Smart Recompilation in Glasgow Haskell

Patrick Sansom
University of Glasgow

Abstract

This paper describes the system of interface files and recompilation checking in the Glasgow Haskell 1.3 Compiler. In developing this system our aim is to provide a separate compilation system which avoids unnecessary recompilation while performing significant inter-module optimisations. We associate a *version number* (based on the current timestamp) with each declaration and record the versions of all the imported declarations *used* when a module is compiled. When the module is recompiled the *usage numbers* are compared with the current version numbers to determine if the recompilation is unnecessary.

The paper also describes a proposed system of *source interfaces* designed to deal with the problem of cyclic module dependencies. This avoids the error-prone duplication of declarations by extracting the interface directly from the source code.

1 Introduction

Selective recompilation is an important technique for reducing the amount of recompilation required after a change has been made in a large software system (Adams, Tichy & Weinert [1994]). This is particularly important in a compilation system which performs significant inter-module optimisations since these optimisations require additional information to be exposed to the importing module.

This paper describes the system of interface files and recompilation checking in the Glasgow Haskell 1.3 Compiler — a state-of-the-art optimising compiler for Haskell, Version 1.3 (Peterson et al. [1996]), which performs significant inter-module optimisations. In developing this system our aim is to provide:

1. a scheme for reducing the amount of unnecessary recompilation,
2. a straightforward means of communicating all the required information between modules, and
3. a mechanism for handling cyclic module dependencies.

Before presenting our compilation system we discuss some of the problems encountered with the Haskell 1.2 module system (Hudak et al. [1992]). These provided the motivation for the development of our compilation system which went hand-in-hand with the adoption of a less specific module system in Haskell 1.3.

1.1 Motivation

Inter-module optimisations require additional pragmatic information to be exposed in a module's interface. For example, consider the module

```
module Decls (Decl(..), Expr, substDecl) where
import Id (Id)
import Expr (Expr, substExpr)
type Decl = (Id, Expr)
substDecl :: (Id -> Id) -> Decl -> Decl
substDecl fn decl = case decl of (id, expr) -> (fn id, substExpr fn expr)
```

from which the following Haskell 1.2 interface is generated.

```
interface Decls where
import Expr(Expr)
import Id(Id)
data Expr
data Id
type Decl = (Id, Expr)
substDecl :: (Id -> Id) -> Decl -> Decl
```

The inter-module optimisations might expose `Expr`'s constructors and an unfolding for `substDefn`. These were typically incorporated in the standard interface file as compiler recognisable "comments".

```
interface Decls where
import Expr(Expr,substExpr)
import Id(Id)
data Expr
  {-# CONSTRS Var Id | App Expr Expr | ... #-}
data Id
type Decl = (Id, Expr)
substDecl :: (Id -> Id) -> Decl -> Decl
  {-# UNFOLD \ f d -> case d of (i,e) -> (f i, substExpr f e) #-}
```

This has number of significant implications. Firstly, the pragmatic information may change when the implementation of a module is modified, even if these modifications do not change the standard interface. This can result in a significant amount of recompilation since a module is recompiled if any of the imported interface files changes. It is often the case that many of the ensuing recompilations are unnecessary as the particular definitions which are used by the module being recompiled may not have changed. This is not a new problem — changes to standard interfaces also cause unnecessary recompilation. However, the presence of pragmatic information makes this problem much worse. The effect on overall turnaround time is further exacerbated by the increase in compilation times caused by the increased size of the interface files and the additional time spent performing the inter-module optimisations.

Secondly, Haskell 1.2 requires that all the entities referred to by the imported declarations are also imported (Hudak et al. [1992, section 5.1.3]). For example, a module which imports the function `substDecl` must also import all the entities mentioned by the interface information for `substDecl`. Thus `Decl(..)` (its a type synonym) must be imported which in turn requires `Expr` and `Id` to be imported. For standard interfaces this closure operation is manageable since it only requires additional type and class information to be imported. Indeed, an interface can be made self-sufficient by ensuring that it exports all the types and classes it mentions.

However, this restriction causes problems when inter-module pragmas are introduced. For example, the unfolding for `substDecl` mentions `substExpr`. A module which imports `substDecl` can only make use of the unfolding if the definition of `substExpr` is also imported. The situation is even worse if an unfolding happens to refer to a local definition, i.e. a definition which is not exported, since this definition cannot be imported. This is all very unsatisfactory — either the programmer has to embark on the cumbersome task of importing (and exporting) all the additional declarations required by the optimisation pragmas or they must accept a reduced level of inter-module optimisation.

