Multi-stage Programs in Context

Matthew Pickering
University of Bristol
United Kingdom

Nicolas Wu
Imperial College London
United Kingdom

Csongor Kiss
University of Bristol
United Kingdom

Abstract

Cross-stage persistence is an essential aspect of multi-stage programming that allows a value defined in one stage to be available in another. However, difficulty arises when implicit information held in types, type classes and implicit parameters needs to be persisted. Without a careful treatment of such implicit information—which are pervasive in Haskell—subtle yet avoidable bugs lurk beneath the surface.

This paper demonstrates that in multi-stage programming care must be taken when representing quoted terms so that important implicit information is kept in context and not discarded. The approach is formalised with a type-system, and an implementation in GHC is presented that fixes problems of the previous incarnation.

CCS Concepts • **Software and its engineering → Functional languages**; *Source code generation*; *Preprocessors*.

Keywords metaprogramming, staging, implicits

ACM Reference Format:
Matthew Pickering, Nicolas Wu, and Csongor Kiss. 2019. Multi-stage Programs in Context. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3331545.3342597

1 Introduction

Staging is a powerful technique that allows programmers to carefully specify how code is constructed. The technique has proven itself to be popular for implementers wishing to fine-tune their libraries with predictable performance. Typed Template Haskell[1] is a *compile-time* metaprogramming facility for GHC that generates programs which are guaranteed to be well-typed and well-scoped. The implementation follows in the tradition of MetaML [18] and MetaOCaml [5].

[1]Not to be confused with (untyped) Template Haskell

Haskell '19, August 22–23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342597

There are two fundamental syntactic constructs, the quote and the splice. In essence, for an expression $e :: T$ the *quote* is written $[\![e]\!] :: Code\ T$ and is an opaque abstract syntax tree that represents e. For $c :: Code\ T$, the *splice* is written $\$(c) :: T$ and evaluates the expression c which is then inserted into the representation.

But what does the programmer mean by $[\![e]\!]$: only the syntactic elements that comprise e, or all of the *implicit* contextual information, such as e's type, that is available at the time of quoting? Those following the Hindley-Milner discipline [2] will say that it should not matter, as the type information can always be rediscovered in a syntactic manner. Indeed, Typed Template Haskell follows this philosophy. Alas, Haskell's type system boasts several extensions that go beyond Hindley-Milner, such as type classes [21] which enable rich overloading of syntax based on type information.

Consider a simple staged program that strips whitespace from the textual representation of an integer numeral by reading it into an *Int* using the *Read* class, then converting it back to a *String* using *Show*. First, quote *show* and *read*:

$$qshow :: Code\ (Int \to String) \qquad qshow = [\![show]\!]$$
$$qread :: Code\ (String \to Int) \qquad qread = [\![read]\!]$$

Then the representation of the normalisation function is obtained by composing the *qshow* and *qread* fragments.

$$trim :: Code\ (String \to String)$$
$$trim = [\![\$(qshow) \cdot \$(qread)]\!]$$

However, when trying to put the function to use by invoking $\$(trim)$ `" 1"`, GHC rejects it with an error:

```
Ambiguous type variable 'a' arising from 'show'
```

What happened? By the time GHC has stitched together *trim*, all that's left is $show \cdot read$, where the intermediate type is indeed ambiguous. This is bad, because the intended meaning of *trim* is clearly to normalise *Int*s because of the types given to *qshow* and *qread*.

In practice, what this means is that the representation ought to contain the implicit information that the compiler holds *at the time of quotation*. This issue has previously been ignored, leading to unexpected behaviour in programs that make sophisticated use of implicit information and quotation. This paper is about fixing this long-standing omission.

After a brief introduction to staging in Typed Template Haskell which provides the necessary background (Section 2), the main contributions of this paper are:

1. a demonstration that cross-stage persistence and elaboration of implicit information have a subtle and intricate interaction (Section 3),
2. a type-system that formalises how the two should relate (Section 4), and
3. the description of an implementation of the system in GHC (Section 5).

Then, comparisons are drawn between the proposed solution and the current implementation of Cloud Haskell which has to deal with similar problems (Section 6). Finally, related work is discussed (Section 7).

2 Background

Multi-stage programming is an important tool that allows programmers to write code that is sensitive to the *stage* that it is evaluated in. For example, a simple Typed Template Haskell program may consist of two stages: compile-time and run-time. The part of the program executed at compile-time generates code which will be executed at run-time. The stages of the program are separated so the fragment of the program which runs at compile-time will not appear in the generated program, thus providing assurance about the efficiency of code produced. The technique has been employed across different languages, and the art of producing multi-stage programs has been explored in depth by many [6, 17].

2.1 Staging

Generated programs in Typed Template Haskell are indicated by the *Code* type constructor.[2] These values are introduced and manipulated by the quote and splice operations that control the structure of the generated program.

To the uninitiated, learning how to write the staged power function $power :: Int \to Code\ Int \to Code\ Int$ is a simple yet instructive example. The expression $power\ n\ k$ computes k^n, and the type signature indicates that the value $k :: Code\ Int$ is a parameter that contains a quoted expression that can only be produced *dynamically* at run-time. On the other hand $n :: Int$ is a parameter known *statically* at compile-time.

The implementation of this function makes use of the static information to unroll the recursion n times:

$$
\begin{aligned}
&power :: Int \to Code\ Int \to Code\ Int \\
&power\ 0\ k = [\![1]\!] \\
&power\ n\ k = [\![\$(k) * \$(power\ (n-1)\ k)]\!]
\end{aligned}
$$

Since the result of $power\ n\ k$ is a value of type $Code\ Int$, it can only be used by applying a splice:

$$
\begin{aligned}
&power5 :: Int \to Int \\
&power5\ k = \$(power\ 5\ [\![k]\!])
\end{aligned}
$$

This has the same effect as defining the function manually as $power5\ k = k * k * k * k * k * 1$, where the multiplications are explicitly unrolled.

2.2 Cross-Stage Persistence

A closed term $e :: T$ can always be quoted as $[\![e]\!] :: Code\ T$. However, it is very common for a term to not be closed: it may, for example, refer to a top level definition or a value held in a variable. In such cases, values need to be persisted from one stage to another. This is know as *cross-stage persistence* [18]. There are two mechanisms to achieve this, *path-based* persistence, and *serialisation-based persistence*.

Path-based persistence Path-based persistence is used to persist top-level identifiers. When a top-level identifier is referenced in a quote then when it is spliced at a future stage, the identifier will still exist in the same module at the top level. For instance, it is possible to persist *not* as it is defined as a top-level definition in the standard Haskell library.

Serialisation-based persistence Serialisation-based persistence is used for values held in variables whose representation can be computed at run-time. For example, all base types such as *Int*, *Bool*, *Float* and so on define an instance of the *Lift* type class which describes how a *Code* value can be generated from the value:

$$
\begin{aligned}
&\textbf{class}\ Lift\ a\ \textbf{where} \\
&\quad lift :: a \to Code\ a
\end{aligned}
$$

The instance for $Lift\ Int$ takes an integer and constructs a syntax node which contains just that integer. Instances of *Lift* are already defined for most primitive types but one notable exception is for functions since there is no general way to inspect and serialise them.

