis and serialization of compilation bases to disk can follow the semantics closely. To simplify the presentation, we make the assumption in Section 5 that MLB files are self-contained, that is, that they do not contain references to other MLB files.

In Section 6, we describe how the framework is instantiated to the context of Standard ML and a particular Standard ML compiler, namely the MLKit [32], which allows compilation information to migrate across compilation unit boundaries at compile time. We also present solutions to some scalability problems incurred by the framework and present experimental evidence that the framework scales to large programs, even in settings that deploy extensive inter-module optimizations.

Finally, in Section 7, we describe future work and conclude.

1.3 Related Work

ML modules and other high-level language constructs, such as classes, allow for programming in the large, but cannot be used to express how source files of a system are combined. In particular, a programmer cannot within the language specify source file dependencies, which could allow the programmer to reason about software components (groups of source files) for programming in the very large. In the context of Standard ML, different ML compilers support their own take on these issues. For instance, the SML/NJ team has developed the concept of CM files [8, 9], the MLKit team has developed the concept of PM files [14, 32], and for Moscow ML, the `mosmake` tool was developed [21]. Recently, the MLton team has developed the concept of ML Basis files [12], an extension of the PM files supported by earlier versions of the MLKit. Each of these tools allows the programmer to specify how source files should be grouped together to form a complete program or library.

Various recompilation schemes have been investigated ranging from smart recompilation [29, 1] to Shao and Appel's *smartest recompilation* scheme [27]. Smart recompilation has the property that a program unit must be recompiled whenever (1) its own implementation changes, or (2) an interface changes upon which the program unit depends. Shao and Appel's scheme uses type

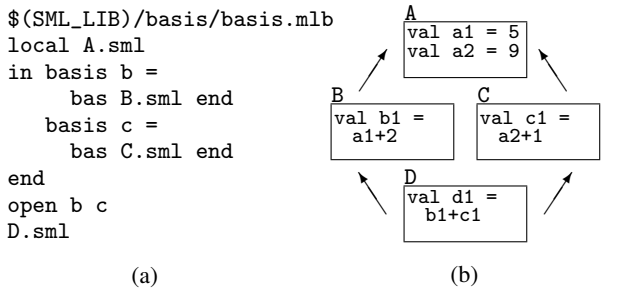

Figure 1. Example ML Basis File (a) and corresponding dependency graph (b) with embedded ML source code.

inference to infer types for undeclared identifiers of a program unit, which leads to the property that a program unit needs not be recompiled unless its own implementation changes.

Frameworks for separate compilation that allow a program unit to be compiled, based on interfaces provided by the programmer, support cut-off (or true) separate compilation. Lately, Swasey et al. [28] have proposed a language extension for Standard ML so as to support true separate compilation. The proposal extends Standard ML with the possibility of specifying functor signatures, which allows for separate compilation of program units that refer to functors declared in other program units. Whereas the language extensions of the proposal do not conflict with the framework we present here, the proposal of Swasey et al. does not provide separate compilation for implementations of Standard ML that critically depend on propagating information other than language-level types across program unit boundaries. Also, the work by Swasey et al. does not focus on avoiding unnecessary recompilation in cases where module interfaces are not provided by a programmer.

Also closely related to the present work is Blume's work on hierarchical modularity [9, 10] in the context of the CM compilation management system for SML/NJ [8]. Blume's work does not allow arbitrary symbol table information to propagate across program unit boundaries at compile time. CM makes use of a program analysis to compute dependencies between source files, which is not required in the case of ML Basis files where source file dependencies are made explicit.

Also related to our work, Ancona et al. have investigated recompilation and separate compilation techniques for Java-like languages [3, 4, 5], but with little focus on inter-module optimization.

Finally, there is a large body of related work on frameworks for whole-program optimizations (e.g., [34]) and whole-program compilation (e.g., [35]). Contrary to the focus of this large body of related work, our work focuses on avoiding unnecessary recompilation in the presence of inter-module optimization and inter-module program analyses.

2. ML Basis Files

To motivate the use of ML Basis Files to specify dependencies between program units, consider the MLB file in Figure 1(a). This MLB file illustrates various aspects of MLB files. The first line illustrates that it is possible to "load" the basis corresponding to another ML Basis file. In this case, the Standard ML Basis Library is loaded, which in effect is made available to all source files mentioned in the MLB file. (The *path variable* `$(SML_LIB)` is used to refer to libraries in a platform independent way.) The MLB file also mentions four source files `A` (i.e., `A.sml`), `B`, `C`, and `D`, and two basis identifiers `b` and `c`. Notice how the `local` declaration of `A` is used to specify that `B` is compiled in the basis resulting from compiling `A` and that the MLB `basis` bindings are used to specify that also `C` is compiled in the basis resulting from compiling

A. Using the open construct, the resulting bases from compiling B and C are made available for compiling D. In this respect, the MLB file expresses dependencies between the source files A, B, C, and D, according to the diagram in Figure 1(b), where arrows indicate dependencies. Thus B and C depend on A, and D depends on B and C. This example illustrates some of the flexibilities of MLB files and how MLB files allow for expressing precise dependencies between source files in terms of the underlying directed acyclic dependency graph.

For illustrating various recompilation scenarios, we consider the example in Figure 1, where source files contain simple ML declarations; all the points we make here carry over to more advanced language constructs such as ML structures, signatures, and functors. When the entire program is compiled, each source file has been compiled into object code that may be linked together and executed, resulting in d1 evaluating to the value 17.

Consider now the effect of modifying the variable binding of a1 in source file A to "val a1 = 4". Further, assume that only simple type information (such as: a1 has type int) is propagated across program unit boundaries at compile time. Due to the source file modification, source file A needs to be recompiled. But due to unchanging compilation assumptions, none of the program units B, C, and D needs to be recompiled. On the other hand, consider the same change in a compiler that features inter-module constant-propagation. By recording constant-propagation information in compilation bases, the assumptions under which source file B is to be compiled have changed, thus recompiling B under the new assumptions is necessary. Moreover, although C needs not be recompiled (restricted to the free identifiers of the source file, the assumptions under which it is compiled have not changed), program unit D does need to be recompiled, because the assumptions stemming from compiling B has changed.

A somewhat simpler example of an MLB file is a list of source files preceded by a reference to the MLB file representing the Standard ML Basis Library. The meaning of such an MLB file is equivalent to the meaning of the program obtained by concatenating all source files (in order), including the source files implementing the Standard ML Basis Library.

Notice that the concept of MLB files serves several purposes. First, MLB files can be seen as a crude way of extending the Standard ML module language, for providing better support for grouping together signatures, structures, and functors (Standard ML has no support for including functors and signatures in structures). Second, MLB files link the Standard ML module language (and the Core language, for that matter) to the file system and make explicit the dependencies between source files. It is this latter property of MLB files that is our focus in this work.

2.1 Grammar for ML Basis Files

The syntax of ML Basis files is given by the syntax for *basis expressions* (*bexp*) and *basis declarations* (*bdec*), as defined below. We assume a denumerable infinite set of *basis identifiers* BId, ranged over by *bid*. We use *longbid* to range over *long basis identifiers*, that is, non-empty lists of basis identifiers separated by a punctuation letter (.). We further assume a denumerable infinite set MlbId of *MLB file identifiers*, ranged over by *mlbid*, and a denumerable infinite set UId of *program unit identifiers*, ranged over by *uid*. Finally, we assume p to range over possible source program units SrcProg.

$$
\begin{array}{rcl}
\mathit{bexp} & ::= & \texttt{bas}\ \mathit{bdec}\ \texttt{end} \\
& | & \texttt{let}\ \mathit{bdec}\ \texttt{in}\ \mathit{bexp}\ \texttt{end} \\
& | & \mathit{longbid}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{bdec} & ::= & \mathit{bdec}\ \mathit{bdec} \\
& | & \varepsilon \\
& | & \texttt{local}\ \mathit{bdec}\ \texttt{in}\ \mathit{bdec}\ \texttt{end} \\
& | & \texttt{basis}\ \mathit{bid} = \mathit{bexp} \\
& | & \texttt{open}\ \mathit{longbid} \\
& | & \mathit{mlbid} \triangleright \mathit{bdec} \\
& | & \mathit{uid} \triangleright p
\end{array}
$$

We refer to the above grammar as the *annotated* MLB grammar and the above grammar with the "$\triangleright$ *bdec*" and "$\triangleright\ p$" parts removed as the *unannotated* MLB grammar. Moreover, we refer to the above grammar with the construct "*mlbid* $\triangleright$ *bdec*" removed as the *MLB file free* MLB grammar.

