Explicit Level Imports

Matthew Pickering[1], Rodrigo Mesquita[1], and Adam Gundry[1]

Well-Typed LLP {matthew,rodrigo,adam}@well-typed.com

Abstract. Cross-stage persistence rules are commonly admitted in multi-stage programming languages. These rules codify the assumption that all module and package dependencies are available at all stages. However, in practice, only a small number of dependencies may be needed at each particular stage.

This paper introduces Explicit Level Imports, a mechanism which gives programmers precise control about which dependencies are required at each stage. Imports are annotated with a modifier which brings identifiers into scope at a specific level. This precision means it is straightforward for the compiler to work out what is exactly needed at each stage, and only provide that. The result is faster compilation times and the potential for improved cross-compilation support.

We have implemented these ideas in GHC Haskell, consider a wide variety of practical considerations in the design, and finally demonstrate that the feature solves a real-world issue in a pragmatic way.

Keywords: Staging · Modules · Compile-time code generation · Haskell

1 Introduction

Haskell is a pure, lazy, functional programming language that supports compile-time code generation using *staged metaprogramming*, a feature called Template Haskell [10]. Staged compilation enables writing code-generating programs (metaprograms) in a safe and expressive way, by writing programs in the host language that themselves generate host language programs.

In practice, the primary use of Template Haskell is to avoid writing boilerplate. Programmers, in their libraries, write metaprograms that accept program fragments as inputs (typically a representation of a type), and generate code that is needed to use that library. Template Haskell is used to generate type class instance definitions, generate lenses for use as record accessors, and many other mundane tasks. This common pattern looks like the following:

```
import Control.Lens.TH (makeLenses)
import App (S)
data T = MkT {_foo :: S}
$(makeLenses ''T)   -- generates foo :: Lens T S
```

Here the *makeLenses* function is imported from a library, and used in a declaration splice to generate some definitions (here a lens binding, but a similar pattern is often used for TH-based deriving of type class instances). The function call *makeLenses* ''*T* will be evaluated at compile-time, during the compilation of the containing module, and its result will be a representation of top-level declarations to be inserted into the program at the location of the splice.

At the moment, the compiler must generate executable code for the dependent module *App* before it starts type-checking this module, because in principle, running the splice might end up executing code from *App*. This has a variety of negative consequences:

- Using Template Haskell causes compile-time performance to suffer due to unnecessary (re)compilation. This is particularly relevant for interactive use of the compiler within an IDE such as Haskell Language Server.
- Cross-compilation is significantly complicated by the need to compile and execute code on the target platform during the build process, as there is no way to execute splices on the host.
- Modules needed exclusively at compile-time must still be linked into the final executable, since any imported module could be used at runtime.

The primary idea of this paper is that the language should make it evident to the compiler which dependencies are needed at runtime and which are needed at compile-time. Once the programmer can be explicit about when each dependency is needed, then the compiler can provide just what is required.

Our proposed language extension will allow the programmer to be explicit about the fact that `makeLenses` is used only in a splice, whereas *App* is imported normally and is definitely not executed in splices:

```
import splice Control.Lens.TH (makeLenses)
import App (S)
```

Not only does this make the code easier to understand, but moreover the compiler can now tell from the imports that the module depends only on the interface of *App*, not on its implementation. Correspondingly, it is possible to start type-checking the module as soon as *App* has been type-checked (before code generation has been completed), and changes to the implementation of *App* that do not affect its interface do not cause recompilation of the importing module.

In practice, many Haskell programs enable `TemplateHaskell` solely to be able to call functions from external packages in top-level splices. Thus versions of this example occur frequently, and using the new feature will merely require the programmer to add the **splice** keyword to a few imports.

1.1 Contributions

In a staged programming language, the type checker uses a level discipline to ensure that evaluation is well-staged [13]. All variables are introduced at with a level, and program contexts require variables to be used at a specific level. The result is a system guaranteeing that compile-time fragments (top-level splices) can be fully evaluated before any runtime fragments (the normal program).

Cross-stage persistence (CSP) rules allow the transport of variables between levels. The admitted CSP rules place constraints on the implementation of the language. For GHC Haskell, baked into the current CSP rules is the assumption that if module M depends on N, then N must be compiled before M, and therefore M is free to use anything from N at compile-time. As demonstrated by the previous example, this is a powerful assumption and a strong constraint on the implementation, as admitting this rule *requires* all dependencies to be compiled, available and ready for compile-time evaluation.

Our first contribution is to name and identify `ImplicitStagePersistence` as a potentially undesirable language feature, and one from which programmers should be able to opt out. Under our new feature `NoImplicitStagePersistence`, imported identifiers are restricted to being used *only* at the level they are explicitly bound at (thus forbidding imported identifiers from occurring within top-level splices or quotes by default). See Section 3.