If a variable is used in a stage after it is defined and its type is an instance of *Lift* then the occurence of the variable is replaced by a combination of a $\$(..)$ and *lift*.

$$
\begin{aligned}
&qplusOne :: Int \to Code\ Int \\
&qplusOne\ x = \\
&\quad [\![x + 1]\!] \\
&\quad \rightsquigarrow \quad \{\ \text{elaborated by cross-stage persistence}\ \} \\
&\quad [\![\$(lift\ x) + 1]\!]
\end{aligned}
$$

Users are able to create instances through quotation. Whilst defining an instance it is possible to recursively appeal to the two principles of cross-stage persistence. In the following example the constructors are persisted by virtue of being defined at the top-level whilst the argument to *Some* is persisted using serialisation-based persistence.

$$
\begin{aligned}
&\textbf{data}\ MInt = Some\ Int \mid None \\
&\textbf{instance}\ Lift\ MInt\ \textbf{where} \\
&\quad lift\ (None) \quad = [\![None]\!] \\
&\quad lift\ (Some\ x) = [\![Some\ x]\!]
\end{aligned}
$$

[2] The abstract data type *Code a* is a newtype wrapper around *Q* (*TExp a*), which makes it possible to write class instances later on. In GHC, typed quotes are implemented by `[|| ||]` and typed splices by `$$( )`, rather than $[\![\_]\!]$ and $\$(\_)$. The *Lift* type class has been modified so that lifting results in a value of type *Code* rather than *Q Exp* as in Template Haskell [15].

Instances of the *Lift* class can also be derived automatically using the `DeriveLift` extension.

If a free variable is used in a quote, the compiler first decides whether it can be lifted using the path-based persistence. If not, the serialisation-based persistence is attempted. Failing both of these methods, an error is issued saying that the value cannot be persisted.

Polymorphism GHC does not support impredicative polymorphism and this has ramifications for Typed Template Haskell. In particular, because type variables can not be instantiated with polymorphic types, generated programs have to be monomorphic. If a user writes $v :: Code\ (\forall b.b \to b)$ then it is eagerly rejected by the compiler in order to avoid errors where impredicative instantiation would occur. Given a function $ident :: Code\ a \to Code\ a$, if v were allowed, then the type variable a would have to be instantiated with $\forall b.b \to b$ when *ident* is applied to v. A promising future direction is the work on guarded impredicative polymorphism [14] which would allow impredicative instantiations of type variables that appear under type constructor applications.

This implicit floating of type variables is a seemingly small design decision, but it has large ramifications for cross-stage persistence: type variables are bound outside of the quote so must be taken into account when considering persistence.

3 Implicit Information and Elaboration

In this section the primary problem is introduced by considering the different kinds of implicit information that GHC supports. As seen in Section 1, the problem is that a quotation appears in the source of a program but the meaning of a quotation must take into account implicit elaboration. The primary reason for this is that the type of a quotation influences the process of implicit resolution. The secondary reason is that if implicits are not taken into account then ill-typed terms will be produced and spliced into the program. These incorrectly staged programs need to be identified and corrected by the same principles of cross-stage persistence as identified with explicit type applications.

These problems develop in this section because not only do we have to worry about cross-stage persistence for implicit information but also about the representation of terms. This section argues that:

- for typed quotations it is necessary to use a representation that preserves types, and
- if a type preserving representation is used then cross-stage persistence for implicit information must be considered.

The structure of this section is to first identify three situations where a type preserving representation is necessary in order to stop unexpected problems appearing in programs. Then, for each situation the implications of preserving the relevant implicit evidence is considered.

Types, type classes and implicit parameters are all different kinds of implicits implemented in GHC which interact with multi-stage annotations in an interesting manner.

3.1 Types

The most common form of implicit argument are types. Most functional programming languages have sophisticated type inference which makes specifying the precise types of terms unnecessary. The process of elaboration makes all types explicit during type checking.

Recall the *trim* function from Section 1:

$$trim :: Code\ (String \to String)$$
$$trim = [\![\$(qshow) \cdot \$(qread)]\!]$$

Today, GHC 8.6.5 hastily throws away the type information for *qshow* and *qread*, which results in the following generated code after splicing:

```
show . read
```

There is no hope for the compiler to infer that the roundtripping is intended to be done via *Int*, so it throws the ambiguity error. The solution is to store the type in the representation, so no guesswork is needed after splicing:

```
show @Int . read @Int
```

3.1.1 Persistence for types

Storing the representation of inferred type information in the representation is enough to solve the problem of ambiguity. Then the type checker will not have to perform the same type inference that it already performed when the quote was first type checked.

However, naively storing the type brings about other problems. Consider the quoted *mempty* function:

$$qmempty :: \forall m.Monoid\ m \Rightarrow Code\ m$$
$$qmempty = [\![mempty]\!]$$

Following the agenda outlined above, the the quote should store the type m at which *mempty* is invoked. This is problematic: m is quantified outside of the bracket, which means that after splicing $\$(qmempty)$, the variable will be *unbound*:

```
mempty @m
```

This is because the type variable m is bound in the first stage but used in the second stage.

The attentive reader will notice that this is the same issue as discussed in Section 2.2, but concerning type variables instead of term variables. The solution for values was to apply cross-stage persistence, and we propose an analogous strategy for types. Recall cross-stage persistence of term variables: if there is a *Lift* instance for the type of the variable then we interpret a variable used across stages as the combination of a splice and a lift.

The solution here is to introduce a similar class to persist types to a future stage.

```
class LiftT (t :: k) where
  liftT :: CodeType
```

CodeType is to types as *Code* is to values. *liftT @Int::CodeType* returns the representation of *Int*.

Instances for *LiftT* are solved automatically by the compiler for all types. The class provides an unindexed representation of the type using type representation. This representation is used in an implicit splice in order to insert the correct type, as decided by the host.

The way to interpret a stage error for type variables is the insertion of a splice and a lift. There is no source syntax for splicing a type into a typed quotation so the following code fragment is indicative of the implicit lifting procedure.

```
qmempty :: ∀m.(LiftT m, Monoid m) ⇒ Code m
qmempty =
    ⟦mempty⟧
  ⇝   { inserted type information }
    ⟦mempty @m⟧
  ⇝   { elaborated by cross-stage persistence }
    ⟦mempty @$(liftT @m)⟧
```

By inserting the implicit splice, the stage error has been corrected and now the choice of instantiation for *m* will be persisted. The type of *qmempty* has now been modified to include the *LiftT m* constraint as it is necessary to persist it by creating and splicing the representation.

The usage of the *LiftT* constraint ensures that the type variable which needs to be persisted is eventually instantiated to a concrete monomorphic type.

Constraints are the standard way in Haskell to defer decisions. The way that the constraints are solved (which is explained fully in Section 5) means that there are no instances mentioning free type variables.

3.2 Type Classes

The second most common form of implicit information in Haskell are type classes [21]. They are elaborated during type checking by the constraint solver [20].

Consider the *Show* class with a single method *show* as well as *showInt* which is implemented in terms of *show*.

```
class Show a where
  show :: a → String

showInt :: Int → String
showInt = show
```

The meaning of *show* in a program depends on its type. The most general type requires a *Show a* constraint and the decision delayed until *show* is called. The meaning can be made more specific by specifying that the argument to *show* should be *Int*. In this case the *Show Int* dictionary is supplied directly rather than being passed as an argument.

Due to the elaborated evidence, the meaning of quoting *show* must also depend on its type. Whilst it would be tempting to give ⟦*show*⟧ type *Code* (∀*a.Show a* ⇒ *a* → *String*), the nested quantifier is forbidden due to impredicativity so the quantifier and constraints are floated outwards.