Notice that the MLB grammar much resembles the grammar for structure declarations and structure expressions in Standard ML [22], but without support for signatures and functors. For simplicity, no Standard ML language constructs have been lifted to the level of ML Basis Files, which leads to a cleaner separation between an actual compiler implementation and an implementation of the cut-off incremental recompilation framework.

Because module identifiers have no special status in the framework, information about type constructors and value identifiers is propagated across program unit boundaries just as well as information about signatures, structures, and functors.

2.2 A Note on the Semantics of ML Basis Files

We shall not in this paper give a concrete static semantics for ML Basis Files based on a static semantics for the concrete embedded language, as in [12]. Nor shall we present a dynamic semantics for ML Basis Files based on a dynamic semantics of the embedded language. Instead, we shall define the semantics of ML Basis Files as the result of *ML Basis File compilation*, which, as we shall see, is defined in terms of the individual translation steps in a compiler.

3. Foundation of Compilation

In this section, we develop the concept of compilation based on individual translation steps in a compiler. At each translation step, compile time information is allowed to migrate across compilation unit boundaries in so-called translation environments, whose product make up what we shall define as compilation bases.

We stress here that compilation abstracts over the particular translation steps used in a compiler, but that each translation step must meet certain requirements, as specified in the sections to follow.

3.1 Translation Environments

We assume a denumerable infinite set Name of names, ranged over by n. For simplicity, we assume that source program identifiers are members of Name. The set of all subsets of Name is written $\mathrm{NameSet}$ and we use N to range over $\mathrm{NameSet}$. The *disjoint union* of name sets N and N', written $N \uplus N'$, equals $N \cup N'$, but is defined only when $N \cap N' = \emptyset$. When A is any object (or sequence of objects), we write $\mathrm{names}(A)$ to mean the set of names that occur free in A; we assume that what free means is defined for actual objects used in the framework.

A *(translation) environment* E for a translation step in the compiler is a finite map from names to translation objects; we use τ to range over translation objects for some translation step, but we shall not define translation objects here, as such objects are specific to actual translation steps. However, we assume a notion of equality on translation objects; when τ_1 and τ_2 are translation objects, we write $\tau_1 = \tau_2$ iff τ_1 equals τ_2. When f is a finite map, we write $\mathrm{Dom}(f)$ and $\mathrm{Ran}(f)$ to denote the domain and co-domain of f, respectively. The set of names that occur free in an environment E, written $\mathrm{names}(E)$, is the set $\mathrm{Dom}(E) \cup \mathrm{names}(\mathrm{Ran}(E))$.

We now define a relation on environments called enrichment:

DEFINITION 1 (Enrichment). *An environment* E_1 enriches *another environment* E_2, *written* $E_1 \sqsupseteq E_2$, *iff* $\mathrm{Dom}(E_1) \supseteq \mathrm{Dom}(E_2)$ *and* $E_1(n) = E_2(n)$ *for all* $n \in \mathrm{Dom}(E_2)$.

Enrichment is reflexive, transitive, and antisymmetric, thus, for a given translation step in the compiler, enrichment defines a partial order on environments. The *restriction* of a finite map E to a set $N \subseteq \mathrm{Dom}(E)$, written $E \downarrow N$, is the finite map with domain N and values $(E \downarrow N)(x) = E(x)$ for all $x \in N$.

We define the *extension* of an environment E with an environment E', written $E + E'$, as the environment with domain $\mathrm{Dom}(E) \cup \mathrm{Dom}(E')$ and values

$$(E + E')(x) = \begin{cases} E'(x) & \text{if } x \in \mathrm{Dom}(E') \\ E(x) & \text{if } x \in \mathrm{Dom}(E) \setminus \mathrm{Dom}(E') \\ \text{undefined} & \text{otherwise} \end{cases}$$

3.2 Translation Steps

We use p to range over program units (of some translation step) of the compiler. When p is a program unit, we write $\mathrm{uses}(p)$ to mean the set of names that appear as uses in p and we write $\mathrm{decls}(p)$ to mean the set of names that are declared by p; declared names of a program unit do not alpha-vary.

In our framework, each translation step is assumed to be given on the form

$$E \vdash p \Rightarrow \exists N.(E', p')$$

where E and E' are environments, p is a source program unit for the translation step, p' is a target program unit for the translation step, and N is the set of names that are generated during translation. Sentences of this form are read "p translates to $\exists N.(E', p')$ in E." The prefix $\exists N$ in the object $\exists N.(E', p')$ binds names and we consider objects of this form equal up to renaming of bound names. We sometimes use Φ to range over objects of the form $\exists N.(E, p)$.

The framework assumes a set of properties be met. First, used and declared names of a program must relate appropriately to environments and generated names must be mentioned explicitly in the names set N:

PROPERTY 1 (Name relations). *If* $E \vdash p \Rightarrow \exists N.(E', p')$ *then*

1. $\mathrm{uses}(p) \subseteq \mathrm{Dom}(E)$
2. $\mathrm{decl}(p) = \mathrm{Dom}(E')$
3. $\mathrm{names}(E', p') \subseteq N \cup \mathrm{names}(E, p)$
4. $\mathrm{uses}(p') \subseteq \mathrm{names}(\mathrm{Ran}(E \downarrow \mathrm{uses}(p)))$

Property 1(4) states that used names of a target program unit of a translation step must stem from propagating uses of the source program unit of the translation step through the translation environment. Thus, if some translation step translates a program unit p to another program unit p' then there is a connection between the uses of p and the uses of p'; we shall return to the importance of this property in Section 3.4.

Second, each translation step is assumed to be closed under enrichment:

PROPERTY 2 (Translation closed under enrichment). *If* $E \vdash p \Rightarrow \Phi$ *and* $E' \sqsupseteq E$ *then* $E' \vdash p \Rightarrow \Phi$.

The third property guarantees that a translation step depends only on those assumptions for which identifiers occur free in the source program unit:

PROPERTY 3 (Translation closed under restriction). *If* $E \vdash p \Rightarrow \Phi$ *then there exists* E' *such that* $E' = E \downarrow \mathrm{uses}(p)$ *and* $E' \vdash p \Rightarrow \Phi$.

It is the combination of Property 2 and Property 3 that allows the result of compiling a program unit to be reused.

Finally, each translation step is assumed to be deterministic:

PROPERTY 4 (Translation is deterministic). *If* $E \vdash p \Rightarrow \Phi_1$ *and* $E \vdash p \Rightarrow \Phi_2$ *then* $\Phi_1 = \Phi_2$.

It is this last property that guarantees completeness of the separate compilation framework. Implicitly, translation derivations are forced to be *principal* in the sense that if some derivation $E \vdash p \Rightarrow \Phi_1$ is possible then no other derivation $E \vdash p \Rightarrow \Phi_2$ with Φ_2 "stronger" than Φ_1 is possible [5, 20].