Once `ImplicitStagePersistence` is controlled, programmers will not get very far writing metaprograms, as CSP is used ubiquitously to transport imported variables between levels. Therefore we also introduce a language extension `ExplicitLevelImports`, which provides explicit annotations on imports to make variables available at a specific level:

- **import splice** A will make bindings from A available in top-level splices.
- **import quote** B will make bindings from B available in quotations.

These extensions will both be specified precisely in terms of levels in Section 4.

Together, `NoImplicitStagePersistence` and `ExplicitLevelImports` solve the shortcomings of Template Haskell presented above, by introducing a finer-grained mechanism for the programmer to control the level at which imported identifiers are introduced, and by restricting identifiers used in expressions to those explicitly bound at the correct level. We have implemented these extensions in the Glasgow Haskell Compiler (GHC) and proposed them to the GHC Steering Committee to be officially accepted as GHC extensions.[1]

2 Background

In this paper we are mostly concerned with *untyped* Template Haskell [10], an extension to Haskell [4] that adds support for metaprogramming.

Template Haskell has support for generating and inspecting expressions, declarations, patterns, types and names. Quotations in these contexts allow users to inspect and manipulate the syntax of these forms, and splices allow users to combine syntax together in order to form larger programs. By judicious combination of quotes and splices, the user writes a program generator (metaprogram) that will be executed during compilation in order to generate part of the final program.

[1] See GHC Proposal #682: Explicit Level Imports.

2.1 Syntax

An expression $e :: a$ can be quoted to generate the expression $[\![\ e\ ]\!] :: Q\ Exp$. Conversely, an expression $c :: Q\ Exp$ can be spliced to extract the expression $\$(c)$ that c represents. Template Haskell also has support for quoting declarations $[\![\ d\ ]\!]_D :: Q\ Dec$, patterns $[\![\ p\ ]\!]_P :: Q\ Pat$, types $[\![\ t\ ]\!]_T :: Q\ Type$, term-level names $'n :: Name$ and type-level names $''N :: Name$.

Intensional code analysis and construction is also supported: after quotation, the user can inspect and manipulate the syntax tree directly, for instance, the Exp type is a normal algebraic datatype that represents Haskell expressions.

A *top-level splice* is a splice ($\$(\cdot)$) not surrounded by any quotations. This marks a location in a program where a metaprogram will be evaluated and the resulting program inserted.

2.2 Levels

In order for a program generator to be executed at compile time, it must depend only on other information available at compile time. A *well-staged program* is one where the compile-time portions of the program can be fully evaluated before the runtime portions.

Every declaration and every (sub-)expression in a program is assigned an integer *level*. These levels are checked by the type-checker in order to ensure that the program is well-staged. The top-level declarations in a module are at level 0. Similarly, any normally-imported bindings are at level 0. The level is increased by 1 when inside a quote and decreased by 1 inside a splice. In short:

- $\$(e)$ is at level n iff e is at level $n - 1$
- $[\![\ e\ ]\!]$ is at level n iff e is at level $n + 1$

Therefore the level of an expression can be calculated as the number of quotes surrounding the expression minus the number of splices.

2.3 Stages

A *stage* is a moment in time for which a module is compiled. We will write $M@S$ to indicate that module M is compiled for stage S. Levels and stages are often confused in literature but it is very important to distinguish between them [13].

In this paper we will typically talk about two-stage evaluation where there are distinct compile-time and runtime stages (C and R, respectively). Compile-time and runtime are distinct stages as programs being executed at compile time may need to be compiled in a different way from those being executed at runtime (e.g. using dynamically-linked object files for compile time, but statically-linked object files for runtime).

In a cross-compilation setting, the need for the stage distinction is even clearer, because the runtime stage needs programs that run on the target architecture, whereas the compile-time stage expects programs to run on the host.[2] Cross-compilation may benefit from three or more stages, because there will likely be a compile-compile-time phase for programs that run at compile-time and generate programs for the host, as opposed to the normal compile-time phase that executes at compile-time and generates programs for the target.

The particular stage structure is not the primary focus of this work, as we regard it as an implementation detail of the compiler, but the implementation is constrained by the assumptions made in the design of the level system about which modules will be available at each stage. More permissive rules lead to more programs being accepted but more requirements placed on making modules available at more stages. Less permissive rules make it harder to write level-correct programs but place fewer requirements on which modules are required.