Following the previous example, the type of ⟦*show*⟧ is:

```
⟦show⟧ :: ∀a.Show a ⇒ Code (a → String)
```

It is the host which makes the decision about which instance needs to be used and their decision must be persisted to the target. What about when *show* is specialised to a monomorphic type? Now there is only one possible type for *qshowInt*.

```
qshowInt :: Code (Int → String)
qshowInt = ⟦show⟧
```

When *qshowInt* is spliced, it works as expected for this simple type class hierarchy. However, it hides a subtle problem with the implementation. Typed Template Haskell is implemented by serialising an AST which contains no type information. Thus, whilst you might expect the elaboration of *qshowInt* to also contain information about the *Show Int* instance, it does not.

Instead, the current representation of ⟦*show*⟧ is simply *show* itself with no elaborated evidence. When *qshowInt* is spliced, only the reference to *show* is inserted into the program. The term is then re-elaborated in the new context.

```
  $(⟦show⟧ @Int $dShowInt)
⇝   { Representation contains no type information }
  show
⇝   { Re-elaborated on the target }
  show @Int $dShowInt
```

How did the type annotation and dictionary application get inferred? The global type class coherence condition ensures that there is only one instance for each type class for each type. This means that if the method is elaborated on the host or the target for a specific type then coherence should ensure that the same instance is selected.

3.2.1 Overlapping Instances and quotation

When overlapping instances are present, it is possible to make GHC select the wrong instance as the example in Figure 1 demonstrates.

In module *A* a type class *Show* is defined with an instance for *Show Int*. The implementation returns a string indicating which instance was selected. In modules *B*1 and *B*2 which both depend on *A* (but not on each other) two overlapping instances are defined. In *B*1 a general instance is defined for [*a*] whilst in *B*2 a specific instance for [*Int*]. Then in *B*1 the code value *qshowList* is created which quotes *show* at the specific type [*Int*] → *String*. *show* uses the instance for [*a*] defined in *B*2 and the instance for *Int* defined in *A*.

```
module A where
class Show a where
  show :: a → String

module B1 where
import A
instance Show [a] where
  show _ = "list"
qshowList :: Code ([Int] → String)
qshowList = ⟦show⟧

module B2 where
import A
instance {-# OVERLAPPING #-}
  Show [Int] where
  show _ = "list2"

module C where
import A
import B1
import B2
main = $(qshowList) [1 :: Int]
```

Figure 1. A library which demonstrates the interaction of overlapping instances and quotation.

The semantics of overlapping instances [10] are that when there are two competing instances then the more specific of the two instances is selected. In using *show* at type [*Int*] in *C* the constraint solver should choose the instance defined in *B*2 as [*Int*] is more specific than [*a*]. However, since the type of *qshowList* does not mention anything about the class *Show*, its meaning should be fully determined when it is quoted in *B*1. Specifically, *qshowList* should use the instance defined in *B*1 rather than *B*2. Therefore, the generated code should be *show* @[*Int*] ($*dShow* [*a*] @*Int*) when *qshowList* is spliced into *C*.

By evaluating *main* it can be observed the instance selected when running *qshowList* is in fact the instance from *B*2 which should be impossible because the splice should have used the instance in *B*1. This is a symptom of the fact that the elaborated type information is not serialised.

3.2.2 Persistence for type class dictionaries

Implicit type class evidence may lead to stage errors if handled naively but special properties of dictionaries can be exploited so that they can always be persisted to a future stage. If the instance is already fully resolved, as in the example such as *qshowInt*, then the evidence is a top-level function. This can be persisted using path-based persistence. If the instance is not yet resolved then it is tempting to want to use serialisation-based persistence in order to persist the dictionary. This approach is doomed to failure because type class dictionaries usually contain functions which cannot be lifted using serialisation-based persistence. For example, the *Show a* dictionary contains a function *show* :: *a* → *String*.

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

Dictionaries however can always be persisted by observing that eventually before the splice is run the *Show a* constraint must be solved. When it is solved it will be solved by a combination of top-level functions which can all be serialised using path-based persistence.

Some special logic to perform this needs to be implemented in the compiler as the native *Lift* type class is not aware of these special rules surrounding dictionaries. One easy implementation strategy is to create a representation of each instance as it is defined. The constraint solver can suitably provide either the instance or its representation depending on whether the evidence is used inside a quotation.

3.3 Implicit Parameters

Finally, implicit parameters [7] also exhibit the same problems as type classes: if a quoted function has an implicit parameter then the implicit argument is not persisted in the representation. An implicit parameter is quite similar to a type class constraint but can be bound and applied locally. Implicit parameters are inferred by their names and not by their types. Since the evidence can not be inferred, it must be stored in the representation explicitly so that it can be used at the splice point.

A named implicit argument is specified by a special constraint. It is discharged by a let binding. For example, below is a version of *sort* which takes the sorting predicate implicitly rather than from *Ord* class as the normal *sort* function.

```
sort :: (?cmp :: a → a → Ordering) ⇒ [a] → [a]
```

The constraint brings into scope the variable ?*cmp* which can be used as normal in the body of *sort*.

```
sort = sortBy ?cmp
```

In order to discharge an implicit argument the argument is let-bound to a variable in an outer-scope. The value is then passed by the compiler to the function with the implicit parameter. For instance *sortInt* can be defined using *compare* :: *Int* → *Int* → *Ordering* as follows:

```
sortInt :: [Int] → [Int]
sortInt = let ?cmp = compare in sort
```

Implicit parameters do not interact well with quotations. In *sortInt*, the type of *sort* must be [*Int*] → [*Int*] and thus the type of the quote is *Code* ([*Int*] → [*Int*]).

```
qsortInt :: Code ([Int] → [Int])
qsortInt = let ?cmp = compare in ⟦sort⟧
```

However things go badly wrong when *qsortInt* is spliced.

```
sortInt' = $(qsortInt)
```

The splice fails with an error about an implicit parameter not being bound despite the fact that the type of *qsortInt* mentioning nothing about implicit parameters.

```
Unbound implicit parameter
  (?cmp :: Int -> Int -> Ordering)
```

Worse still, the implicit parameter can be dynamically bound at the splice site.

$$sortInt'' = \textbf{let}\ ?cmp = flip\ compare\ \textbf{in}\ \$(qsortInt)$$

Imagine if another program uses an implicit with the same type and same name. If *qsortInt* is spliced into this program then this unrelated comparison function will be used instead of the one bound when *qsortInt* was defined. More seriously, since the quoted value comes from an external library there is no indication that the implementation uses implicit parameters which would lead to a very hard to find bug.

3.3.1 Persistence for Implicit Parameters

Implicit parameters have to be persisted in the same manner as normal values. This means where there is a persistence error the implicit parameter must be liftable and the reference to the variable is replaced by a splice and lift combination.

Implicit parameters can be defined in an ad-hoc basic which means there are no special conditions which can be exploited to automatically create liftings. They are much more like explicit functions than type classes. This means the treatment of implicit parameters must be similar to the treatment of explicit parameters.

For example, *add* uses an implicit parameter of type *a*, this means that when *add* is quoted a *Lift a* constraint is emitted as *x* is implicitly applied to *add* inside the quotation.

$$add :: (Num\ a, ?x :: a) \Rightarrow a \to a$$
$$add\ y = ?x + y$$
$$qadd :: (Lift\ a, Num\ a) \Rightarrow Code\ (a \to a)$$
$$qadd = \textbf{let}\ ?x = 1\ \textbf{in}\ [\![add]\!]$$

3.4 Solution

In this section the problem of missing evidence manifested when interacting with types, type classes and implicit parameters. It should be well motivated now that the representation for a quotation must retain type information because the meaning of a quotation depends on the context where the quotation occurs. Further to this we also saw how the principles of cross-stage persistence had to be extended to account for the introduction of new implicit evidence.