An Example: In-lining of open terms. As an example, we now consider a translation step for inter-module in-lining of small, possibly open, functions. For the example, we are considering the following two intermediate language programs, p_A and p_B for program units `A` and `B`, where `a`, `b`, `f`, and `x` are names, and `ref`, `!`, and `+` are considered built-in constructs:

```
pA = val a = ref 5
     fun f x = x + !a

pB = val b = f 3
```

We now assume that the in-lining translation step for p_A is given by the judgement

$$\{\} \vdash p_\mathrm{A} \Rightarrow \exists\{\}.(E_\mathrm{A}, p'_\mathrm{A}) \tag{1}$$

where

$$E_\mathrm{A} = \{\ \mathtt{a} \mapsto \_\,,\ \mathtt{f} \mapsto \lambda \mathtt{x}.\mathtt{x} + \mathtt{!a}\ \}$$

and $p'_\mathrm{A} = p_\mathrm{A}$ (no in-lining has occurred). Assuming program unit `B` follows immediately after `A`, the in-lining translation step for p_B is given by the judgement

$$E_\mathrm{A} \vdash p_\mathrm{B} \Rightarrow \exists\{\}.(E_\mathrm{B}, p'_\mathrm{B}) \tag{2}$$

where $E_\mathrm{B} = \{\mathtt{b} \mapsto \_\,\}$ and

```
p'B = val b = 3 + !a
```

with the body of the function `f` in-lined. There are two important points to be made here. First, notice that Property 1 (in particular, part 3 and part 4) holds for both (1) and (2). This point demonstrates that the framework is flexible enough to support propagation of open terms (i.e., terms containing free occurrences of names). Second, notice that soundness of the actual in-lining procedure must be established outside of the framework.

3.3 Compilation Bases

For the purpose of composing translation steps to form a notion of compilation, we first define a notion of compilation basis. A *(compilation) basis* B is a sequence $E_1.\cdots.E_n$ of one or more translation environments. We use CompBasis to denote the set of all compilation bases. The translation environment E_i, $1 \leq i \leq n$, in this sequence provides assumptions for the ith translation step in the compiler. The set of names that occur free in B, written $\mathrm{names}(B)$, is the set $\mathrm{names}(E_1) \cup \ldots \cup \mathrm{names}(E_n)$.

Enrichment is extended to bases. A basis $B = E_1.\cdots.E_n$ *enriches* another basis $B' = E'_1.\cdots.E'_n$, written $B \sqsupseteq B'$, if $E_i \sqsupseteq E'_i$ for all $i \in \{1, \cdots, n\}$.

Enrichment on bases is reflexive, transitive, and antisymmetric. These properties follow from the properties of enrichment on environments. It follows that enrichment on bases defines a partial order on bases.

The *restriction* of a basis B to a name set N, written $B \Downarrow N$, is defined inductively by the following equations:

$$\begin{aligned} E \Downarrow N &= E \downarrow N \\ (E.B) \Downarrow N &= (E \downarrow N).(B \Downarrow N') \\ &\quad \text{where } N' = \mathrm{names}(\mathrm{Ran}(E \downarrow N)) \end{aligned} \tag{3}$$

Equation 3 is best illustrated with a small example. Consider the basis $B = E_1.E_2$ composed by the two environments $E_1 = \{a \mapsto t, b \mapsto s, c \mapsto t\}$ and $E_2 = \{t \mapsto l_1, s \mapsto l_2\}$, where a, b, c, s, t, l_1, and l_2 are names. Then the restriction of B to the name set $\{a\}$ is the basis $\{a \mapsto t\}.\{t \mapsto l_1\}$.

We now demonstrate some properties, which describe the relationship between enrichment and restriction.

PROPOSITION 1. *If* $B \sqsupseteq (B' \Downarrow N)$ *and* $N' \subseteq N$ *then* $B \sqsupseteq (B' \Downarrow N')$.

PROOF The proof is by induction on the structure of bases. □

We now show that if some basis B enriches another basis B_0 and if B_0 is identical to B_0 restricted to some name set N then B_0 equals B restricted to N.

PROPOSITION 2. *If* $B \sqsupseteq B_0$ *and* $B_0 = B_0 \Downarrow N$ *then* $B_0 = B \Downarrow N$.

PROOF The proof is by induction on the structure of bases. □

Extension of translation environments ($+$) is lifted to extension of compilation bases by point-wise extension. The *closure* of a compilation basis B with respect to another compilation basis B_0, written $\mathrm{Clos}_{B_0}(B)$, is defined as follows—in case B_0 is "big enough":

$$\mathrm{Clos}_{B_0}(B) = (B_0 + B) \Downarrow \mathrm{Dom}(B)$$

3.4 Compilation

Compilation is defined in terms of translation steps. The rules for compilation allow inferences among sentences of the form

$$B \vdash p \Rightarrow \exists N.(B', p')$$

where B and B' are bases, p is a source program unit, p' is a target program unit, and N is the set of names that are generated during compilation. Sentences of this form are read "p compiles to $\exists N.(B', p')$ in B." Again, the prefix $\exists N$ in the object $\exists N.(B', p')$ binds names and we consider objects of this form equal up to renaming of bound names. We refer to bases in objects of the form $\exists N.(B, p)$ as *export* bases.

Program units $\boxed{B \vdash p \Rightarrow \exists N.(B', p')}$

$$\frac{E \vdash p \Rightarrow \exists N.(E', p')}{E \vdash p \Rightarrow \exists N.(E', p')} \tag{4}$$

$$\frac{E \vdash p \Rightarrow \exists N.(E', p') \quad (N \cup N') \cap \mathrm{names}(E.B) = \emptyset \qquad B \vdash p' \Rightarrow \exists N'.(B', p'') \quad N' \cap \mathrm{names}(E', p') = \emptyset}{E.B \vdash p \Rightarrow \exists (N \uplus N').(E'.B', p'')} \tag{5}$$

The side conditions in (5) ensure that generated names are unique.

The following proposition states that compilation of a program unit depends only on the part of the basis that describes names used in the program unit:

PROPOSITION 3 (Compilation closed under restriction). *If* $B \vdash p \Rightarrow \exists N.(B', p')$ *then there exists* B'' *such that* $B'' = B \Downarrow \mathrm{uses}(p)$ *and* $B'' \vdash p \Rightarrow \exists N.(B', p')$.

PROOF By induction over the structure of bases. The proof makes essential use of Property 1(4), the usage propagation property of translation steps. □

The following proposition states that compilation of a program unit is closed under enrichment of bases:

PROPOSITION 4 (Compilation closed under enrichment). *If* $B \vdash p \Rightarrow \exists N.(B', p')$ *and* $B'' \sqsupseteq B$ *then* $B'' \vdash p \Rightarrow \exists N.(B', p')$.

PROOF The proof is by induction over the structure of bases. □

Finally, due to the property that translation steps are deterministic, compilation is deterministic:

PROPOSITION 5 (Compilation is deterministic). *If* $B \vdash p \Rightarrow \exists N_1.(B_1, p_1)$ *and* $B \vdash p \Rightarrow \exists N_2.(B_2, p_2)$ *then* $\exists N_1.(B_1, p_1) = \exists N_2.(B_2, p_2)$.

PROOF The proof is by induction on the structure of bases. □

4. MLB Compilation

As mentioned, we present MLB (re)compilation management in two steps. In this section, we present a set of MLB compilation inference rules, which closely resembles the MLB static semantics of [12]. Then in Section 5, we present a set of MLB recompilation inference rules, which closely resembles an implementation based on serialization and deserialization [16] for storing and loading bases from appropriate files on a file system.

The essence of MLB compilation management is to convert source code, as provided by an MLB file, into sequences of linkable object code. We use m to range over *sequences of object code*:

$$m \quad ::= \quad c \mid \epsilon \mid m; m$$

We use CodeSeq to denote the set of all possible object code sequences.

When two sequences m_1 and m_2 are put together to form the sequence $m = m_1 \ ; \ m_2$, the sequences m_1 and m_2 are implicitly linked in the sense that used names of m_2 may be bound by declared names of m_1. The set of declared names of m is the union of the declared names of m_1 and m_2. Moreover, the set of used names of m is the union of used names of m_1 and the subtraction of the used names of m_2 with the declared names of m_1. Sequences of object code are considered equal up-to removal of empty object code (ϵ) and associativity of the sequence operator (;).

Further semantic objects used for compilation management include the following:

$$\begin{array}{rcl} \Gamma & \in & \mathrm{ExtCompBasis} = \mathrm{CompBasis} \\ & & \quad \times (\mathrm{BId} \xrightarrow{\mathrm{fin}} \mathrm{ExtCompBasis}) \\ C & \in & \mathrm{MlbCache} = \mathrm{MlbId} \\ & & \quad \xrightarrow{\mathrm{fin}} (\mathrm{ExtCompBasis} \cup \{\textsc{none}\}) \end{array}$$

An *extended compilation basis* (Γ) is a product of a compilation basis and a finite map from basis identifiers to extended compilation bases. An *ML Basis cache* (C) is, essentially, a finite map from MLB file identifiers to extended compilation bases. Intuitively, ML Basis caches are used to guarantee that an ML Basis file is compiled at most once; succeeding references to an MLB file make use of the first compiled instance of the MLB file, by checking if a compiled instance of an MLB file is already present in the ML Basis cache. The ML Basis cache is also used to enforce that there are no cycles in the MLB-file dependency graph.