2.4 Cross-Stage Persistence

If an identifier is used at a level different from the level at which it is bound, the program is level-incorrect, but Template Haskell provides two implicit mechanisms that are used to attempt to fix its level via *cross-stage persistence* (CSP) [12]:

Path-based persistence: allows global definitions at level m to be made available at a different level n in two cases:

- If $n > m$, intuitively because all global definitions will still exist in the defining module even if references to them are spliced at a future stage. For example, this allows a module to define a top-level identifier and refer to it in a quote in the same module.
- If $n < m$ and the definition was *imported* rather than being defined in the current module, intuitively because the dependency order on modules ensures the definition must have been compiled already. For example, this allows an imported identifier to be used in a splice.

It is not possible for a global definition to be used in its defining module at a level earlier than its definition, because that would require parts of the module to be compiled to executable code before other parts were type-checked.

Serialisation-based persistence (Lift): locally-bound variables can't be persisted using path-based persistence, but when the variable's type is serialisable, its value can be serialised to persist it to *future* stages. This serialisation is defined as the *lift* method of the *Lift* type class. For instance, the following program is level-incorrect as x is bound at level 0 but used at level 1. However, it is fixed by serialisation-based persistence, which transforms the program into one where x is used at level 0 by the compiler automatically inserting a call to *lift*:

$$\mathit{tardy}\ x = [\![\ x\ ]\!] \quad \Longrightarrow \quad \mathit{tardy}\ x = [\![\ \$(\mathit{lift}\ x)\ ]\!]$$

[2] GHC currently gets around this requirement by compiling all splices for the target, by having a target environment around at compile time in which to execute them, which is complex and limits contexts in which Template Haskell can be cross-compiled.

All base types such as *Int*, *Bool*, *Float*, etc, instantiate *Lift*, and user types can instantiate it automatically with the `DeriveLift` extension (which will generate code that relies on path-based persistence).

It is not possible for a locally-bound variable to be used earlier than the stage at which it is bound, e.g. $[\![\ \lambda x \rightarrow \$(x)\ ]\!]$ is irredeemably stage-incorrect.

Example: The following program requires both implicit mechanisms in order to be accepted. *Path-based persistence* explains why the occurrence of *suc* in examples *one* and *anotherOne* is accepted (since it is defined at level 0 but used at level 1), and why *anotherOne* can be used in a top-level splice (since it is imported at level 0 but used at level -1):

```
module M2 where
  suc :: Int → Int
  suc = (+1)
  one :: Q Exp
  one = ⟦ λx → suc x ⟧
  anotherOne :: Int → Q Exp
  anotherOne y = ⟦ suc y ⟧
```

```
module M3 where
  import M2 (anotherOne)
  two = $(anotherOne 1)
```

Serialisation-based persistence explains why the *y* in *anotherOne* can be moved from a value that exists at level 0 to one that exists at level 1. The compiler will implicitly introduce a call to *lift*:

$$\mathit{anotherOne}\ y = [\![\ \mathit{suc}\ y\ ]\!] \quad \Longrightarrow \quad \mathit{anotherOne}\ y = [\![\ \mathit{suc}\ \$(\mathit{lift}\ y)\ ]\!]$$

And *lift* will take care of converting the compile-time *y* into a runtime value.

Cross-stage persistence and Stages: When compiling, the build system demands the compilation of modules at particular stages. Then by reading the module header, further demands are placed upon the imported modules. It is a key design constraint that the entire build plan for a multi-module program can be determined solely by reading the module headers.

Serialisation-based persistence elaborates a level-incorrect program into a level-correct one that the user themselves could have written. Therefore it does not impose any requirements or use any assumptions about the stages for which modules are compiled.

However, modules using path-based cross-stage persistence place strong requirements on the set of dependencies that must be demanded. Consider two modules *B* and *C* that use cross-stage persistence:

```
module A where {a = 1 :: Int}
module B where
  import A
  foo = a
  bar = ⟦ foo ⟧
```

```
module C where
  import B
  c :: Int
  c = $(bar)
```

Cross-stage persistence means that any identifier in scope may be used in a top-level splice or a quotation. When compiling *C*@*R*, *bar* from *B* is used only in a top-level splice, but this can't be determined from the module header. Instead, since *C* imports *B*, the build system must presume that both *B*@*R* and *B*@*C* are needed. Similarly, if compilation of *B*@*C* is required, then it is also necessary to compile *B*@*R* because the *foo* identifier appears in a quote and is persisted from level 0 to level 1 (so the resulting program may splice *foo* and hence it may appear at runtime).

Ultimately, cross-stage persistence forces the build system to compile all modules and require all dependencies for all stages, even if the final program uses only a small fragment of its dependency tree at any particular stage.

3 Implicit stage-persistence considered harmful

Implicit stage persistence seems convenient at first, but is the root of many performance and cross-compilation issues in practice. If imported identifiers can be arbitrarily used at any stage, the compiler must pessimistically assume they will be used at all stages, and therefore it needs to compile all modules in a project for both runtime and compile-time.