4 Type System

In order to formally explain the interaction between elaboration and staging we introduce $\lambda_{\Uparrow}$, a simple lambda calculus with elaboration and explicit stage annotations, by providing its syntax, static semantics and dynamic semantics.

4.1 Syntax

The syntax (Figure 2) for $\lambda_{\Uparrow}$ shows that it is an explicitly typed language that the elaborator produces from the source by inferring type annotations.

The language e is a core language of elaborated expressions with explicit type annotations and multi-stage constructs. The constructs for constants, variables, abstraction, application, and types are all standard. This is extended with a special evidence application form ($e\ \iota$) that is used to apply a function to evidence introduced by the elaboration procedure. Note that there is no corresponding evidence abstraction form, as an evidence is just a term in the calculus. Stage annotations are introduced with quotation ($[\![\ e\ ]\!]$) and splice ($\$(e)$) constructs. The language also contains two special operations for lifting values ($\uparrow e$) and types ($\Uparrow \tau$).

Evidence (ι) is used to represent elaborated information for dictionaries and implicit parameters. A dictionary evidence ($D\ x$) is provided in order to solve a type class where x is a dictionary variable. This is in contrast to the evidence for implicit parameters ($I\ e$) which can be any normal expression.

The syntax for types (τ) is mostly standard. The non-standard additions are representation types for expressions ($Code\ \tau$) and types ($CodeT$), and a splice operator ($\$(e)$) which can appear in types. The type splice is used in conjunction with $\Uparrow$ and only introduced by elaboration.

4.2 Typing Rules

The typing rules are given for expressions (Figure 3) and evidence (Figure 4), as are the type formation rules (Figure 5). These rules focus only on the forms that are specific to $\lambda_{\Uparrow}$, the others are standard and can be found in Appendix **??**.

The $\Gamma \vdash^n e : \tau$ judgement dictates that an expression e has type τ at level n. The level is increased when inside a bracket and decreased when inside a splice. The level at the top-level is 0. Likewise the judgements for evidence and types are also indexed by a level. $\Gamma \vdash^n_{ty} \tau$ indicates that τ is well-formed in context Γ at level n, types are unikinded. $\Gamma \vdash^n_{ev} \iota : \tau$ indicates that evidence ι has type τ at level n.

The environment Γ is heterogeneous with four different categories. Γ_e is the expression environment and maps a variable to its type and its level. Γ_τ is the type environment and maps a type variable to its level. Γ_{TL} maps a constant to its type. Finally, Γ_ι maps a dictionary variable to its type.

For the most part, the typing rules for expressions (Figure 3) are standard. The most interesting rule is E_CSP rule that describes cross-stage persistence for variables: they can be used in stages after they are defined. If a variable $x : \tau$ is introduced at level k then it can be used at level n as long as $k < n$ and τ is liftable. The E_Var rule separately specifies that a variable can be freely used at the level it is bound. The E_Var_TopLevel rule specifies that top-level identifiers can be likewise persisted to any level. The rules for abstraction and application are standard. Evidence application is

e	::=	elaborated expressions	ι	::=	evidence	τ	::=	types
	$\mid K$	constants		$\mid D\ x$	dictionary		$\mid H$	type constants
	$\mid x$	variables		$\mid I\ e$	implicit parameter		$\mid a$	variables
	$\mid \lambda x : \tau.\ e \mid e_1\ e_2$	abstraction/application					$\mid \tau_1 \rightarrow \tau_2$	abstraction
	$\mid \Lambda a.\ e \mid e\ \tau$	type abstraction/application					$\mid \forall a.\ \tau$	type polymorphism
	$\mid e\ \iota$	evidence application					$\mid Code\ \tau$	representation
	$\mid [\![\ e\ ]\!] \mid \$(e)$	quotation/splice					$\mid CodeT$	type representation
	$\mid \uparrow e \mid \Uparrow \tau$	lift/lift types					$\mid \$(e)$	type splice

Figure 2. Syntax of $\lambda_{\Uparrow}$

given by E_EvApp, and follows the rule for application except that evidence is looked up in its own environment. The rules E_Quote and E_Splice increase and decrease levels respectively. E_Run is used to run a quotation at the top-level. Finally, the rules for E_Lift and E_Lift_Type describe how the lift operation can yield the representation of a type without modifying the current level. Type constants and representations have their liftings introduced by the elaborator.

The purpose of the evidence typing rules (Figure 4) is so that it is syntactically obvious where the elaborator has inserted evidence. The Ev_Dict rule looks up the dictionary variable in the Γ_ι environment whilst the Ev_Imp rule types $I\ e$ just as the expression e is typed.

Most of the type formation rules (Figure 5) are also standard. Cross-stage persistence considerations in the T_CSP rule follow the pattern for expressions. The rules T_Code, T_Ty_Splice, and T_CodeT are unsurprising.

4.3 Transformation Rules

Once the elaborator has produced a $\lambda_{\Uparrow}$ term the cross-stage persistence elaboration phase (Figure 6) normalises the expression. This ensures that each variable is only used at the level it is bound and therefore contains no persistence errors.

The relation $\Delta \vdash e \rightsquigarrow^n e'$ indicates that in a context Δ which maps variables to the level they are defined, e evaluates to e' at level n. The normal forms for this relation are expressions where every variable is used at the level it is defined: no persistence errors remain. This corresponds to removing the E_CSP and E_EvApp rules (Figure 3), T_CSP (Figure 5) and the evidence typing rules (Figure 4).

The primary workhorse is the E_Var_CSP rule which dictates how to elaborate a variable x that is used at a level after it is defined. The level in this judgement is used to determine how much an occurrence of a variable needs to be lifted. For instance, if a variable is defined at level 3 and used at level 5 then the E_Var_CSP rule means that x will be evaluated to $\uparrow_3^5 x$ which is equal to $\$(\uparrow\$(\uparrow x)))$. A similar lifting for types is also required. These are given by:

$$\uparrow_k^n e = \begin{cases} e & k = n \\ \uparrow_{k+1}^n \$(\uparrow e) & k < n \end{cases} \qquad \Uparrow_k^n e = \begin{cases} e & k = n \\ \Uparrow_{k+1}^n \$(\Uparrow e) & k < n \end{cases}$$

The level of a variable is corrected by inserting a combination of $\uparrow$ and $\$(..)$. The $\uparrow$ is guaranteed to be well-typed due to the E_CSP rule which ensures that $Lift\ \tau$.

Just like the typing judgements moving inside a quote increases the level by one and moving inside a splice decreases the level by one. The E_Lam_CSP and E_TLam_CSP rules extend the context because they introduce new type variables. The E_Lam_CSP also delegates to the $\Delta \vdash \tau \rightsquigarrow_{ty}^n \tau'$ judgement (Figure 7) so that the persistence errors in types are also solved.

The cross-stage persistence rules for evidence (Figure 8) show that implicit evidence is elaborated in the same manner as normal expressions, a fact expressed by the definition of Ev_Imp_CSP. Evidence for dictionary variables is contained in the Δ_ι environment.