Another problem associated with the Haskell 1.2 module system arises from the fact that an interface must contain the definitions of *all* the declarations exported, even when they are actually defined in another module. This results in a lot of information being duplicated in different interface files, especially if a large body of pragmatic information is exported with each declaration. When the imported interfaces are read all this duplicate information has to be read and checked for consistency.[1]

Acknowledging these problems the Haskell 1.3 language definition does not attempt to define a standard interface format — each implementation is left free to define its own interface conventions. The sole purpose of the import and export declarations is to control the names which are brought into scope, i.e. those names which can be explicitly mentioned in the source of the module. The compilation system is responsible for finding any additional information

[1] While the duplicate definitions are normally consistent, cycles in the module structure can result in inconsistencies while the new definition of a modified declaration is being propagated.

needed for compilation without the help of the programmer (Peterson et al. [1996, section 5.2]). (We use the term *implicit* imports to refer to these additional imports.)

1.2 Overview

The Glasgow Haskell 1.3 compilation system consists of four main components:

Recompilation Checker: A *version number* is associated with every top-level declaration. When a module is compiled the versions of all the imported declarations which were actually *used* during the compilation are recorded. When the module is subsequently recompiled the *usage numbers* recorded during the previous compilation are compared with the current version numbers to determine if recompilation is actually required (see Section 2).

Interface Files: As well as recording the *exported* interface of a module the interface file records all the version numbers and inter-module information for every *exposed* top-level declaration which was defined in the module, i.e. any declaration which the compiler may need to know about when compiling another module (see Section 3).

Source Interfaces: Haskell allows cyclic module dependencies which pose particular problems to the compilation system. Our solution introduces the idea of a *source interface* (see Section 4).

Interface Processing: During compilation the compiler must read all the required information out of the interface files. This process involves a transitive closure operation since the information for a particular declaration may refer to additional declarations which were not imported by the source (see Section 5).

After describing each of these components I briefly discuss related work (Section 6) and present our plans to evaluate the effectiveness of the recompilation checker (Section 7).

2 The Recompilation Checker

The basic idea is to avoid recompiling a module if the source has not been changed and the interface of the imported declarations used when the module was last compiled have not changed. This corresponds to asking the question: “Are the inputs to the compilation of this module identical to the inputs to the previous compilation of this module?”. A system of version numbers is used to identify when an interface or declaration has changed.

2.1 Version numbers

Each module and each declaration within a module has associated with it a *version number*. These are recorded in the module’s interface file (see Section 3). The version numbers are computed as follows:

- Every time a module is actually (re)compiled it’s version number is updated to the current timestamp (but see Section 2.5).
- If the interface[2] for a particular declaration changes or a new declaration is created its version number is updated to the current timestamp.
- It follows that the module version number is always greater than all the declaration version numbers within that module.

A crucial property of version numbers is that they are strictly increasing — the current version number of a particular entity is greater than all previous version numbers associated with that entity.

[2] As far as the compilation system is concerned the *interface* of a declaration includes its type *and* any optimisation pragmas.

2.2 Usage numbers

When a module is compiled it records the version numbers of all the imported declarations which were *used* by that compilation. We call this set of imported version numbers the *usage numbers*. They are also recorded in the module's interface file (see Section 3).

If the module is then recompiled without any source modifications the usage numbers recorded during the previous compilation are compared with the current version numbers to determine if the recompilation is unnecessary.

2.3 Original names

To avoid any ambiguity the compilation system always refers to an entity using its *original name*. This consists of the name of the module in which the entity was originally defined and its name within that module and is written *OriginalModule*`.`*Name*. The use of original names by the compilation system should not be confused with the use of *qualified names* (*ImportedModule*`.`*Name*) in the source code. Qualified names are a Haskell 1.3 language feature used to disambiguate source names within a particular module.