We sometimes silently inject objects into products when the meaning is obvious from the context. For instance, we often write $\{\}$ to denote the extended compilation basis $(\{\}, \{\})$.

We now present a set of inference rules for specifying MLB compilation. The inference system allows inferences among sentences of the form

$$\Gamma, C \vdash \textit{phrase} \Rightarrow \exists N.(\Gamma', m, C')$$

where *phrase* ranges over either basis expressions or basis declarations. Sentences of the above form are read: "In the extended compilation basis Γ and MLB cache C, *phrase* compiles to an extended compilation basis Γ', link code m, and MLB cache C', with N being a set of newly generated names." We consider *compilation*

results $\exists N.(\Gamma, m, C)$ identical up-to capture-free renaming of the bound names N.

Expressions $\boxed{\Gamma, C \vdash \mathit{bexp} \Rightarrow \exists N.(\Gamma', m, C')}$

$$\frac{\Gamma, C \vdash \mathit{bexp} \Rightarrow \exists N.(\Gamma', m, C')}{\Gamma, C \vdash \texttt{bas}\ \mathit{bdec}\ \texttt{end} \Rightarrow \exists N.(\Gamma', m, C')} \quad (6)$$

$$\frac{\begin{array}{c}\Gamma, C \vdash \mathit{bdec} \Rightarrow \exists N_1.(\Gamma_1, m_1, C_1) \\ \Gamma + \Gamma_1, C + C_1 \vdash \mathit{bexp} \Rightarrow \exists N_2.(\Gamma_2, m_2, C_2) \\ \mathrm{names}(\Gamma, C) \cap N_1 = \emptyset \quad \mathrm{names}(\Gamma, C, \Gamma_1, C_1) \cap N_2 = \emptyset \\ m = m_1; m_2 \quad C' = C_1 + C_2 \quad N = N_1 \uplus N_2\end{array}}{\Gamma, C \vdash \texttt{let}\ \mathit{bdec}\ \texttt{in}\ \mathit{bexp}\ \texttt{end} \Rightarrow \exists N.(\Gamma_2, m, C')} \quad (7)$$

$$\frac{\Gamma(\mathit{longbid}) = \Gamma'}{\Gamma, C \vdash \mathit{longbid} \Rightarrow \exists \emptyset.(\Gamma', \varepsilon, \{\})} \quad (8)$$

Declarations $\boxed{\Gamma, C \vdash \mathit{bdec} \Rightarrow \exists N.(\Gamma', m, C')}$

$$\frac{\begin{array}{c}\Gamma, C \vdash \mathit{bdec}_1 \Rightarrow \exists N_1.(\Gamma_1, m_1, C_1) \\ \Gamma + \Gamma_1, C + C_1 \vdash \mathit{bdec}_2 \Rightarrow \exists N_2.(\Gamma_2, m_2, C_2) \\ \mathrm{names}(\Gamma, C) \cap N_1 = \emptyset \quad \mathrm{names}(\Gamma, C, \Gamma_1, C_1) \cap N_2 = \emptyset \\ C' = C_1 + C_2 \quad N = N_1 \uplus N_2 \quad \Gamma' = \Gamma_1 + \Gamma_2\end{array}}{\Gamma, C \vdash \mathit{bdec}_1\ \mathit{bdec}_2 \Rightarrow \exists N.(\Gamma', m_1; m_2)} \quad (9)$$

$$\frac{}{\Gamma, C \vdash \varepsilon \Rightarrow \exists \emptyset.(\{\}, \varepsilon, \{\})} \quad (10)$$

$$\frac{\begin{array}{c}\Gamma, C \vdash \mathit{bdec}_1 \Rightarrow \exists N_1.(\Gamma_1, m_1, C_1) \\ \Gamma + \Gamma_1, C + C_1 \vdash \mathit{bdec}_2 \Rightarrow \exists N_2.(\Gamma_2, m_2, C_2) \\ \mathrm{names}(\Gamma, C) \cap N_1 = \emptyset \quad \mathrm{names}(\Gamma, C, \Gamma_1, C_1) \cap N_2 = \emptyset \\ C' = C_1 + C_2 \quad N = N_1 \uplus N_2 \quad m = m_1; m_2\end{array}}{\Gamma, C \vdash \texttt{local}\ \mathit{bdec}_1\ \texttt{in}\ \mathit{bdec}_2\ \texttt{end} \Rightarrow \exists N.(\Gamma_2, m, C')} \quad (11)$$

$$\frac{\Gamma, C \vdash \mathit{bexp} \Rightarrow \exists N.(\Gamma', m, C')}{\Gamma, C \vdash \texttt{basis}\ \mathit{bid}\ \texttt{=}\ \mathit{bexp} \Rightarrow \exists N.(\{\mathit{bid} \mapsto \Gamma'\}, m, C')} \quad (12)$$

$$\frac{\Gamma(\mathit{longbid}) = \Gamma'}{\Gamma, C \vdash \texttt{open}\ \mathit{longbid} \Rightarrow \exists \emptyset.(\Gamma', \varepsilon, \{\})} \quad (13)$$

$$\frac{B \vdash p \Rightarrow \exists N.(B', c)}{(B, \_), C \vdash \mathit{uid} \triangleright p \Rightarrow \exists N.(\mathrm{Clos}_B(B'), c, \{\})} \quad (14)$$

$$\frac{C(\mathit{mlbid}) = \Gamma' \quad \Gamma' \neq \textsc{none}}{\Gamma, C \vdash \mathit{mlbid} \triangleright \mathit{bdec} \Rightarrow \exists \emptyset.(\Gamma', \varepsilon, \{\})} \quad (15)$$

$$\frac{\begin{array}{c}\mathit{mlbid} \notin \mathrm{Dom}(C) \\ \{\}, C + \{\mathit{mlbid} \mapsto \textsc{none}\} \vdash \mathit{bdec} \Rightarrow \exists N.(\Gamma', m, C')\end{array}}{\Gamma, C \vdash \mathit{mlbid} \triangleright \mathit{bdec} \Rightarrow \exists N.(\Gamma', m, C' + \{\mathit{mlbid} \mapsto \Gamma'\})} \quad (16)$$

There are several points to be made here.

First, notice the side conditions on the choices of the existentially quantified name sets in rules (7), (9), and (11). These side conditions ensure that names are chosen sufficiently fresh [14, 15, 26].

Second, to motivate the need for $\mathrm{Clos}_B(B')$ in (14), consider a compiler with two translation phases; one that translates source program identifiers into intermediate language names and one that implements a simple constant-propagation phase. Further, consider the following MLB declaration (annotated with file contents):

```
local A.sml ▷ "val a = 5"
in B.sml ▷ "val b = a"
end
C.sml ▷ "val c = b"
```

Compiling `A` (i.e., `A.sml`) may result in a basis $B_1 = \{\texttt{a} \mapsto l\}.\{l \mapsto 5\}$, where l is a freshly generated intermediate language name. Moreover, compiling `B` in the compilation basis B_1 may result in the basis $B_2 = \{\texttt{b} \mapsto l\}.\{\}$. Now, without closing B_2 with respect to B_1, the constant propagation pass will fail to propagate the value of 5 for `b` in `C`. On the other hand, compiling `C` in the basis $\mathrm{Clos}_{B_2}(B_1) = \{\texttt{b} \mapsto l\}.\{l \mapsto 5\}$ can successfully make use of the constant propagation compilation phase. This example illustrates the importance of ensuring that compilation always occurs in closed bases.