Example Recalling the simple example of quoting the identity function, in $\lambda_{\Uparrow}$ the following is well-typed:

$$\{id : \forall a.a \rightarrow a\}_{TL} \vdash^0 \Lambda a.[\![\ id\ a\ ]\!] : \forall a.\, Code\ (a \rightarrow a)$$

The type variable a is used at level 1 but bound at level 0. The T_CSP rule allows this usage because $k = 0$ and $n = 1$. This usage is corrected by $\rightsquigarrow^n$ and in particular T_Var_CSP replaces the usage of a with $\Uparrow_0^1$ which equates to $\$(\Uparrow a)$. The level correct expression is:

$$\{id : \forall a.a \rightarrow a\}_{TL} \vdash^0 \Lambda a.[\![\ id\ \$(\Uparrow a)\ ]\!] : \forall a.\, Code\ (a \rightarrow a)$$

At this point, the variable a is bound at the level it is used. Therefore, when the expression is applied to a type, it is substituted into the splice as desired.

4.4 Dynamic Semantics

The second set of dynamics semantics mostly standard and so only the unusual ones are discussed (Figure 10). The relation $e \rightsquigarrow e'$ indicates that the expression e evaluates in one-step to e'. $\tau \rightsquigarrow_{ty} \tau'$ indicates that the type τ evaluates to τ' in one step. The evaluation rules for $\uparrow$ show how $\uparrow$ is implemented for the two base types Nat and $Bool$. The evaluation of types is necessary in order to evaluate the type splices which are introduced by $\Delta \vdash \tau \rightsquigarrow_{ty}^n \tau'$. The evaluation rules are simple but are necessary to ensure that type variables can be persisted but only if they are instantiated to a known type. In particular considering the

$$\boxed{\Gamma \vdash^n e : \tau}$$

$$\frac{x : (\tau, k) \in \Gamma_e \qquad k < n \qquad \mathit{Lift}\ \tau}{\Gamma \vdash^n x : \tau}\text{E\_CSP} \qquad \frac{x : (\tau, n) \in \Gamma_e}{\Gamma \vdash^n x : \tau}\text{E\_Var} \qquad \frac{K : \tau \in \Gamma_{TL}}{\Gamma \vdash^n K : \tau}\text{E\_Var\_TopLevel}$$

$$\frac{\Gamma_e, x : (\tau, n) \vdash^n e : \sigma}{\Gamma \vdash^n \lambda x : \tau . e : \tau \to \sigma}\text{E\_Abs} \qquad \frac{\Gamma_\tau, a : n \vdash^n e : \tau}{\Gamma \vdash^n \Lambda a . e : \forall a . \tau}\text{E\_TAbs} \qquad \frac{\Gamma \vdash^n e : \tau \to \sigma \qquad \Gamma \vdash^n_{ev} \iota : \tau}{\Gamma \vdash^n e\ \iota : \sigma}\text{E\_EvApp}$$

$$\frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^n [\![e]\!] : \mathit{Code}\ \tau}\text{E\_Quote} \qquad \frac{\Gamma \vdash^{n-1} e : \mathit{Code}\ \tau}{\Gamma \vdash^n \$(e) : \tau}\text{E\_Splice} \qquad \frac{\Gamma \vdash^0 e : \mathit{Code}\ \tau}{\Gamma \vdash^0 !(e) : \tau}\text{E\_Run} \qquad \frac{\Gamma \vdash^n e : \tau \qquad \mathit{Lift}\ \tau}{\Gamma \vdash^n \uparrow e : \mathit{Code}\ \tau}\text{E\_Lift}$$

$$\frac{\Gamma \vdash^n_{ty} \tau}{\Gamma \vdash^n \Uparrow \tau : \mathit{CodeT}}\text{E\_Lift\_Type}$$

Figure 3. Core Expression typing

$$\boxed{\Gamma \vdash^n_{ev} \iota : \tau}$$

$$\frac{x : \tau \in \Gamma_\iota}{\Gamma \vdash^n_{ev} D\ x : \tau}\text{Ev\_Dict} \qquad \frac{\Gamma \vdash^n e : \tau}{\Gamma \vdash^n_{ev} I\ e : \tau}\text{Ev\_Imp}$$

Figure 4. Evidence Typing

$$\boxed{\Gamma \vdash^n_{ty} \tau}$$

$$\frac{\tau : k \in \Gamma_\tau \qquad k < n}{\Gamma \vdash^n_{ty} \tau}\text{T\_CSP} \qquad \frac{\tau : n \in \Gamma_\tau}{\Gamma \vdash^n_{ty} \tau}\text{T\_Var}$$

$$\frac{\Gamma_\tau, a : n \vdash^n_{ty} \tau}{\Gamma \vdash^n_{ty} \forall a . \tau}\text{T\_For\_All} \qquad \frac{\Gamma \vdash^n_{ty} \tau}{\Gamma \vdash^n_{ty} \mathit{Code}\ \tau}\text{T\_Code}$$

$$\frac{\Gamma \vdash^n e : \mathit{CodeT}}{\Gamma \vdash^n_{ty} \$(e)}\text{T\_Ty\_Splice} \qquad \frac{}{\Gamma \vdash^n_{ty} \mathit{CodeT}}\text{T\_CodeT}$$

Figure 5. Type formation

term $(\Lambda a.[\![\ id\ a\ ]\!])\ \mathit{Bool}$ then the persistence elaboration will elaborate this to $(\Lambda a.[\![\ id\ \$(\Uparrow a)\ ]\!])\ \mathit{Bool}$ which evaluates to $[\![\ id\ \$(\Uparrow \mathit{Bool})\ ]\!]$ which when spliced $\$[\![\ id\ \$(\Uparrow \mathit{Bool})\ ]\!]$ evaluates to *id Bool*. This extra indirection via type splices and $\Uparrow$ is necessary because we don't want to end up in a situation with an unbound variable *a* being applied to *id*.

It should also be noted that there is no congruence rule for evaluating inside a quotation. The D_E_Splice eliminates a quotation when it appears directly inside a splice. At this point, the inside of a quotation can also be evaluated. On the other hand, evaluation is permitted inside splices and this corresponds to the intuition that splices drive evaluation and quotations stop evaluation.

5 Implementation

The current implementation of typed quotations is based on a representation of renamed expressions [15]. This means that when the representation is inserted into a program by splicing, it has to be typechecked again, causing the problems described with implicits (Section 3).

The representation of terms needs to contain typechecked information therefore needs to be one of the representations that GHC uses after typechecking [8]. There are two obvious candidates, *LHsExpr GhcTc* terms which are fully-elaborated source syntax, or *CoreExpr* terms which are the representation terms for the Core language. In our implementation we choose to use *CoreExpr*s.

A Core expression is chosen as it is the simplest representation which retains the type information of a typechecked expression. It would also be possible to represent an expression as STG or Cmm but a Core expression is the perfect choice as it can be inserted at the start of the Core optimisation pipeline. The optimiser can then use its definition as normal during optimisation. The advantages to the Core representation are numerous:

- Core expressions are explicitly typed so after they are serialised the type information is fixed.
- Core expressions are already serialised and loaded by the compiler in order to support inlining definitions across modules. The same machinery can be reused for representing typed quotations.
- There is no redundant work performed typechecking expressions which have already been typechecked.
- Program compilation is faster. Modules which use a lot of quotations generate large programs as the existing representation type needs to closely mirror the source language as it can be inspected. These big terms programs can take a long time to compile.