2.4 Checking if recompilation is required

Recompilation of a module is invoked by the `make` utility (Feldman [1979]) if its source has changed or the interface of any module imported directly or indirectly has changed. However, recompilation of a module A, will be deemed unnecessary and abandoned if:

1. The source file A`.hs` has not been modified,[3] AND

2. The interface file A`.hi` exists, AND

3. $\forall$ declarations $M.d \in$ *usages*(A): $\mathcal{U}(A.M.d) = \mathcal{V}(M.d)$

   where *usages*(A) refers to all the imported declarations used when A was last compiled i.e. the declarations with usages recorded in A`.hi`.
   $\mathcal{U}(A.M.d)$ is the usage of declaration $M.d$ recorded in A`.hi`
   $\mathcal{V}(M.d)$ is the current version of $M.d$ recorded in M`.hi`

   If the interface for a particular module cannot be found or does not contain a version number for a particular declaration then recompilation must proceed.

One optimisation which can be made to the recompilation checking algorithm is to compare the usage and version numbers of the module before comparing the usage and version numbers of the declarations within that module. If the current version number of the module is the same as its usage number then the module has not been recompiled (see Section 2.1). It follows that the version numbers of all the declarations within cannot have changed. So condition 3 becomes:

$3'$. $\forall$ modules $M \in$ *modules_used*(A): $\mathcal{U}(A.M) = \mathcal{V}(M)$ OR
$\forall$ declarations $d \in$ *decls_used*$(A.M)$: $\mathcal{U}(A.M.d) = \mathcal{V}(M.d)$.

where *modules_used*(A) refers to all the modules which have usages in A`.hi`
$\mathcal{V}(M)$ is the current version of module M recorded in M`.hi`
$\mathcal{U}(A.M)$ is the usage of module M recorded in A`.hi`
decls_used$(A.M)$ refers to all the declarations from module M which have usages in A`.hi`
$\mathcal{U}(A.M.d)$ is the usage of declaration $M.d$ recorded in A`.hi`
$\mathcal{V}(M.d)$ is the current version of definition $M.d$ recorded in M`.hi`

[3] Currently the test for source modification is is a simple time-stamp comparison — we do not perform any syntactic analysis to determine if the source modifications are irrelevant, i.e. confined to comments and white space.

Checking for recompilation is done by a Perl script before invoking the compiler if recompilation is actually required. Another script is responsible for updating the version numbers in an interface after recompilation. A textual comparison of the current and previous interface information for each declaration is used to determine if the version number for that declaration is updated. Note that *all* the interface information for a particular declaration must be compared, including any pragma information.

2.5 Dependencies

In Section 2.4 we stated that `make` must invoke a recompilation if the source of a module has changed or the interface of any module imported directly or indirectly has changed. Rather than requiring our `make depend` utility to determine the transitive closure of import dependencies we always touch interface files when a module is recompiled, even if the recompilation is deemed unnecessary. In this way the direct import dependencies recorded by `make depend` are propagated.

The only exception to this is if recompilation was required solely because the source file had changed and the new interface is identical to the previous interface (modulo a possibly modified set of usages). In this situation no interface modifications need to be propagated — the new interface file is given the same Unix timestamp and version number as the old interface.

2.6 Instance Declarations

In Haskell 1.3 instances can be declared in any module and are always exported. Rather than recording all the instances exported by a module (which includes all the instances declared in the transitive closure the module's imports) we adopt a scheme which searches for all the instance declarations which may be required by the compilation. The cross-product of all the classes and type constructors required by the compilation is used to determine the possible class/type-constructor combinations for which instance declarations may subsequently be required. (Which instances are actually required is only determined during the type-checking phase of the compiler. For now, at least, we read all the instances which may be required.)

To ensure that we can always find all the instance(s) for a particular class/type-constructor combination we keep track of all the modules which declare an instance, but do not declare either the class or the type constructor involved. In each interface we record the list of *instance modules* which this module knows about. This consists of the module itself if it contains any special instance declarations and the union of the instance modules of its direct imports. The instance(s) for a particular class/type-constructor combination can be located by looking in the interfaces of: the module which declared the class, the module which declared the type constructor and the list of special *instance modules*.

We record the instances declared within a module in a separate section of the interface file. Each interface which contains instance declarations has associated with it a single version number for all its instance declarations. If the interface for any of its instances changes the instance version number is updated. We intend to improve the recompilation resolution of instances by introducing a separate version number for each class/type-constructor instance combination.

3 Interface files

The Haskell 1.3 definition does not define a standard interface format — each implementation is free to define its own interface conventions.