Third, notice that the rules for MLB file references ensure that there are no cyclic references to MLB files and that each MLB file is compiled only once, but that the compilation result can be used multiple times by referring to the MLB file multiple times. Moreover, notice that if $\Gamma, C \vdash \mathit{phrase} \Rightarrow \exists N.(\Gamma', m, C')$ is derivable and *phrase* is MLB file free then $\Gamma, \{\} \vdash \mathit{phrase} \Rightarrow \exists N.(\Gamma', m, \{\})$ is derivable. In the following, we shall write $\Gamma \vdash \mathit{phrase} \Rightarrow \exists N.(\Gamma', m)$ to mean $\Gamma, \{\} \vdash \mathit{phrase} \Rightarrow \exists N.(\Gamma', m, \{\})$.

5. MLB Recompilation Management

In this section, we present a set of MLB inference rules for cut-off incremental recompilation, which are straightforward to implement based on techniques for serializing and deserializing bases [16] from appropriate files on a file system.

A *repository* R maps source paths to products (B, p, N, B', c), where B is an *import compilation basis*, p is the source program unit, N is a set of names generated during compilation, B' is an *export compilation basis*, and c is the generated object code:

$$\begin{array}{rcl} R & \in & \mathrm{Rep} = \mathrm{UId} \xrightarrow{\mathrm{fin}} \mathrm{CompBasis} \times \mathrm{SrcProg} \\ & & \quad \times \mathrm{NameSet} \times \mathrm{CompBasis} \times \mathrm{Code} \end{array}$$

Notice that names in N do not alpha-vary.

A repository R is *well-formed* if for all $(B, p, N, B', c) \in \mathrm{Ran}(R)$, we have $B \vdash p \Rightarrow \exists N.(B', c)$ and $B = B \Downarrow (\mathrm{uses}(p))$.

Before we present the inference rules for recompilation management, we define a boolean function for determining if recompilation of a program unit is necessary. The function, named Reuse, is defined as follows:

$$\begin{array}{l} \mathrm{Reuse}(R, B, \mathit{uid}, p) = \\ \quad R(\mathit{uid}) = (B_0, p', N, B', c) \wedge \\ \quad B \sqsupseteq B_0 \wedge \\ \quad p = p' \end{array}$$

The intuition here is that, in contexts where R is known to be well-formed, the function Reuse can be used to determine if a judgement from the repository may be used instead of deriving a new judgement.

Source lists (ranged over by L) and dependency maps (ranged over by D and M) are defined as follows:

$$\begin{array}{rcl} L & \in & \mathrm{SourceList} = \mathrm{UId}^{(k)} \\ M & \in & \mathrm{BidDepMap} = \mathrm{BId} \xrightarrow{\mathrm{fin}} \mathrm{DepMap} \\ D & \in & \mathrm{DepMap} = \mathrm{SourceList} \times \mathrm{BidDepMap} \end{array}$$

When $D = (L, M)$ and $D' = (L', M')$, we define $D \oplus D'$ to mean $(L@L', M + M')$, where @ denotes list concatenation. Associativity of $\oplus$ follows from associativity of @ and $+$.

We now present a set of inference rules for MLB recompilation management. The inference system allows inferences among

sentences of the form

$$R_0, R, D \vdash \mathit{phrase} \Rightarrow D', \exists N.(m, R')$$

where *phrase* ranges over either basis expressions or basis declarations. Sentences of the above form are read: "Under the repository assumptions R_0, R, and dependency map D, *phrase* compiles to a new dependency map D', repository R', and link code m, with N being a set of newly generated names." Whereas the repository assumption R_0 serves as the repository from which compilation judgements can be reused during recompilation management, R and R' serve to accumulate new repository judgements to be used for future recompilations.

Notice, that in objects of the form $\exists N.(m, R)$, the name set N binds names and two such objects are considered identical up-to capture free renaming of bound names.

Also notice that the link code m in the judgements are used to relate the MLB recompilation semantics to the simpler MLB compilation semantics from Section 4.

Expressions $\boxed{R_0, R, D \vdash \mathit{bexp} \Rightarrow D', \exists N.(m, R')}$

$$\frac{R_0, R, D \vdash \mathit{bexp} \Rightarrow D', \exists N.(m, R')}{R_0, R, D \vdash \texttt{bas}\ \mathit{bdec}\ \texttt{end} \Rightarrow D', \exists N.(m, R')} \tag{17}$$

$$\frac{\begin{array}{c}R_0, R, D \vdash \mathit{bdec} \Rightarrow D_1, \exists N_1.(m_1, R_1) \\ R_0, R + R_1, D \oplus D_1 \vdash \mathit{bexp} \Rightarrow D_2, \exists N_2.(m_2, R_2) \\ N = N_1 \uplus N_2 \quad m = m_1; m_2 \quad R' = R_1 + R_2 \\ N_1 \cap \mathrm{names}(R) = \emptyset \quad N_2 \cap \mathrm{names}(R, R_1) = \emptyset\end{array}}{R_0, R, D \vdash \texttt{let}\ \mathit{bdec}\ \texttt{in}\ \mathit{bexp}\ \texttt{end} \Rightarrow D_2, \exists N.(m, R')} \tag{18}$$

$$\frac{D(\mathit{longbid}) = D'}{R_0, R, D \vdash \mathit{longbid} \Rightarrow D', \exists \emptyset.(\varepsilon, \{\})} \tag{19}$$

Declarations $\boxed{R_0, R, D \vdash \mathit{bdec} \Rightarrow D', \exists N.(m, R')}$

$$\frac{\begin{array}{c}R_0, R, D \vdash \mathit{bdec}_1 \Rightarrow D_1, \exists N_1.(m_1, R_1) \\ R_0, R + R_1, D \oplus D_1 \vdash \mathit{bdec}_2 \Rightarrow D_2, \exists N_2.(m_2, R_2) \\ N = N_1 \uplus N_2 \quad R' = R_1 + R_2 \quad D' = D_1 \oplus D_2 \\ N_1 \cap \mathrm{names}(R) = \emptyset \quad N_2 \cap \mathrm{names}(R, R_1) = \emptyset\end{array}}{R_0, R, D \vdash \mathit{bdec}_1\ \mathit{bdec}_2 \Rightarrow D', \exists N.(m_1; m_2, R')} \tag{20}$$

$$\frac{}{R_0, R, D \vdash \varepsilon \Rightarrow ([], \{\}), \exists \emptyset.(\varepsilon, \{\})} \tag{21}$$

$$\frac{\begin{array}{c}R_0, R, D \vdash \mathit{bdec}_1 \Rightarrow D_1, \exists N_1.(m_1, R_1) \\ R_0, R + R_1, D \oplus D_1 \vdash \mathit{bdec}_2 \Rightarrow D_2, \exists N_2.(m_2, R_2) \\ N = N_1 \uplus N_2 \quad m = m_1; m_2 \quad R' = R_1 + R_2 \\ N_1 \cap \mathrm{names}(R) = \emptyset \quad N_2 \cap \mathrm{names}(R, R_1) = \emptyset\end{array}}{R_0, R, D \vdash \texttt{local}\ \mathit{bdec}_1\ \texttt{in}\ \mathit{bdec}_2\ \texttt{end} \Rightarrow D_2, \exists N.(m, R')} \tag{22}$$

$$\frac{\begin{array}{c}R_0, R, D \vdash \mathit{bexp} \Rightarrow D', \exists N.(m, R') \\ D'' = \{\mathit{bid} \mapsto D'\}\end{array}}{R_0, R, D \vdash \texttt{basis}\ \mathit{bid}\ \texttt{=}\ \mathit{bexp} \Rightarrow D'', \exists N.(m, R')} \tag{23}$$

$$\frac{D(\mathit{longbid}) = D'}{R_0, R, D \vdash \texttt{open}\ \mathit{longbid} \Rightarrow D', \exists \emptyset.(\varepsilon, \{\})} \tag{24}$$