$$\boxed{\Delta \vdash e \rightsquigarrow^{n} e'}$$

$$\frac{\Delta_e(x) = k}{\Delta_e \vdash x \rightsquigarrow^{n} \uparrow^{n}_{k} x}\text{E\_Var\_CSP} \qquad \frac{\Delta_e, x : n \vdash e \rightsquigarrow e' \qquad \ldots \vdash \tau \rightsquigarrow^{n}_{ty} \tau'}{\Delta \vdash \lambda x : \tau . e \rightsquigarrow^{n} \lambda x : \tau' . e'}\text{E\_Lam\_CSP} \qquad \frac{\Delta \vdash e \rightsquigarrow^{n+1} e'}{\Delta \vdash [\![e]\!] \rightsquigarrow^{n} [\![e']\!]}\text{E\_Code\_CSP}$$

$$\frac{\Delta \vdash e \rightsquigarrow^{n-1} e'}{\Delta \vdash \$(e) \rightsquigarrow^{n} \$(e')}\text{E\_Splice\_CSP} \qquad \frac{\Delta_\tau, x : n \vdash e \rightsquigarrow^{n} e'}{\Delta \vdash \Lambda x . e \rightsquigarrow^{n} \Lambda x . e'}\text{E\_TLam\_CSP} \qquad \frac{\Delta \vdash e \rightsquigarrow^{n} e' \qquad \Delta \vdash \iota \rightsquigarrow^{n}_{ev} e_\iota}{\Delta \vdash e\iota \rightsquigarrow^{n} e' e_\iota}\text{E\_EvApp\_CSP}$$

Figure 6. Cross-Stage Persistence Expressions

$$\boxed{\Delta \vdash \tau \rightsquigarrow^{n}_{ty} \tau'}$$

$$\frac{\Delta_\tau(x) = k}{\Delta \vdash x \rightsquigarrow^{n}_{ty} \Uparrow^{n}_{k} x}\text{T\_Var\_CSP}$$

Figure 7. Cross-Stage Persistence Types

$$\boxed{\Delta \vdash \iota \rightsquigarrow^{n}_{ev} e}$$

$$\frac{\Delta \vdash e \rightsquigarrow^{n} e'}{\Delta \vdash I\ e \rightsquigarrow^{n}_{ev} e'}\text{Ev\_Imp} \qquad \frac{\Delta_\iota(x) = e}{\Delta \vdash D\ x \rightsquigarrow^{n}_{ev} e}\text{Ev\_Dict}$$

Figure 8. Cross-Stage Persistence Evidence

$$\boxed{\mathit{Lift}\ \tau}$$

$$\frac{}{\mathit{Lift\ Nat}}\text{Lift\_Nat} \qquad \frac{}{\mathit{Lift\ Bool}}\text{Lift\_Bool}$$

Figure 9. Definition of *Lift*

- There is no confusion between a user-written program and a generated program. The compiler tries to keep track of what is "generated" but there are several bugs (#14838, #16169) in GHC where warnings are reported.
- Generated programs are optimised like normal user written programs so new optimisation opportunities can be taken advantage of. This is important as distant fragments of code come together to unveil simple optimisation opportunities such as β-reduction or case elimination. The automatic cleanup is important for any explicit staging framework. Lower level representations start to introduce undesirable machine dependence and reduce other optimisation opportunities.

In the new representation there are new kinds of values which are introduced by the elaborator. Therefore the new representation forces you to take care of cross-stage persistence as if you do not, scope extrusion errors are very easy to introduce during elaboration. In particular, the elaborated expression has to be checked for stage errors after elaboration has finished. For the three cases that were considered in Section 3 this step fixes the issues which arise there.

The primary disadvantage of using Core terms as the representation form is that users no longer have the ability to inspect quoted expressions. It is a defining feature of untyped Template Haskell to be able to deconstruct and construct expressions by inspecting an modifying the AST. In general, this is unsafe as there are no guarantees that generated programs will be well-typed. In typed multi-stage programming the term representation is almost always opaque so this consequence is not surprising nor undesirable.

We now turn to the precise details of our implementation which may be useful for other implementors.

5.1 Implementation Strategy

The first question that has to be answered is the precise representations of expressions. Storing a single Core expression is not enough as it doesn't account for nested splices. The internal representation contains two environments for storing delayed splices as all splices are run when the top-level splice is invoked. The expression is serialised and stored in a temporary file pointed to by the *FilePath*.

```
data CoreRep = CoreRep { tenv :: [(Int, CoreTypeRep)]
                       , eenv :: [(Int, CoreRep)]
                       , expr :: FilePath }
```

Using an untyped internal representation is much easier as the representations have to be stored in environments. The typed representation is a newtype wrapper around the untyped representation with a phantom type parameter.

```
newtype Code a = Code { untype :: CoreRep }
```

Quoting an expression of type e results in an expression of type *Code e*. For example, $[\![True]\!] :: Code\ Bool$ would be represented by:

```
qtrue = Code (CoreRep [] [] "/tmp/file")
```

The two environments in the *CoreRep* are for type splices and value splices respectively. The value *qtrue* uses neither so they are empty in this instance. The expression *True* is turned into a Core expression by desugaring and serialised to a temporary file.

$$\boxed{e \rightsquigarrow u}$$

$$\frac{}{!([\![\ e\ ]\!]) \rightsquigarrow e}\text{D\_E\_Run} \qquad \frac{e \rightsquigarrow e'}{!(e) \rightsquigarrow !(e')}\text{D\_E\_Run'} \qquad \frac{}{\$([\![\ e\ ]\!]) \rightsquigarrow e}\text{D\_E\_Splice} \qquad \frac{e \rightsquigarrow e'}{\$(e) \rightsquigarrow \$(e')}\text{D\_E\_Splice'}$$

$$\frac{}{\uparrow True \rightsquigarrow [\![\ True\ ]\!]}\text{D\_E\_True} \qquad \frac{}{\uparrow False \rightsquigarrow [\![\ False\ ]\!]}\text{D\_E\_False} \qquad \frac{}{\uparrow Z \rightsquigarrow [\![\ Z\ ]\!]}\text{D\_E\_Z} \qquad \frac{}{\uparrow (S\ n) \rightsquigarrow [\![\ S\ \$(\uparrow n)\ ]\!]}\text{D\_E\_S}$$

Figure 10. Dynamic Semantics

$$\boxed{\tau \rightsquigarrow_{ty} \sigma}$$

$$\frac{e \rightsquigarrow e'}{\$(e) \rightsquigarrow_{ty} \$(e')}\text{D\_T\_Splice} \qquad \frac{\tau \rightsquigarrow_{ty} \tau'}{Code\ \tau \rightsquigarrow_{ty} Code\ \tau'}\text{D\_T\_Code} \qquad \frac{}{\$(\Uparrow \tau) \rightsquigarrow_{ty} \tau}\text{D\_T\_Splice\_Eval}$$

Figure 11. Dynamic Semantics for Types

Splice Points The environments are mappings from integers to representations. When a quotation uses a splice explicitly or implicitly this is replaced with a *splice point* which is representated as a unique integer. The result of evaluating the splice will be inserted after it has been evaluated at the splice point. At quotation time, it is not yet possible to run the nested splices. If it were, then the values they evaluated to could be inserted into the Core expression before serialisation. However, Template Haskell only allows running splices in another module from which they are defined so all nested splices are delayed until the top-level splice is executed. The environments are used to delay this evaluation until the top-level splice is used in another module.

```
qfalse :: Code Bool
qfalse = [|| not $(qtrue) ||]
```

qfalse uses an explicit value splice which means the environment is populated with one pair for the single splice. 213131 is the unique integer representing the splice point.

```
qfalse = Code (CoreRep [ ] [(213131, qtrue)]
  "/tmp/file2")
```

Once the top-level splice is evaluated to a *Code a* then the Core expression stored in the file is loaded and inserted into the program. Whilst the expression is being loaded the splice points are replaced by the necessary evaluated expressions. This is all kept track of by extending local environments in the interface file loader.