In Glasgow Haskell we adopt the approach that there is only ever one copy of the definition of any particular entity — this resides in the interface for the module in which it was defined. Whenever the compiler requires the definition of an entity the interface of the defining module must be read (even if this module has not been directly imported into the module being compiled). This avoids the duplication of information described in Section 1.1.

An interface file consists of the following sections:

Header: The module name and the *version number* of the interface.

Usages: The *usage numbers* numbers recorded when this module was last compiled (see Section 2.2).

```
interface Digraph 6
__usages__
Bag 4 :: Bag 2 bagToList 2 listToBag 2;
FiniteMap 6 :: FiniteMap 1 listToFM 2 lookupFM 2 lookupWithDefaultFM 2;
GHCbase 22 :: CCallable 8 CReturnable 8 IO 22 lex 14 readList__ 12 showList__\
GHCio 6 :: IOError 1;
Ix 7 :: Ix 1;
List 6 :: partition 2;
Maybes 6 :: MaybeErr 2 maybeToBool 2;
Prelude 15 :: && 2 . 8 Bounded 7 Either 7 Enum 7 Eq 7 Eval 7 Floating 7 Fract\
Ratio 14 :: Ratio 9 Rational 2;
Unique 6 :: Unique 1;
Util 4 :: isIn 2 panic 2;
__versions__
Cycle 1
Edge 1
SCC 1
dfs 2
findSCCs 2
stronglyConnComp 2
topologicalSort 2
__exports__
Bag.Bag                                       -- exported abstractly
Maybes.MaybeErr
Digraph.SCC (..)                              -- exported with constructors
Digraph.dfs
Digraph.findSCCs
Digraph.stronglyConnComp
Digraph.topologicalSort
__declarations__
type Digraph.Cycle a = [a];
type Digraph.Edge a = (a, a);
data Digraph.SCC a = Digraph.AcyclicSCC a | Digraph.CyclicSCC (Bag.Bag a);
Digraph.dfs :: __forall__ [a] (a -> a -> Prelude.Bool) -> (a -> [a]) -> ([a],\
Digraph.findSCCs :: __forall__ [a,b] {{Prelude.Ord b}} => (a -> (b, Bag.Bag b\
Digraph.stronglyConnComp :: __forall__ [a] (a -> a -> Prelude.Bool) -> [Digra\
Digraph.topologicalSort :: __forall__ [a] (a -> a -> Prelude.Bool) -> [Digrap\
__instances__
instance __forall__ [a] Prelude.Eval (Digraph.SCC a);
```

Figure 1: An example interface file (with sanitised version numbers) — `Digraph.hi`

Versions: The *version numbers* of the module's exposed declarations.

Exports: A list of all the entities exported by this interface. It may include entities which were defined in another module and have been re-exported. For type constructors and classes we include a notion of abstract/non-abstract `(..)`, but we do not include the names of the constructors/methods. These are found by looking at the declaration of the entity.

Fixities: Any fixity declarations for the local declarations which are exported.

Declarations: The definition of all local declarations which need to be *exposed*, i.e. the exported declarations and any other declaration which can be reached from the exported declarations.

Instance Modules: A list of modules which contain special instance declarations (see Section 2.6).

Instances: A list of the instances defined in this module.

Pragmas: Any pragmas for the local declarations.

An example of a straightforward interface file is given in Figure 1. For the purposes of readability the actual timestamps have been substituted with readable small integers.

4 Cyclic module dependencies

Cyclic module dependencies pose a bootstrapping problem. Traditional solutions require the programmer to embark on the cumbersome and error-prone task of hand-writing separate loop-breaking interface files. An inconsistent interface file breaks the security of the compilation system and often remains undetected until a runtime crash occurs! In addition, Haskell 1.3 no longer defines a standard interface format — any loop-breaking interface files would be implementation dependent.