$$\frac{\begin{array}{c}\neg\mathrm{Reuse}(R_0, B, \mathit{uid}, p) \quad L \text{ of } D = [\mathit{uid}_1, \cdots, \mathit{uid}_n] \\ R(\mathit{uid}_i) = (B_i, p_i, N_i, B_i', c_i)\ \ i = [1..n] \\ B = B_1' + \cdots + B_n' \quad B_0 = B \Downarrow \mathrm{uses}(p) \\ B \vdash p \Rightarrow \exists N.(B', c) \quad B'' = \mathrm{Clos}_B(B') \\ R' = \{\mathit{uid} \mapsto (B_0, p, N, B'', c)\} \quad N \cap \mathrm{names}(R) = \emptyset\end{array}}{R_0, R, D \vdash \mathit{uid} \triangleright p \Rightarrow [\mathit{uid}], \exists N.(c, R')} \tag{25}$$

$$\frac{\begin{array}{c}\mathrm{Reuse}(R_0, B, \mathit{uid}, p) \quad L \text{ of } D = [\mathit{uid}_1, \cdots, \mathit{uid}_n] \\ R(\mathit{uid}_i) = (B_i, p_i, N_i, B_i', c_i)\ \ i = [1..n] \\ B = B_1' + \cdots + B_n' \quad N \cap \mathrm{names}(R) = \emptyset \\ R_0(\mathit{uid}) = (B_0, p', N, B', c) \quad R' = \{\mathit{uid} \mapsto R_0(\mathit{uid})\}\end{array}}{R_0, R, D \vdash \mathit{uid} \triangleright p \Rightarrow [\mathit{uid}], \exists N.(c, R')} \tag{26}$$

Notice how the rules propagate program unit dependencies in source lists and dependency maps. Dependency information is used to build compilation bases in rules for compilation (25) and reuse (26). In these rules, the current source list is used for determining which compilation bases should be extracted from the repository, so as to form the compilation basis to be used for checking if recompilation is necessary (and for compilation in case rule (26) does not apply).

Rule (26) allows for a compilation result in the repository for *uid* to be reused if certain side conditions hold. First, the program unit must not have changed since the result was stored in the repository. This requirement is expressed in the rule with the side condition $p = p'$; in an implementation, file modification dates or cryptographic check-sums may be used to check for this requirement. Second, the basis constructed from the dependency information must enrich the import basis of the repository entry for the program unit. Finally, the generated names of the object found in the repository must be fresh with respect to the repository R.

Rule (25) corresponds to compilation. If the side condition in this rule is satisfied then there is no object in the repository that can be reused; thus, the program unit must be (re)compiled. It is never the case that both rule (26) and rule (25) are applicable, given R_0, R, D, and *bdec*.

The rules for recompilation are non-deterministic because the choice of the name set N_1 in rule (20), for instance, has influence on whether rule (26) is applicable for program units in bdec_2. This non-determinism, however, has no influence on the correctness result that we shall demonstrate. However, the flexibility in the rules is exactly what allows for cut-off recompilation. We shall return to this issue in Section 5.2.

We note here that the implementation of the recompilation framework in the MLKit makes a few optimizations to make it cheaper to check whether recompilation is necessary than first loading all dependent compilation bases; we will come back to this implementation aspect in Section 6.

5.1 Properties of the Recompilation Semantics

We now present some properties of the MLB recompilation semantics.

We have earlier argued (in Section 4) for the importance of closing compilation bases. We say that a compilation basis B is *closed* if $B = \mathrm{Clos}_{\{\}}(B)$. Moreover, we say that a repository R is *closed* if for all $(B, p, N, B', c) \in \mathrm{Ran}(R)$, the bases B and B' are closed. The following proposition states that compilation results in closed repositories:

PROPOSITION 6 (Compilation Closedness). *Assume R_0 and R closed. If $R_0, R, D \vdash \mathit{phrase} \Rightarrow D', \exists N.(c, R')$ then R' closed.*

PROOF The proof is a simple inductive argument on the structure of *phrase*, making use of the properties that if B and B' are closed then so are $B + B'$ and $B \Downarrow N'$, where $N' \subseteq \mathrm{Dom}(B)$. □

Well-formedness of repositories is preserved by compilation, as stated by the following proposition:

PROPOSITION 7 (Preservation of Well-formedness). *Assume R_0 and R well-formed. If $R_0, R, D \vdash \mathit{phrase} \Rightarrow D', \exists N.(m, R')$ then R' is well-formed.*

Also the proof of Proposition 7 follows a simple inductive argument.

Correctness of the recompilation framework, in the sense that the result of compilation is independent of whether repository judgements have been used, is expressed by the following theorem:

THEOREM 1 (Recompilation Correctness). *Assume R_1, R, and R_2 are well-formed. If $R_1, R, D \vdash$ phrase $\Rightarrow D', \exists N.(m, R')$ then $R_2, R, D \vdash$ phrase $\Rightarrow D', \exists N.(m, R')$.*

PROOF (Sketch) The proof follows a simple inductive argument and makes use of Proposition 3, Proposition 4 and Proposition 5. □

The following two corollaries follow immediately from Theorem 1:

COROLLARY 1 (Recompilation Soundness). *Assume R_1 is well-formed. If $R_1, \{\}, [] \vdash$ phrase $\Rightarrow D', \exists N.(m, R')$ then $\{\}, \{\}, [] \vdash$ phrase $\Rightarrow D', \exists N.(m, R')$.*

COROLLARY 2 (Recompilation Completeness). *Assume R_1 is well-formed. If $\{\}, \{\}, [] \vdash$ phrase $\Rightarrow D', \exists N.(m, R)$ then $R_1, \{\}, [] \vdash$ phrase $\Rightarrow D', \exists N.(m, R)$.*

Corollary 1 expresses that the same compilation result as one obtained from compiling in an arbitrary well-formed repository can be obtained by compiling "from scratch" in an empty repository. Corollary 2 expresses that instead of compiling in the empty repository, the same result can be obtained by compiling in any well-formed repository.

5.2 Non-Determinism and Matching

As mentioned earlier, the rules for MLB recompilation are non-deterministic in the sense that the choice of the name set N_1 in rule (20), for instance, may influence whether rule (26) is applicable for program units in $bdec_2$.

Almost all non-determinism may be eliminated by always generating fresh names during compilation and by allowing renaming of bound names only in rule (25) when the program unit p is compiled to the object $\exists N.(B', c)$. At this point, if an object is available in the repository for the program unit identifier *uid* and B is the associated export basis in the repository, the goal is to choose the name set N such that the basis B' agrees, on as many entries as possible, with B. In an implementation, the process of renaming N can be done by *matching* the basis B' to agree with the basis B on as many entries as possible. Now, the reason it is not possible to eliminate the non-determinism completely is that different choices of the name set N can satisfy agreement for B and B' for different entries and thus may result in different repository objects being reused. As a simple example, assume a compilation basis is an environment that maps program variables, ranged over by `a` and `b`, to machine code labels, ranged over by l. Further, assume $B = \{\mathtt{a} \mapsto l_1, \mathtt{b} \mapsto l_2\}$, for some distinct machine code labels l_1 and l_2, and assume $\exists N.(B', c) = \exists\{l\}.(\{\mathtt{a} \mapsto l, \mathtt{b} \mapsto l\}, c)$, for some c. There are now two possibilities for the choice of N, each of which satisfy agreement for either `a` or for `b`, exclusively.