Our design allows implicit path-based cross-stage persistence to be disabled. Identifiers must be used at precisely the level they are bound, and no other levels. Instead, we should be able to explicitly control the level at which identifiers from a module are imported. By being very precise about which levels modules are needed at, there are many real world advantages:

1. Currently, if a module enables `TemplateHaskell`, then all imported modules are compiled to object code before name resolution takes place. This ensures that any top level splices that may be encountered are able to be fully evaluated. This is a pessimisation because most of the imported identifiers, which we have taken such pains to ensure we can run, will not actually be used in a top-level splice. Proposals to increase build parallelism (such as #14095) are far less effective in projects that use `TemplateHaskell`, because name resolution depends on code generation for all dependencies. By distinguishing imported modules whose code is executed only at compile time (which in common cases will be a small fraction of imported modules), we are able to improve this pessimisation.
2. GHC offers an `-fno-code` flag that instructs the compiler to parse and typecheck Haskell modules, but not to generate code, so as to offer quicker feedback to the user. However, any modules imported by a module using `TemplateHaskell` must be compiled to object code, despite the fact that we will not generate object code for the module itself. By distinguishing imported modules whose code is executed only at compile time, we can significantly reduce this unfortunate work, and entirely eliminate it in many cases.
3. IDEs such as Haskell Language Server face similar problems, where they are interested only in the result of type-checking modules, but when `TemplateHaskell`

is enabled a large number of modules have to be cautiously compiled to bytecode.

4. Currently, when cross-compiling modules that use `TemplateHaskell`, all imported modules must be compiled for both host and target. By distinguishing imported modules not used at runtime, we can avoid the need to compile them for the target. Similarly, by distinguishing imported modules not used at compile-time, we can avoid the need to compile them for the host. It can be very hard or impossible to make some packages available on some cross-compilation target platforms, so this change would significantly improve the applicability of `TemplateHaskell` in these scenarios.

4 Specification

The purpose of this work is to design a different level system which allows finer grained control of which imported identifiers are available at which level, and hence, which modules will be required at specific stages. In order to do this, we introduce two new language extensions: `NoImplicitStagePersistence` disables path-based cross-stage persistence and forces the programmer to ensure their programs are level-correct explicitly (getting performance benefits as a result); `ExplicitLevelImports` allows for explicit level control via imports.

4.1 `ImplicitStagePersistence`

When the language extension `ImplicitStagePersistence` is disabled for a module (e.g. using `-XNoImplicitStagePersistence`), path-based cross-stage persistence will be disallowed by the compiler. That is, use of a binding at a level other than the level at which it was defined or imported will result in a type error. In particular, bindings imported using traditional **import** statements may not be used inside of top-level splices, nor within quotes. For example, the following is accepted with `ImplicitStagePersistence`, but rejected under `NoImplicitStagePersistence`:

```
import B (foo)       -- foo :: Q Exp
data C = MkC

quoteC = [[ MkC ]]   -- Error: MkC defined at level 0 but used at level 1
spliceC = $(foo)     -- Error: foo imported at level 0 but used at level -1
```

`ImplicitStagePersistence` is enabled by default in all existing language editions, preserving backwards compatibility. Under `NoImplicitStagePersistence` it is an error to use `DeriveLift` on a type unless all its definition is imported at both level 0 and level 1. This is discussed in more detail in Section 6.2.

When a module uses `TemplateHaskell` with `NoImplicitStagePersistence`, the module dependencies no longer need to be pessimistically compiled and loaded at compile time. Instead, the modules that are needed at compile-time versus runtime are determined by the explicit **splice** and **quote** imports relative to the module being compiled, which are enabled by `ExplicitLevelImports`.

4.2 `ExplicitLevelImports`

The `ExplicitLevelImports` extension introduces two new import modifiers to the import syntax, **splice** and **quote**, which control the level at which identifiers from the module are brought into scope:

- A **splice** import of *A* imports all bindings of *A* to be used *only* at level -1.
- A **quote** import of *B* imports all bindings of *B* to be used *only* at level 1.

For example, the following is accepted with `ExplicitLevelImports`:

```
import quote Foo (bar)    -- bar is introduced at level 1
import Foo (baz)          -- baz is introduced at level 0
import splice Foo (qux)   -- qux is introduced at level -1
foo = baz ⟦ bar ⟧ $(qux)
```

`ExplicitLevelImports` implies `NoImplicitStagePersistence`, to ensure users importing modules just at the correct levels benefit from the compiler performance benefits by default. Nonetheless, it is permitted to enable together `ExplicitLevelImports` and `ImplicitStagePersistence`. This allows **splice** and **quote** imports to be used, but `ImplicitStagePersistence` still allows cross-stage persistence (and thus the compiler must still assume all modules are needed at all stages). This combination is supported to allow gradual migration of code bases following the change, and for corner cases such as programmatic code generation, where the programmer may wish to use the syntax of **splice** and **quote** imports without obliging the whole module to be level-correct.