In the light of these problems we intend to implement a scheme which eliminates the task of hand-writing loop-breaking interfaces. Instead, loops are broken by importing the loop-breaking declarations directly from the **source** of the module concerned. This eliminates the possibility of any inconsistencies since the same text is used for both the source and the interface. For example, suppose we have two modules defined as follows:

```
module A
import B
...

module B (B(..),b)
import A (A,a)
data B a b = B1 a | B2 a b
b :: A -> Int
instance Eq a => Eq (B a b) where
...
```

The dependencies generated by these modules are shown in Figure 2. (These dependencies look quite complicated since they are designed to enable a simple cut-off recompilation scheme — if a new interface file is identical to the previous interface file it is not updated and recompilation is not propagated.) To break the loop, one of the import declarations is annotated with the `{-#SOURCE#-}` pragma. For example,

```
module A
import {-#SOURCE#-} B (B(..),b)
```

The resulting dependencies are also depicted in Figure 2. First `A.hs` is compiled with the source interface for module `B` being directly extracted from `B.hs`. Then `B.hs` is compiled with the interface for module `A` being read from `A.hi` as usual.

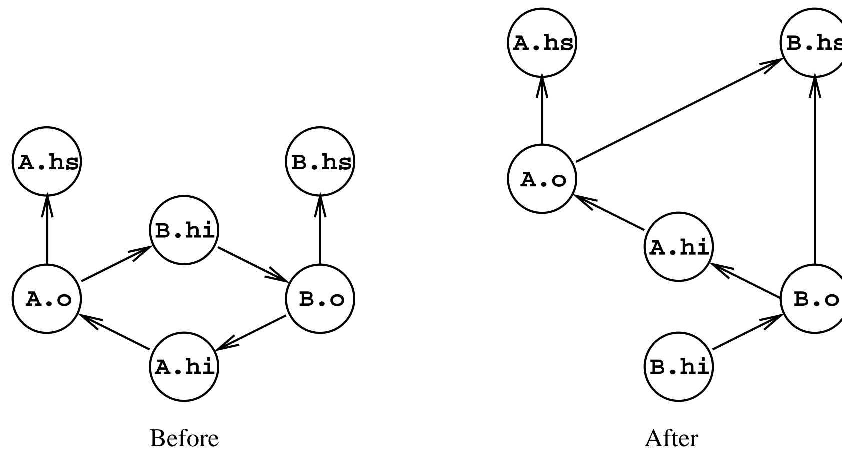

Figure 2: Cyclic module dependencies before and after source import

4.1 Source interfaces

For a module to be the target of an `import {-#SOURCE#-}` declaration it must:

1. Specify a *source interface* using the `{-# SOURCE (`*export-list*`) #-}` pragma (see below).

2. Specify explicit signatures for all the declarations exported by the source interface.

3. Explicitly name any instances to be exported by the source interface in the `SOURCE` pragma by specifying the class/type-constructor pair.

4. Explicitly qualify or explicitly mention in an import list all the external names referred to by the declarations exported in the source interface. This simplifies the process of resolving these external names when extracting a source interface since it enables the interface from which the name is imported to be identified without searching through all the interfaces of unqualified imports. This restriction does not apply to names from the Prelude since this is assumed to be the default.

5. Ensure that all the declarations exported by the source interface are also exported by the normal interface of the module. This is checked when the module is subsequently compiled.

For example,

```
module B (B(..),b)
import A (A,a)
{-# SOURCE (B(..), b, Eq B) #-}

data B a b = B1 a | B2 a b
b :: A -> Int
instance Eq a => Eq (B a b) where
...
```

The `SOURCE` pragma identifies the subset of declarations to be extracted from the source when constructing the source interface. It also provides explicit documentation of those declarations used to break the loops in the module structure.

A source interface does not include any pragmatic information. Such information is compiler generated and, as such, only appears in compiler generated interface files. Though it would be possible to extract pragmatic information

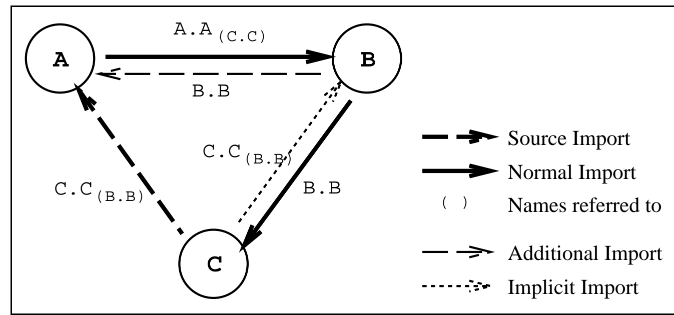

Figure 3: Implicit source imports

from the implementation of function in the source we do not believe that this would be of much benefit since source interfaces are intended to be used only when necessary and only at the points of least coupling.