The result of loading a single *Code t* is a Core expression which when evaluated produces a result of type *t*.

Loading Expressions Splices are run in the desugarer. This is the natural implementation as the representation is Core expressions which are produced by desugaring. Top-level splices are evaluated using the bytecode interpreter and their value made available in the program by dynamically loading the result back into the runtime. After the result is loaded, the Core expressions are treated as normal in the rest of the compilation pipeline.

In order to load a Core expression the serialisation is read from where it is stored. This produces an *IfaceExpr* which is deserialised to produce the necessary *CoreExpr*. Whilst deserialising the *IfaceExpr* any splice points are replaced by the result of performing the splice. The result of performing a splice is an expression of type *CoreRep* which could itself contain splices, therefore, in order to load it, the same method is called recursively interpreting the *CoreRep* as a *CoreExpr*.

It is necessary to delay running and typechecking the nested splices so that quotations which capture variables are deserialised in the right scope and their variables bound appropriately. The simplest example being a function which splices the result of quoting a lambda bound variable. The quotation $[\![x]\!]$ must be loaded in a context where x is bound.

```
lam = [|| λx → $([|| x ||]) ||]
```

This means that the desugarer calls the interface file loader which calls the desugarer in a mutually recursive knot.

Scope Extrusion Scope extrusion is when a piece of generated code contains reference to a variable that is not in scope. In systems such as MetaOCaml, the scope extrusion check is performed explicitly at splice time [5]. Any unbound variable is detected and an error is raised.

When a Core expression is deserialised from an *IfaceExpr* a similar check has already been performed because locally bound names are stored as strings. Deserialisation resolves the scope of these strings and raises an error on extrusion.

Implicit Type Splices Due to the elaboration step implemented in GHC most type variables are instantiated implicitly. This means that the situations where there will be stage errors are not obvious from inspecting the source syntax before typechecking. Stage errors can only be identified and corrected after all implicit arguments have been resolved.

The annealing phase that corrects implicit stage errors can't be implemented naively because a term which has a persistence error will need to be given a different type. If this is not observed until after type checking has finished then modifying the type of a single definition will cause other inconsistencies. At least in the manner that GHC is currently architectured, incrementally changing a type of a subexpression is not possible.

Before unification is finished which metavariables will be quantified over and which instantiated is not known. To resolve this it is necessary to be pessimistic. Whenever a metavariable is created inside a bracket context it might be the source of a stage error. As a result, a new constraint is emitted which indicates the metavariable should be an instance of the *LiftT* class. The evidence variable for this constraint is stored associated with the metavariable. *LiftT* constraints are solved automatically for every type which is not a type variable so the only situation where the constraint will propagate to the type as it is failed to be solved is precisely when it is a simple type variable which will be quantified over.

During zonking, which is when metavariables are instantiated, for any skolem variables which appear inside of brackets (remember that Typed Template Haskell generates monomorphic Haskell programs so all occurrences of type variables must be bound at the top-level) the evidence variable is recovered. The type variable is replaced by a splice point and a new typed splice is recorded in the environment

This means that if the *LiftT* constraint is solved, then the constraint solver will not abstract over the *LiftT* constraint. However, if the type variable is never instantiated to a concrete type then it will be abstracted over as the constraint solver will not be able to solve the constraint.

***Solving** LiftT* The *LiftT* class is solved automatically for all types which are not type variables. For compound types, which contain type variables, the instance is solved recursively and constraints emitted for any free type variables. The solver creates a representation form which combines together the recursive evidence into a representation for types based on GHC's internal representation. This can be serialised and stored in the same manner as Core expressions.

Explicit Type Variables Persistence errors for explicitly mentioned type variables are solved in the same manner as value-level persistence errors. Type variables can appear explicitly in explicit type applications [3] and expression type signatures.

$$crossTy :: \forall a.Code\ (a \to Int) \qquad crossTy' :: \forall a.Code\ Int$$
$$crossTy = [\![const\ 0 :: a \to Int]\!] \qquad crossTy' = [\![get\ @a]\!]$$

The a is bound at stage 0 but used in stage 1. When the type is kind-checked the variable case consults the level of a and discovers that there is a stage error. This causes the implicit splice to be inserted which corrects the usage stage.

$$crossTy :: \forall a.LiftT\ a \Rightarrow Code\ (a \to Int)$$
$$crossTy = [\![const\ 0 :: \$(liftT\ @a) \to Int]\!]$$

The evidence for type splices is stored in the type environment and loaded similarly to the evidence for value splices.

***Modifications to** Lift* The *Lift* type class is fundamental in the implementation of Template Haskell in order to support cross-stage persistence. Unfortunately it has long been tied to the untyped term representation that Template Haskell uses. Users who have implemented *Lift* have been free to do so by explicitly using the Template Haskell combinators. These instances will not work with the modified backend which uses Core expressions as the serialisation form.

Since GHC 8.0.1, the recommended way to implement *Lift* has been with the `DeriveLift` extension. The current implemention still works by explicitly generating the untyped representation terms which makes it harder to modify to support the Core expression representation we have described. In the forthcoming GHC 8.10 release, `DeriveLift` will be implemented in terms of quotation brackets [19] which means that any derived instance will work seamlessly with this modified backend.

5.2 Other Considerations

Using type signatures In section 1 we introduced the simple *show* · *read* example and argued that the contextual type information from quoting should be retained in the representation. Another approach might be to insert a type signature at every typed splice. The generated code would then be:

```
(show :: Int -> String) . (read :: String -> Int)
```

The type variables still need to be persisted as described in Section 5.1. This would integrate more easily with the existing Template Haskell implementation. However, it would not work in general for the examples involving type classes (Section 3.2) nor implicit parameters (Section 3.3) because the type alone does not provide enough context to infer the correct evidence.

A kind indexed type representation An obvious question to ask is why the *CodeType* representation is not kind-indexed in a similar manner to how *Code* is type indexed.

The reason for this is that in this current work no additional syntax is added to the language to allow the user to create or splice the representation of types. In particular the lack of splicing for *CodeType* in the source language means that the only way which they can be used is in implicit lifting. It is left as future work to implement a kinded type representation and syntax for splicing kinded type representations into typed quotations.

***Whether** Typeable?* One may wonder why we are not using *Typeable*: a type indexed representation of types implemented in GHC [11]. The *Typeable t* class implements one

method *typeRep* :: *TypeRep t*. *typeRep* @*t* is a value representation of the type *t*, the representation is of type *TypeRep t*.

Typeable is not appropriate as, despite giving a type representation at runtime, it cannot be converted into the representation of types used in this implementation. One could consider extending the *Typeable* interface to provide a suitable representation but it is simpler to create a representation with the exact properties needed for the implementation.

User-Written Core By choosing Core as the representation for terms it would be straightforward to add combinators to allow users to directly generate core terms to be inserted into their programs rather than using quotations. This would be unsafe but appealing to users who care about efficiency because Core is much lower-level than source Haskell.

6 Cloud Haskell

It is useful to compare the treatment of typed quotations to a different metaprogramming facility implemented in Haskell: Cloud Haskell [4]. Both approach the issues of persistence and polymorphism in a slightly different manner but are implemented independently.