4.3 Names and Exports

Name resolution ("renaming") does not take account of the level at which an identifier was imported when disambiguating ambiguous names, even though this is sometimes more conservative than necessary. For example, the following program is rejected:

```
import A (x)
import splice B (x)
foo = $(x) x
```

In this case, there is, in principle, no ambiguity because *A.x* isn't allowed to be used in the top-level splice, and *B.x* isn't allowed to be used outside the splice. However, we choose to reject this disambiguation to keep the design simple and prevent any confusion about what is in scope. This position is conservative, and can be relaxed in the future if more flexibility appears worthwhile. This choice follows the Lexical Scoping Principle. A positive consequence of this design choice is that if a program is accepted with `ExplicitLevelImports`, it will be accepted after erasing all **splice**/**quote** keywords and using `ImplicitStagePersistence` instead of `ExplicitLevelImports`.

Exports: Under `NoImplicitStagePersistence`, modules can only export bindings available at level 0. For example, the following program is is rejected because *bad* is imported at level -1 but used at level 0:

```
module M (bad) where
  import splice N (bad)
```

4.4 Type-class instance resolution

Type-class instances are available at the levels they were imported, much like identifiers, can only be used at those levels under `NoImplicitStagePersistence`. In detail:

- Instance resolution views the set of instances from all imports together and thus instances from normal and leveled imports must agree with each other.
- After instance resolution has selected an instance, it is checked which levels the instance is available at and an error is raised if the instance is not available at the correct level.
- Instances defined in the current modules are at level 0, just like top-level variable definitions in a module.

This design for instances mirrors the situation for name resolution. As with ambiguous names, it would in principle be possible for the type-checker to make use of level information to accept more programs, but this seems like an undesirable level of complexity. Consider the following example modules:

```
module X where
  data X = MkX
```

```
module Normal where
import X
instance Show X where
  show _ = "normal"
```

```
module Splice where
import X
instance Show X where
  show _ = "splice"
```

The following program, in principle, could be accepted since the overlapping instances for *Show X* in the *show* call are available at different levels, however, we choose to reject the program (just like we do for ambiguous names):

```
import X (X (..))
import splice X (X (..))
import Normal ()         -- imports instance at level 0
import splice Splice ()  -- different instance at level -1
s1 = show MkX
```

On the other hand, the following program imports the same *Show X* instance at both level 0 and level -1, allowing it to be used at both levels. It is accepted:

```
module X where
  data X = MkX deriving Show
module Bottom where
```

```
import X (X (..))             -- imports instance at level 0
import splice X (X (..))      -- imports the same instance at level -1
import splice Language.Haskell.TH.Lib (stringE)
s1 = show MkX                 -- Uses instance at level 0
s2 = $(stringE (show MkX))    -- Uses instance at level -1
```

Exports of class instances: Only instances available at level 0 are re-exported from a module, just like for identifiers. For example, the following is rejected in the call to *show* since no instance for *Show X* is in scope:

```
module X where
  data X = MkX

module Splice where
  import X
  instance Show X where
    show _ = "splice"

module Y where
  import splice Splice ()

module Bottom where
  import X (X (..))
  import Y ()
  s1 = show MkX
```

Even though *Y* has access to the instance at level -1, it does not re-export it. Thus *Bottom* does not import the instance. This is necessary for a clean separation between stages, because instances may exist only at compile-time or only at runtime, just like identifiers.

5 Examples

5.1 Splice imports

A "splice" import is prefixed with **splice**. In this example, identifiers from *A* can be used only in top-level splices and identifiers from *B* cannot be used in quotes or splices:

```
import splice A (foo)   -- foo :: Int → Q Exp
import B (bar)          -- bar :: Int → Q Exp

x = $(foo 25)   -- Accepted
y = $(bar 33)   -- Error: bar imported at level 0 but used at level -1
```

Thus:

1. When compiling module *Main* only identifiers from module *A* will be used in top-level splices so only *A* (and its dependencies) needs to compiled to object code before starting to compile *Main*.
2. When cross-compiling, *A* needs to be built only for the host and *B* only for the target.