The interface file for a module which imports a source interface records a single *source usage number* for the module. This is based on the timestamp of the source file from which the interface was extracted. A source usage number is identified by a `!`. When checking if recompilation is required a source usage number is compared with the current timestamp of the source file rather than the version number in the module's interface file.

4.2 Implicit source imports

Consider a cyclic module dependency involving three modules in which module `A` imports the source interface declaration of `C.C` which refers to `B.B` and exports a declaration of `A.A` which refers to `C.C` (see Figure 3). After a change to the source of module `C` recompilation would proceed as follows:

1. `A` is compiled importing the source interface from `C.hs`; `A.hi` is updated.
2. `B` is compiled importing `A.hi`; `B.hi` is updated.
3. `C` is compiled importing `B.hi`; `C.hi` is updated.

This scenario contains two different forms of implicit imports which are resolved in different ways.

Implicit names referred to by a source interface. The source interface of module `C`, imported when module `A` is compiled, refers to `B.B`. To ensure that a change to the declaration of `B.B` would cause module `A` to be recompiled we require that `B.B` be explicitly imported by `A`. In this case `B.B` must be imported from `B`'s source interface. In general, we require that all the names referred to by a source interface are explicitly imported by the module importing the source interface.

Implicit source-interface names referred to by a normal interface. The interface of module `A`, imported when module `B` is compiled, refers to `C.C`. Since `C.hi` has not yet been updated the declaration must be imported from the source interface extracted from `C.hs`. In general, when resolving implicit imports the source interface must be extracted if the actual interface file is out of date. In addition, implicit imports are allowed from an implicitly imported source interface since the dependencies will already have been captured when the source interface was explicitly imported. In this case `C.C` refers to `B.B` which is actually one of the declarations currently being compiled!

4.3 Discussion

The system of source interfaces which we have proposed may seem quite cumbersome. However, we believe that it is a significant improvement on the status quo since it eliminates the very cumbersome and error-prone task of hand-writing separate interface files. It is intended to be used only when cyclic module dependencies are present and only

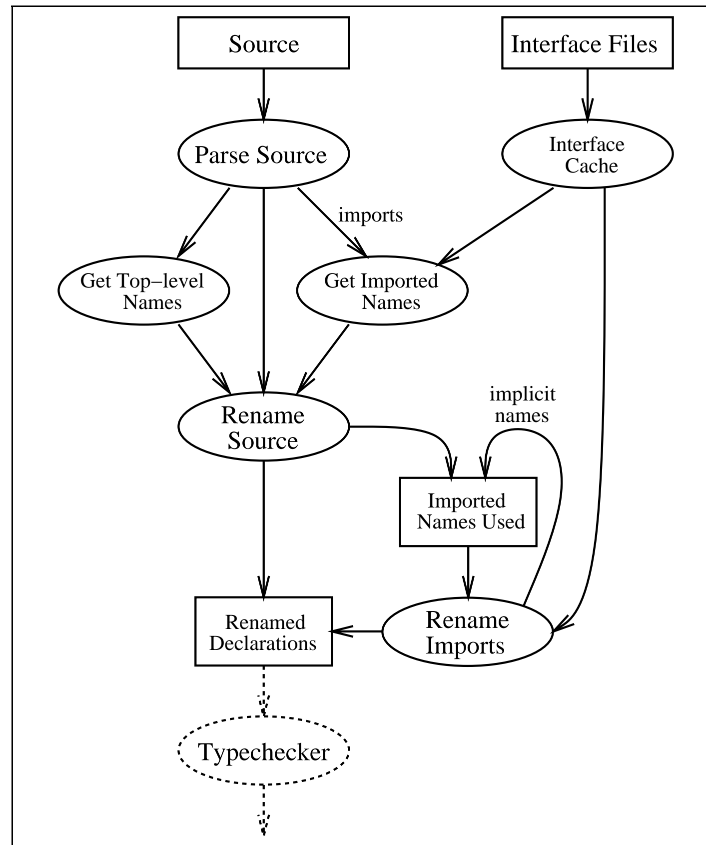

Figure 4: Structure of Renamer and Interface processing

at the points of least coupling. In practice, most systems will have no need to use it at all since they do not have cyclic module dependencies. However, when there are inherent cyclic dependencies we believe it will prove to be a satisfactory solution.