5.3 Relating the Compilation and Recompilation Semantics

For relating the MLB compilation semantics with the more complicated MLB recompilation semantics, we first define a binary *consistency* relation $\Gamma \,||_D\, R$ between an extended compilation basis Γ on one side and a pair of a dependency map D and a repository R on the other side. The consistency relation is defined inductively over the structure of Γ and D as follows:

Basis Consistency $\boxed{\Gamma \,||_D\, R}$

$$\frac{\begin{array}{c} L = [\mathit{uid}_1, \cdots, \mathit{uid}_n] \\ R(\mathit{uid}_i) = (B_i, p_i, N_i, B_i', c_i) \quad i = 1..n \\ B = B_1' + \cdots + B_n' \quad \mathrm{Dom}(\mathit{BB}) = \mathrm{Dom}(M) \\ \forall \mathit{bid} \in \mathrm{Dom}(\mathit{BB}).\mathit{BB}(\mathit{bid}) \,||_{(M(\mathit{bid}))}\, R \end{array}}{(B, \mathit{BB}) \,||_{(L,M)}\, R} \tag{27}$$

The following proposition states the compositional property of consistency:

PROPOSITION 8 (Consistency Merging). *If $\Gamma \,||_D\, R$ and $\Gamma' \,||_{D'}\, R'$ and $\mathrm{Dom}(R) \cap \mathrm{Dom}(R') = \emptyset$ then $\Gamma + \Gamma' \,||_{D \oplus D'}\, R + R'$.*

PROOF (Sketch) By induction over the structure of Γ using $\Gamma' = \{\}$ and $D' = []$ at the inductive step to get $\Gamma \,||_D\, R + R'$ before establishing the conclusion from the definition of consistency. □

The following proposition states that consistency is preserved under lookup of long basis identifiers:

PROPOSITION 9 (Consistency Lookup). *If $\Gamma \,||_D\, R$ and $\Gamma' = \Gamma(\mathit{longbid})$ then $\Gamma' \,||_{D'}\, R$, where $D' = D(\mathit{longbid})$.*

PROOF By induction over the structure of *longbid*. □

We can now state a theorem saying that the MLB compilation semantics is equivalent to the MLB recompilation management semantics:[1]

THEOREM 2 (Semantics equivalence). *If $\Gamma \,||_D\, R$ and $\Gamma \vdash$ phrase $\Rightarrow \exists N.(\Gamma', m)$ then $\{\}, R, D \vdash$ phrase $\Rightarrow D', \exists N.(m, R')$ and $\Gamma' \,||_{D'}\, R + R'$.*

PROOF By induction over the structure of *phrase*. See Appendix A for a detailed proof. □

6. MLB Recompilation in the MLKit

The MLKit [31, 32] is a Standard ML compiler, which allows for type and compilation information to migrate across module boundaries at compile time [14, 15] using the framework presented in this paper. Compilation is defined as the composition of a series of translation phases, which includes type inference, various optimizing translations [6], elimination of polymorphic equality [13], region inference [30, 33], various region representation analyses [7], closure conversion, instruction selection, and register allocation. Many of the translation phases make use of the possibility of passing information across compilation unit boundaries. For instance, region inference is a type-based analysis, which associates function identifiers with so called region type schemes that provide information about in which regions arguments to the function should be stored and in which regions the result of the function is stored. Because region inference tracks the effects (represented as graphs) of calling the function, region type schemes can become large compared to the underlying ML type schemes, which also leads to large compilation bases.

For the MLKit instance of the framework, elaboration information typically amounts to only five percent of the total size of compilation bases, thus, some overhead must be expected compared to compilers that propagate only elaboration information across program unit boundaries. Yet, the MLKit instance of the framework is used extensively in practice both for compiling the MLKit itself and in the context of SMLserver [17, 18], a platform for programming Web applications in Standard ML. SMLserver is used as the

[1] Notice that we do not consider objects of the form $\exists N.(\Gamma, m)$ equal up-to removal of names in N that do not appear in Γ and m.

development and production platform for a variety of administrative Web applications (more than 250.000 lines of Standard ML) at the IT University of Copenhagen, including a course evaluation system, online course registration and course administration systems, and a human resource system. For all the systems, MLB files are used for organizing and grouping source files and common libraries.

In the remainder of this section, we will comment on some issues we have found important in relation to the practical implementation of the framework in the MLKit.

6.1 Serialization and Deserialization of Compilation Bases

The MLB recompilation semantics presented in Section 5 allows for a straightforward implementation of the framework with repository information (compilation bases and target code) stored on disk. For storing compilation bases on disk, the MLKit makes use of a serialization library, written in Standard ML [16]. Efficiency of serialization and deserialization as well as efficiency of summing compilation bases ($+$) turn out to be critical. On a 2.8GHz Intel Pentium 4 Linux box with 512Mb of RAM, measurements show that serialization of all compilation bases for the MLKit implementation of the Standard ML Basis Library takes 14.2 seconds and amounts to 1.88Mb of data. The data includes, for instance, region type schemes for all visible identifiers and inline code for small functions propagated by the MLKit inter-module optimizer.

Deserialization and summing of all compilation bases for the library takes only 4.0 seconds, which, however, is still too costly for ordinary use; notice that almost all files in a programming project depends on the Standard ML Basis Library. In Section 6.3, we describe a technique for minimizing further the deserialization of compilation bases. The implementation uses Patricia trees [25, 23] for representing translation environments, which leads to close to constant-time summing of deserialized compilation bases.

Compilation bases and target code are stored on disk in a directory `MLB` located in the same directory as the program unit. Thus, deleting the `MLB` directory effectively resets the repository (although, as we have shown, this should not be necessary.)

Matching in the MLKit is composed from matching functions for each translation step in the compiler. In the case that matching results in the resulting compilation basis to be identical to the compilation basis for the unit in the repository, the MLKit avoids the serialization of the compilation basis.

6.2 A More Liberal Dependency Analysis

The dependency analysis integrated with the MLB recompilation semantics in Section 5 assumes that each source file identifier appears at most once in the MLB files for an entire program. The MLKit implements a more liberal dependency analysis, which allows for two different program units with the same name to appear in different MLB files.

For making unique names, the MLKit makes use of the above requirement by assuming that for each program unit *uid*, the pair $(\mathit{uid}, \mathit{mlbid})$ is unique, where *mlbid* denotes the MLB file referring to the program unit. In effect, one can think of repositories not to be indexed by program units, but instead to be indexed by pairs of a program unit and an MLB file.

6.3 Separating Elaboration and Compilation Bases

Although the dependency analysis integrated with the MLB recompilation semantics in Section 5 limits the number of bases to be deserialized upon compilation of a program unit, it is still possible, using a source code dependency analysis, to decrease the number of bases to be deserialized.

Unfortunately, source code dependency analysis for Standard ML is possible only by integrating the dependency analysis with a kind of elaboration so as to determine which identifiers have constructor status [22]. The reason for this is that an occurrence of an identifier with constructor status in a pattern should not be considered a binding occurrence of the identifier. To be sound with respect to the semantics of Standard ML, the MLKit takes the approach of storing the elaboration part of a compilation basis separately from the compilation basis proper. Whereas it is still necessary to deserialize and sum all elaboration bases corresponding to the program units the particular compilation unit depends on, compilation bases need only be deserialized for those program units that the compilation unit refers to.

For the MLKit implementation, elaboration bases typically amount to only five percent of the size of the corresponding compilation basis. In effect, deserializing all dependent elaboration bases, running the source code dependency analysis on the compilation unit, and deserializing only the compilation bases that the compilation unit refers to amounts to less than one second for each source file of a typical large program (such as the MLKit itself).

6.4 Using Time Stamps to Avoid Unnecessary Deserialization

The MLB recompilation framework implemented in the MLKit makes use of time stamps to avoid unnecessary deserialization of bases, when possible. Consider rule (26). If all files containing the bases B'_i, $i = 1..n$, are older than the file containing the target code c then reuse is immediately possible (without deserializing bases), provided the source code is unchanged and the dependency information is unchanged. Only when time stamps cannot be used to conclude that reuse is safe are bases deserialized and strong enrichment used to check for the possibility of reuse.

The above optimization is enhanced by the following test: If, after compiling a program unit, matching results in the export compilation basis to be identical to the export compilation basis for the program unit in the repository then serialization of the compilation basis is avoided, which leaves intact the time stamp of the file containing the export compilation basis.

7. Conclusion and Future Work

We have presented a framework for cut-off incremental recompilation and shown that recompilation in this framework, with respect to so-called well-formed repositories, is both sound and complete. The framework is based on particular abstract properties of individual translation phases of a compiler and supports even open terms (objects containing free occurrences of names) to propagate across program unit boundaries.

There are several possibilities for future work. First, improving the efficiency of the underlying serialization techniques would further improve the overall compilation and recompilation times. Second, it could make sense to explore the possibilities for extending the MLB language with functional features.

References

[1] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[2] Ravi Sethi Alfred V. Aho and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison Wesley, 1986.

[3] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Even more principal typings for java-like languages. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2004)*, June 2004. Oslo, Norway.