If the same module is needed to be used at different levels then two import declarations can be used:

```
import C
import splice C
```

5.2 Quote imports

A quote import is prefixed with **quote**. In this example, identifiers from A can be used only in quotes, while identifiers from A cannot be used at the top-level or in splices:

```
import quote A (foo)   -- foo :: Int → Int
import B (bar)         -- bar :: Int → Int
x = ⟦ foo 25 ⟧   -- Accepted
y = ⟦ bar 33 ⟧   -- Error: bar imported at level 0 but used at level 1
```

When a quote such as $x = [\![\ foo\ 25\ ]\!]$ is spliced, i.e. $z = \$(x)$, its contents will be needed to execute the program at runtime ($z = foo\ 25$, so evaluating z at runtime requires *foo* to be available).

5.3 Module Stages

In section 2.3 we said that modules were compiled for either the C or R stages. Levelled imports make it possible to be precise about what stages we need dependencies.

- The main module is compiled for R. This is where the *main* function lives and the entry-point to running the resulting executable.
- A normal import does not shift the stage at which the dependent module is required.
- If a module M splice imports module A, then compiling $M@R$ requires compiling module $A@C$.
- If a module M splice imports module A, then compiling $M@C$ requires compiling module $A@C$.
- If a module N quote imports module B, then compiling $N@C$ requires compiling module B at $N@R$.
- If a module N quote imports module B, then compiling $N@R$ requires compiling module B at $N@R$

Stage arithmetic is saturating. Thus, when there are two stages, a quote import corresponds to requiring the module at R, and a splice import to requiring a module at C. When there are more than two stages then the imports can have different meanings depending on the stage a module is compiled for. The compiler can then choose appropriately how modules needed at C are compiled and how modules needed at R are compiled. For example:

- In `-fno-code` mode, C modules may be compiled in dynamic way, but R modules are not compiled at all.
- When using a profiled GHC. C modules must be compiled in profiled way but R modules will be compiled in static way.

Further level structure as needed by cross-compilation settings may require more stages. This will be easily possible to change once the level discipline is enforced.

5.4 Module stage offsetting example

The interaction between stages and level offsetting can be understood more clearly through an example. Module *A* splices *foo* from module *B* which both quotes *bar* from module *C* and uses *baz* from *D*:

```
module A where
  import splice B (foo)
  x = $(foo 10)
module C where
  bar = 42
module D where
  baz 0 = True
  baz _ = False

module B where
  import D (baz)
  import quote C (bar)
  foo x
    | baz x = [[ bar * 2 ]]
    | otherwise = [[ bar ]]
```

In *A*, *foo* can be used within a splice (level -1) because of the splice import (-1). In *B*, *bar* can be used within a quote (level +1) because of the quote import (+1) Now, consider compiling *A*@*R*.

- *B* is required at stage *C*, as it is splice imported from *A*@*R*.
- *C* is required at stage *R*, as it is quote imported from *B*@*C*.
- *D* is required at stage *C*, as it is normally imported from *B*@*C*.

Therefore in order to compile *A*@*R*, we have performed dependency resolution and require *B*@*C*, *C*@*R* and *D*@*C*.

The perhaps curious case is *D*: is it needed at compile-time or runtime? It does not use a splice import, so one could think it is needed at runtime – but here is where the distinction between the import level offset and base stage is relevant. *D* is only being imported as a dependency of *B*, which is at *C* stage. This makes *D* *also* at the *C* stage! Note how *baz* is needed at compile time just to define *foo*, which is properly **splice** imported.

The levels of all modules in the transitive closure of a **splice**-imported module are offset by -1. Conversely, **quote** imports offset the levels by +1, thereby making all the levels align correctly.

6 Discussion

6.1 Case Study: `pandoc`

The pandoc library is a medium-sized package that contains approximately 200 modules. It uses `TemplateHaskell` in a light manner in order to embed some data files and derive some JSON instances.

Modifying the package to use `ExplicitLevelImports` required little effort and involved modifying the imports of the 5 modules in the project which use `TemplateHaskell`.

Previously, type-checking the library (loading it into GHCi using the `-fno-code` option) would needlessly compile the majority of modules, as one module near the root of the module graph used `TemplateHaskell`. Following our changes, this is no longer necessary and the recompile time is halved.

From looking at the imports of modules, it can observed that no modules from the `pandoc` library are used in compile-time evaluation and only a few external packages are needed at compile-time.