5 Renaming and Interfaces

The Renaming pass of the compiler is responsible for resolving all the names in the source and reading all the required interface information. The term "renaming" refers to the substitution of source and/or interface identifiers with the appropriate `RnName` — an internal data structure which contains all the information about the name as well as a globally `Unique` value which is subsequently used to compare names efficiently. The pass proceeds in four phases:

1. The top-level names are extracted from the parsed source.
2. The source `import` declarations are processed and the directly imported names are read from their original interface file.
3. The parsed source is renamed and the imported names actually used in the source are accumulated. In addition, after resolving the names the renamer sorts out the precedence of any infix operator applications.
4. The full interface information for those names actually *used* in the source are read from their original interface file and renamed. This avoids the "big inhale" of all the interface information. reading and processing of any imported declarations which are not actually used. This process involves a transitive closure operation since these

definitions and pragmas may refer to additional definitions which must also be read and renamed. It is further complicated by the need to read and rename all the imported instance declarations which may be required by the type checker (see Section 2.6).

Underlying this is the interface cache which caches all the interfaces read so far. The main facility it provides is to return the declaration which defines a given original name, reading and caching the interface if necessary. When a normal interface is requested its timestamp must be checked against the timestamp of the corresponding source file. If the interface is out of date a source interface is extracted instead (see Section 4.2).[4]

5.1 The renaming monad

The source and interface parsers both return the same `AbstractSyn` data structures. This enables the same code to be used to rename both source declarations and imported declarations. However, different action is required when resolving names:

- When renaming a source declaration the renaming monad carries down a renaming environment which maps source names in scope to `RnNames`. If the name is not found an `Unbound` name is returned and an error generated.
- When renaming an imported declaration the environment maps original names to `RnNames`. If the name is not found an `Implicit` name is created on-the-fly and returned. The name is also recorded in a special implicit environment which is threaded through the monad. Any subsequent reference to the same original name must return the already created `Implicit` name. When renaming of the declaration is complete the names in the implicit environment are added to the list of imported names still to be read and processed and to the original name environment before processing the next imported declaration. In this way we allocate the `Uniques` of the implicitly imported names as they are encountered.

This is captured by the renaming monad which has two different "modes" of operation. In addition the renaming monad is also responsible for allocating `Uniques`, accumulating errors and warnings, and carrying down any additional information (e.g. the environment of fixity declarations used by the precedence parser).

At the top-level the renamer uses the `IO` monad since retrieving declarations from the interface cache may require a new interface file to be read.

6 Related Work

None of the ideas presented here are particularly new or novel. Rather, this work is an attempt to apply a combination of straightforward ideas to an implementation of a modern programming language which performs significant inter-module optimisations.

The basic recompilation scheme described here is essentially equivalent to Tichy's *smart recompilation* scheme (Tichy [1986]). However, the presentations differ — Tichy presents his *change analysis* in terms of *change sets*, while we introduce and compare version numbers.

Gutknecht [1986] makes the distinction between *resolved* and *unresolved* interfaces — a resolved interface is one which has been extended with all the information it refers to. Our requirement that the original interface contain the only definition of an entity is essentially a strict system of unresolved interfaces.

The idea of processing only those declarations which are actually needed was proposed by Cashin et al. [1981]. We believe that *selective embedding* (as Cashin et al. termed it) is particularly important when each declaration contains a significant body of additional pragmatic information. A less selective approach is *environment pruning* which avoids reading an interface if it can determine that none of the imported entities are used. However, this approach is not really feasible in the presence of unqualified imports.

[4] The module should contain a source interface since the only way for a request for an out-of-date interface to arise is if a reference to a source-imported declaration is exported. If the module does not contain a source interface an error is reported. This can arise if the makefile dependencies are out of date.

The CHIPSY CHILL compiler (Eidnes, Hallsteinsen & Wanvik [1989]) adopted a very similar combination of ideas including a version-based recompilation checker and selective embedding. Though we believe that our detailed presentation is of value, the real interest in our work lies in the proposed evaluation of the scheme which will quantify the effect of introducing the inter-module pragmatic information on a range of straightforward compilation schemes (see Section 7).

The separate compilation system recently developed for the SML/NJ compiler (Appel & MacQueen [1994]) has an incremental recompilation manager built on top (Harper et al. [1994]). This system implements a cut-off recompilation scheme based on module-level dependencies. This is equivalent to our original compilation scheme which avoids propagating recompilation if the new interface is identical to the previous interface.