[4] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *32nd ACM Symposium on Principles of Programming Languages (POPL'05)*, pages 26–37. ACM Press, January 2005.

[5] D. Ancona and E. Zucca. Principal typings for java-like languages. In *31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 306–317. ACM Press, January 2004.

[6] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[7] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 171–183. ACM Press, January 1996.

[8] Matthias Blume. CM, a compilation manager for SML/NJ. Technical report, Princeton University, Department of Computer Science, April 1995. User Manual.

[9] Matthias Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, Department of Computer Science, November 1997.

[10] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4), July 1999.

[11] Luca Cardelli. Program fragments, linking, and modularization. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, January 1997.

[12] Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. Formal specification of the ML Basis System, January 2005. Available from `http://www.mlton.org`.

[13] Martin Elsman. Polymorphic equality—no tags required. In *Second International Workshop on Types in Compilation (TIC'98)*, March 1998.

[14] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, Department of Computer Science, University of Copenhagen, January 1999.

[15] Martin Elsman. Static interpretation of modules. In *Procedings of Fourth International Conference on Functional Programming (ICFP'99)*, pages 208–219. ACM Press, September 1999.

[16] Martin Elsman. Type-specialized serialization with sharing. In *Sixth Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

[17] Martin Elsman and Niels Hallenberg. Web programming with SMLserver. In *International Symposium on Practical Aspects of Declarative Languages (PADL'03)*. Springer-Verlag, January 2003.

[18] Martin Elsman and Ken Friis Larsen. Typing XHTML Web applications in ML. In *International Symposium on Practical Aspects of Declarative Languages (PADL'04)*. Springer-Verlag, June 2004.

[19] S. Feldman. Make—a computer program for maintaining computer programs. *Software-Practice and Experience*, 9(4):255–265, April 1979.

[20] Trevor Jim. What are principal typings and what are they good for? In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 42–53. ACM Press, January 1996.

[21] Henning Makholm. *Mosmake*, 2002. Available from the Web page `http://www.diku.dk/~makholm/mosmake/`.

[22] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[23] Donald R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[24] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In *34th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'07)*, pages 143–154. ACM Press, January 2007.

[25] C. Okasaki and A. Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[26] Claudio V. Russo. *Types for Modules*. PhD thesis, University of Edinburgh, June 1998.

[27] Zhong Shao and Andrew Appel. Smartest recompilation. In *20th ACM Symposium on Principles of Programming Languages*, January 1993.

[28] David Swasey, Tom Murphy VII, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML. In *ACM SIGPLAN Workshop on ML (ML'06)*, September 2006.

[29] Walter Tichy. Smart recompilation. In *ACM Transactions on Programming Languages and Systems*, pages 273–291, July 1986.

[30] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998. (plus 24 pages of electronic appendix).

[31] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation (HOSC)*, 17(3):245–265, September 2004.

[32] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical Report TR-2001-07, IT University of Copenhagen, October 2001.

[33] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[34] Spyridon Triantafyllis, Matthew J. Bridges, Easwaran Raman, Guilherme Ottoni, and David I. August. A framework for unrestricted whole-program optimization. In *ACM Conference on Programming Language Design and Implementation (PLDI'06)*, pages 61–71. ACM Press, January 2006.

[35] Stephen Weeks. Whole-program compilation in MLton, September 2006. Talk at ML Workshop, 2006. Slides available from http://mlton.org/.

A. Proof of Theorem 2 [Semantic Equivalence]

The proof is by induction on the structure of *phrase*. We proceed by case analysis.

CASE $phrase = longbid$ From assumptions and (8), we have $\Gamma(longbid) = \Gamma'$, thus from Proposition 9, we have $D(longbid) = D'$ and $\Gamma' \,||_{D'}\, R$. Thus, from (19), we have $\{\}, R, D \vdash longbid \Rightarrow D', \exists\emptyset.(\varepsilon, R')$ and $\Gamma' \,||_{D'}\, (R + R')$, as required with $R' = \{\}$.

CASE $phrase = uid \triangleright p$ From assumptions we have [1] $B = B$ of Γ and [2] $B \vdash p \Rightarrow \exists N.(B', c)$ and [3] $B'' = \mathrm{Clos}_B(B')$ and [4] $\Gamma' = (B'', \{\})$. Let [5] $D' = ([uid], \{\})$. From assumptions we have [6] $\Gamma \,||_D\, R$, thus from the definition of consistency, we have [7] L of $D = [uid_1, \cdots, uid_n]$ and [8] $R(uid) = (B_i, p_i, N_i, B'_i, c_i)$, $i = 1..n$ and [9] $B = B'_1 + \cdots B'_n$. From Proposition 3 and [2], we have there exists B_0 such that [10] $B_0 = B \Downarrow \mathrm{uses}(p)$. By appropriate renaming, we can assume [11] $N \cap \mathrm{names}(R) = \emptyset$. Let [12] $R' = \{uid \mapsto (B_0, p, N, B'', c)\}$. From rule (25) and [2,3,5,7,8,9,10,11,12], we have [13] $\{\}, R, D \vdash uid \triangleright p \Rightarrow D', \exists N.(c, R')$, as required. Moreover, from [4,5,12] and the definition of consistency, we have $\Gamma' \,||_{D'}\, (R + R')$, also as required.

CASE $phrase = bdec_1\ bdec_2$ From assumptions we have [1] $N = N_1 \uplus N_2$ and [2] $N_1 \cap \mathrm{names}(\Gamma) = \emptyset$ and [3] $N_2 \cap \mathrm{names}(\Gamma, \Gamma_1) = \emptyset$ and [4] $\Gamma \vdash bdec_1 \Rightarrow \exists N_1.(\Gamma_1, m_1)$ and [5] $\Gamma + \Gamma_1 \vdash bdec_2 \Rightarrow \exists N_2.(\Gamma_2, m_2)$ and [6] $m = m_1; m_2$ and [7] $\Gamma' = \Gamma_1 + \Gamma_2$. By immediate induction using [4] and assumptions, we have [8] $\{\}, R, D \vdash bdec_1 \Rightarrow D_1, \exists N_1.(m_1, R_1)$ and [9] $\Gamma_1 \,||_{D_1}\, (R + R_1)$. Using Proposition 8, [9], and assumptions, we have [11] $\Gamma + \Gamma_1 \,||_{D \oplus D_1}\, (R + R_1)$. By induction using [5] and [11], we have [12] $\{\}, R + R_1, D \oplus D_1 \vdash bdec_2 \Rightarrow D_2, \exists N_2.(m_2, R_2)$ and [13] $\Gamma_2 \,||_{D_2}\, (R + R_1 + R_2)$. By Proposition 8 and [9,13,7], we have [15] $\Gamma' \,||_{D_1 \oplus D_2}\, (R + R_1 + R_2)$, as required with [16] $D' = D_1 \oplus D_2$ and [17] $R' = R_1 + R_2$. By appropriate renaming,

we can assume (due to [2] and [3]) [18] $N_1 \cap \mathrm{names}(R) = \emptyset$ and [19] $N_2 \cap \mathrm{names}(R, R_1) = \emptyset$. It follows that we can apply rule (20) to [6,8,12,16,17,18,19] to get $\{\}, R, D \vdash \mathit{bdec}_1\ \mathit{bdec}_2 \Rightarrow D', \exists N.(m, R')$, also as required.

CASE $\mathit{phrase} = \texttt{local}\ \mathit{bdec}_1\ \texttt{in}\ \mathit{bdec}_2\ \texttt{end}$ The proof of this case is similar to the proof for the case $\mathit{phrase} = \mathit{bdec}_1\ \mathit{bdec}_2$, but instead of [7], we have [7a] $\Gamma' = \Gamma_2$. Moreover, we need not establish [15] using Proposition 8. Instead, [13] suffices and with [7a] and [16a] $D' = D_2$ (instead of [16]), we have $\Gamma' \,||_{D'}\, (R + R_1 + R_2)$, as required. It also follows that we can apply rule (22) using [6,8,12,16a,17,18,19] to get $\{\}, R, D \vdash \mathit{phrase} \Rightarrow D', \exists N.(m, R')$, also as required.

The remaining cases follow similarly. □