6.2 Implicit lifting and deriving *Lift* instances

Lift instances are used to provide serialisation-based cross-stage persistence. For example, a typical *Lift* instance looks like:

```
data MInt = Some Int | None
instance Lift MInt where
  lift :: MInt → Q Exp
  lift None = ⟦ None ⟧
  lift (Some x) = ⟦ Some $(lift x) ⟧
```

The presence of this instance means the following declaration will be accepted:

```
foo :: MInt → Q Exp
foo x = ⟦ x ⟧    ⟹    foo x = ⟦ $(lift x) ⟧
```

Defining a *Lift* instance requires the datatype constructors to be available both at compile-time and runtime, so defining *Lift* within the same module as the datatype itself requires path-based cross-stage persistence. Operationally, *None* and *Some* are needed both at compile-time and runtime since they are both matched on at compile time, and also persisted to be spliced in the future into a program that can make use of them at runtime. As a result, it isn't possible to define or derive a (non-orphan) *Lift* instance under `NoImplicitStagePersistence`.

An orphan *Lift* instance can be defined thus:

```
module N where
  import M
  import quote M
  instance Lift MInt where
    lift :: MInt → Q Exp
    lift None = ⟦ None ⟧
    lift (Some x) = ⟦ Some $(lift x) ⟧
```

```
module M where
  data MInt
     = Some Int
     | None
```

This isn't technically problematic, rather it is just a result of what *Lift* means. However, it means some users may need to modify their use of *Lift* instances if they wish to benefit more from `NoImplicitStagePersistence`. Users are free to use `ImplicitStagePersistence` in selected modules to allow defining *Lift* instances, but doing so means all the dependencies of the module will need to be available both at compile-time and runtime.

As a general rule, *Lift* instances should be defined only for simple datatypes near the root of the module hierarchy of an application.

Just as `NoImplicitStagePersistence` allows users to disable implicit path-based cross-stage persistence, it would make sense to have an extension flag to disable implicit lifting (serialisation-based persistence). This would allow the programmer to ensure they are explicit about where calls to *lift* occur in their programs, which is sometimes desirable when using staging for runtime performance.

6.3 Future work: Level-correct package dependencies

The **splice** and **quote** imports in this work make it possible to express which module dependencies are required at which stages, within the Haskell language.

However, large Haskell programs are typically organised into multiple packages, using the Cabal package system to describe their dependencies. While our proposed feature delivers significant benefits to the compilation process for a single package, it would ultimately make sense to expose level distinctions in Cabal package dependencies, so that Cabal could build package dependencies only for the stages at which they are required. This would primarily be of value in cross-compilation scenarios.

In the interests of keeping the work manageable, changes to Cabal are out of scope of the current paper, but we believe it lays a foundation for future work to improve Cabal's cross-compilation support.

6.4 Future work: Typed Template Haskell

Typed Template Haskell (TTH) [15] is an extension of Template Haskell that allows using type-safe staged programming for program optimisation. Its typical use cases are rather different from untyped TH, since in particular it does not support declaration splices.

The same level checks can be used for typed quotes and splices as for the untyped case. However, when using TTH and explicit level imports, the programmer can introduce level errors that cannot currently be worked around. For example, the following program contains a stage error as the evidence for *Show a* is bound earlier than it is used, but it is currently mistakenly accepted by GHC:

```
foo :: Show a ⇒ Code Q (a → String)
foo = ⟦ show ⟧
```

The language of constraints is not yet expressive enough to communicate that the *Show a* evidence needs to be available at a later stage. Fixing this problem will require significant additional effort to resolve other known issues with TTH [8].

7 Related Work

The idea to use import modifiers to accept the level of identifiers originates from Racket [2]. We are also inspired by MacoCaml [17,16] which suggests the an import modifier similar to **splice**. Our work brings those ideas to Haskell, and tackles the challenges faced when integrated leveled imports into Haskell, building on language design discussions in the GHC proposals process.[3]

7.1 Multiple levels within a single module

The design in this paper requires each module to exist at a single level, which may sometimes necessitate users introducing more modules than would be ideal.

One possible design that mitigates the need for module-level granularity of imports, inspired by the Racket [2], MacoCaml [16,1] and MetaFM [11] languages, is the introduction of an additional **macro** keyword that introduces bindings at a different level. A **macro** annotated binding would introduce a binding at the -1 level, without requiring it to be **splice** imported from a different module.

Our design lays out the foundation for well-leveled programs, and is forward-compatible with such a **macro** keyword, or other possible features that relax Haskell's identification of compilation units with source files, such as the proposed Local Modules feature. [4]

Another angle is to allow users to define their own level contexts, an idea proposed for Racket [3]. In the future it would also be interesting to explore extending this system to different stages. This could be useful in order to support embedding other modal languages into Haskell.

7.2 Module Generation

Functors in ML family languages allow users to parameterise modules. It's therefore a natural consideration to remove the abstraction overhead by using ideas from staging [5,14,9]. Modules are neither first-class nor parameterised in Haskell but Explicit Level Imports could be considered a simple form of module generation: by making a demand on a module at a particular stage, the compiler will generate it for that stage. The choice is either to wholly include a module or not include it at all.