More advanced, fine-grain recompilation schemes have been proposed for dealing with inter-module dependencies arising from pragmatic information (Burke & Torczon [1993]; Chambers, Dean & Grove [1995]). The basic idea is to record separate versions and usages for each "piece" of pragmatic information associated with a declaration rather than the declaration as a whole. Recompilation is only required if there is a change to the particular pieces of pragmatic information which the compiler made use of. Such schemes are only really feasible in an integrated compilation system where a database of information is maintained.

Finally, we observe that *smartest recompilation* (Shao & Appel [1993]) is not really compatible with inter-module optimisation since no inter-module information is available at compile time. Inter-module consistency is checked during the linking phase of compilation. While it would be feasible, even desirable, to delay optimisation to the linking phase, we have not explored this further.

7 Evaluation

Unfortunately a comprehensive evaluation of the recompilation scheme is not yet feasible — the GHC 1.3 compiler has only just been released and cross-module pragmas have not yet been implemented. However, initial feedback from users has been very positive.

We intend to conduct an experiment similar to that conducted by Adams, Tichy & Weinert [1994], comparing the amount of recompilation required for a naive cascading recompilation scheme with the interface cut-off recompilation scheme used previously and the smart recompilation scheme described here. Within this framework we intend to examine the effect of introducing the inter-module pragmatic information on the performance of the different compilation schemes. The results will be included in a subsequent version of this paper.

We also intend to implement the system of source interfaces described in Section 4 — we currently still rely on hand written interfaces to bootstrap cyclic module dependencies.

A Glasgow Haskell

The Glasgow Haskell Compiler is freely available from a number of ftp sites. For more information please consult the GHC home page:

`http://www.dcs.glasgow.ac.uk/fp/software/ghc`.

Note that the recompilation scheme described here is only available with the Glasgow Haskell 1.3 Compiler — i.e. GHC version 2.xx. The current version (2.01) does not implement source interfaces or cross-module pragmas.

Reference List

R Adams, W Tichy & A Weinert [1994], "The cost of selective recompilation and environment processing," *ACM Transactions on Software Engineering and Methodology* 3(1), Jan 1994, 3–28.

AW Appel & DB MacQueen [1994], "Separate compilation for Standard ML," *SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.

M Burke & L Torczon [1993], "Interprocedural optimisation: Eliminating unnecessary recompilation," *ACM Transactions on Programming Languages and Systems* 15(3), July 1993, 367–399.

PM Cashin, ML Joliat, RF Kamel & DM Lasker [1981], "Experience with a modular typed language," in *Proceedings of the 5th International Conference on Software Engineering*, IEEE Computer Society Press, New York, March 1981, 136–143.

C Chambers, J Dean & D Grove [1995], "A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies," *ICSE'95, Seattle*, April 1995.

H Eidnes, SO Hallsteinsen & DH Wanvik [1989], "Seperate Compilation in CHIPSY," *ACM SIGSOFT* 17(7), Nov 1989.

SI Feldman [1979], "Make — A program for maintaining computer programs," *Software — Practice and Experience* 9(3), Mar 1979, 255–265.

J Gutknecht [1986], "Seperate compilation in Modula-2: An approach to efficient symbol files," *IEEE Software* 3(6), Nov 1986, 29–38.

R Harper, P Lee, F Pfenning & E Rollins [1994], "Incremental Recompilation for Standard ML of New Jersey," Technical Report CMU-CS-94-116, Carnegie Mellon University, Feb 1994.

P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson [1992], "Report on the functional programming language Haskell, Version 1.2," *ACM SIGPLAN Notices* 27(5), May 1992.

JC Peterson, K Hammond, L Augustsson, B Boutel, FW Burton, JH Fasel, AD Gordon, RJM Hughes, P Hudak, T Johnsson, MP Jones, SL Peyton Jones, A Reid & PL Wadler [1996], "Report on the functional programming language Haskell, Version 1.3," In preperation Yale University and University of St Andrews, 1996.

Z Shao & AW Appel [1993], "Smartest recompilation," *20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, Jan 1993.

W Tichy [1986], "Smart recompilation," *ACM Transactions on Programming Languages and Systems* 8(3), July 1986, 273–291.