In ML family languages, modules and module functors are a primary means of abstraction. In Haskell, a similar role is instead played by type classes. Therefore the ideas in these papers are more suitably considered to apply to modifications to type class mechanisms than modules themselves. For example, you can see parallels in techniques used in CFTT [6] or Staged SOP [7].

Package-level abstraction is implemented for GHC in the Backpack [18] extension, which currently sees little practical use. Similar ideas might also be useful in reviving interest in this area.

---

[3] See GHC proposals #243: Stage Hygiene for Template Haskell and #412: Explicit Splice Imports.

[4] See GHC proposal #283: Local Modules

8 Conclusion

We have presented the design of Explicit Level Imports, a simple system that allows programmers to be precise about module dependencies when using Template Haskell. By using explicit dependencies, the modules required for each stage are evident to the compiler. This leads to important practical benefits gained by separating runtime from compile-time dependencies. The system described has been implemented and verified to achieve significant practical improvements to projects using Template Haskell. In the next stage of the project we aim to finish the implementation and contribute the feature upstream to GHC.

9 Acknowledgments

We thank the participants in the GHC proposal process, in particular John Ericson, Sebastian Graf and Arnaud Spiwack, for their helpful comments on the GHC proposal that preceded this paper. Thanks also to Mercury for funding this work.

References

1. Chiang, T.J., Yallop, J., White, L., Xie, N.: Staged compilation with module functors. Proc. ACM Program. Lang. **8**(ICFP) (Aug 2024). https://doi.org/10.1145/3674649, https://doi.org/10.1145/3674649
2. Flatt, M.: Composable and compilable macros: you want it when? ACM SIGPLAN Notices **37**(9), 72–83 (2002)
3. Flatt, M.: Submodules in Racket: you want it when, again? In: Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences. p. 13–22. GPCE '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2517208.2517211, https://doi.org/10.1145/2517208.2517211
4. Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M.M., Hammond, K., Hughes, J., Johnsson, T., et al.: Report on the programming language Haskell: a non-strict, purely functional language version 1.2. ACM SigPlan notices **27**(5), 1–164 (1992)
5. Inoue, J., Kiselyov, O., Kameyama, Y.: Staging beyond terms: Prospects and challenges. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 103–108 (2016)
6. Kovács, A.: Closure-free functional programming in a two-level type theory. Proceedings of the ACM on Programming Languages **8**(ICFP), 659–692 (2024)
7. Pickering, M., Löh, A., Wu, N.: Staged sums of products. In: Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell. pp. 122–135 (2020)
8. Pickering, M.T.: Understanding the Interaction Between Elaboration and Quotation. Ph.D. thesis, University of Bristol (2021)
9. Sato, Y., Kameyama, Y., Watanabe, T.: Module generation without regret. In: Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. p. 1–13. PEPM 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372884.3373160, https://doi.org/10.1145/3372884.3373160
10. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 1–16. Haskell '02, ACM, New York, NY, USA (2002). https://doi.org/10.1145/581690.581691, http://doi.acm.org/10.1145/581690.581691
11. Suwa, T., Igarashi, A.: An ML-style module system for cross-stage type abstraction in multi-stage programming. In: Functional and Logic Programming: 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15–17, 2024, Proceedings. p. 237–272. Springer-Verlag, Berlin, Heidelberg (2024). https://doi.org/10.1007/978-981-97-2300-3_13, https://doi.org/10.1007/978-981-97-2300-3_13
12. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. pp. 203–217. PEPM '97, ACM, New York, NY, USA (1997). https://doi.org/10.1145/258993.259019, http://doi.acm.org/10.1145/258993.259019
13. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1-2), 211–242 (2000). https://doi.org/10.1016/S0304-3975(00)00053-0, https://doi.org/10.1016/S0304-3975(00)00053-0
14. Watanabe, T., Kameyama, Y.: Program generation for ML modules (short paper). In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 60–66 (2017)

15. Xie, N., Pickering, M., Löh, A., Wu, N., Yallop, J., Wang, M.: Staging with class: a specification for Typed Template Haskell. Proceedings of the ACM on Programming Languages **6**(POPL), 1–30 (2022)
16. Xie, N., White, L., Nicole, O., Yallop, J.: MacoCaml: staging composable and compilable macros. Proceedings of the ACM on Programming Languages **7**(ICFP), 604–648 (2023)
17. Yallop, J., White, L.: Modular macros. In: OCaml Users and Developers Workshop. vol. 6 (2015)
18. Yang, E.Z.: Backpack: Towards practical mix-in linking in Haskell. Ph.D. thesis, Stanford University (2017)