Cloud Haskell is a distributed computing framework. One additional keyword, ***static***, enables expressions to be serialised and transmitted across the cluster. The type of ***static*** *e* for a closed expression of monotype type τ is *StaticPtr* τ.

static 1 :: *StaticPtr Int*

In the case where $e :: \forall a_1 \ldots a_n.\tau$, the inferred type of of ***static*** *e* is $\forall a_1 \ldots a_n.Typeable\ (a_i) \Rightarrow StaticPtr\ \tau$. As in Template Haskell quotes, the free variables are floated outside of the ***static*** keyword. Further to this, all the type variables are constrained by *Typeable* constraints. Unlike Template Haskell, expressions involving constraints and implicit arguments are explicitly rejected, they are not floated outside of the *StaticPtr* constructor. Expressions which mention free value variables are also rejected. Cross-stage persistence is not implemented for values.

Representation *StaticPtr a* is an abstract data type with the following API:

```
deRefStaticPtr :: StaticPtr a → a
staticKey :: StaticPtr a → StaticKey
unsafeLookupStaticPtr :: StaticKey
                      → IO (Maybe (StaticPtr a))
```

The host can dereference the *StaticPtr* using *deRefStaticPtr* in order to retrieve the value referenced by the pointer. Otherwise, the reference can be transmitted over the network by transmitting the *StaticKey*. A client can retrieve the static pointer from the key using *unsafeLookupStaticPtr*.

A *StaticKey* is a pointer into a global shared map called the static pointer table. The table stores pointers to where the top level definitions are defined. It does not store serialised core expressions directly.

It is intended that all nodes are running the same executable and thus will maintain identical static pointer tables. The dereferencing step is unsafe, the guarantee provided by the API is only that if you lookup a *StaticKey* using *unsafeLookupStaticPtr* and specify the correct type *a* then the lookup will succeed. Any other behaviour is undefined. This includes errors involving dereferencing values at the incorrect type but also polymorphic values at a type at which they were not serialised.

We have previously argued that if you float the type variables outside the representation type constructor then you have to persist the type information in the representation. Static forms do not persist the type information in the representation resulting in the weaker guarantee described above. If we recall the problem with types from Section 3.1 it was caused by an ambiguity caused by splicing together programs. This potential for ambiguity isn't present when using static pointers because fragments can't be composed. When a *StaticPtr* is dereferenced its type must be provided explicitly and there is no further re-elaboration procedure.

The *Typeable* constraints ensure that the type of the static form is monomorphic and determined before it is placed into the static pointer table. The *Typeable* evidence is not used at all in the implementation to ensure that the dereferencing step is safe. The intention behind this is to not sacrifice any performance as the dereferencing steps happens at runtime. Clients can build safer APIs to perform a runtime typecheck in order to protect against undefined behaviour.

Serialising Programs The Cloud Haskell interface is intended to serialise closed expressions but is not compositional; there is no way in which to combine two static pointers together. For this reason, the common way to interact with static pointers is to define a simple DSL for building larger programs which contain static pointers. The most elegant example is in the static-closure[3] library. Static forms are used to construct the leaves by implementing cross-stage persistence for top-level functions and closed expressions.

Elaboration Evidence Functions which use type class constraints and implicit parameters are forbidden from appearing in static forms. This is commonly solved by defining wrapper data types which remove the constraint.

```
data Dict c = c ⇒ Dict

showInt :: Dict (Show a) → a → String
showInt Dict = show

showIntPtr :: Typeable a
           ⇒ StaticPtr (Dict (Show a) → a → String)
showIntPtr = static showInt
```

The *Dict* data type is used to store the constraint. When it is pattern matched on the constraint is available in the

[3] http://hackage.haskell.org/package/static-closure

definition. *showInt* no longer contains a type class constraint and so can appear as the argument to the static keyword.

7 Related Work

In this section the approaches taken by other typed multi-stage languages are discussed.

Haskell Winant et al. [22] generate well-typed core programs by embedding the Core language into Agda. Since they use a dependently typed language, Agda, as the host language only well-typed Core terms can be represented and hence only well-typed terms can be generated. This approach fits naturally into our idea of using the Core representation for terms. It would be interesting to see if their implementation could be simplified using our implementation and the recent developments in source plugins [12].

MetaOCaml The expression representation is an untyped syntax tree which is re-elaborated [5]. It is deemed that a representation which maintains types is more complicated and not as composable. This implementation strategy follows the original implementation by Calcagno et al. [1].

An alternative implementation [13] to MetaOCaml exists that uses Lambda, the OCaml equivalent to Haskell's Core language, in order to represent terms. The motivation for this implementation is in order to avoid repeatedly typechecking terms rather than correctness as in our case. We conjecture that because OCaml is a simpler language than Haskell so ambiguities relating to implicit arguments are not observed as frequently.

Dotty Dotty (Scala 3) also has support for typed staged programming [16]. Dotty chooses not to implement implicit cross-stage persistence for locally bound values (unlike Haskell) but does implement path-based persistence. Locally bound types are implicitly lifted using the *Liftable* type class which is similar to *Lift* and *LiftT*.

Scala has a raft of implicit arguments which are resolved in a type directed manner. Through experimentation is has been observed that cross-stage usage of implicits is rejected. Scala also sometimes requires that expressions are annotated with types, types used across stages are persisted implicitly using *Liftable* where possible. There is source syntax for quoting and splicing types into typed quotations. The representation form for quotations are Tasty [9] syntax trees which is the intermediate typed serialisable representation akin to Core and Lambda.

8 Conclusion

For a language with a complicated type inference process then it is necessary to retain information about the result of the inference process in the term representation. The further consequences of this decision on issues such as cross-stage persistence are quite subtle as you now have to also consider persisting the results of elaboration rather than just user-written programs.

In future work we would like to explore further extensions to Haskell's type system including recent work on quantified constraints and dependent Haskell. We predict that their interaction with quotation will be similarly interesting.

Acknowledgments

We thank Jamie Willis and Ryan Scott for comments on earlier drafts and implementations. This work has been supported by EPSRC grant number EP/S028129/1 on "SCOPE: Scoped Contextual Operations and Effects".

References

[1] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing multi-stage languages using ASTs, Gensym, and reflection. In *Proceedings of the 2nd international conference on Generative programming and component engineering (GPCE03)*. Association for Computing Machinery, 57–76. https://doi.org/10.5555/954186.954190

[2] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages (POPL '82)*. ACM.

[3] Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. 2016. Visible type application. In *European Symposium on Programming*. Springer, 229–254.

[4] Jeff Epstein, Andrew P. Black, and Simon Peyton Jones. 2011. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, NY, USA, 118–129. https://doi.org/10.1145/2034675.2034690

[5] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

[6] Oleg Kiselyov. 2018. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Foundations and Trends in Programming Languages* 5, 1 (2018), 1–101. https://doi.org/10.1561/2500000038

[7] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. ACM, New York, NY, USA, 108–118. https://doi.org/10.1145/325694.325708

[8] Simon Marlow and Simon Peyton Jones. 2012. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications*, Amy Brown and Greg Wilson (Eds.). Chapter 5. https://www.aosabook.org/en/ghc.html

[9] Martin Odersky, Eugene Burmako, and Dmytro Petrashko. 2016. A TASTY Alternative. (2016). http://infoscience.epfl.ch/record/226194

[10] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell Workshop*.

[11] Simon Peyton Jones, Stephanie Weirich, Richard A Eisenberg, and Dimitrios Vytiniotis. 2016. A reflection on types. In *A List of Successes That Can Change the World*. Springer, 292–317.

[12] Matthew Pickering, Nicolas Wu, and Boldizsár Németh. 2019. Working with Source Plugins. In *Proceedings of the 2019 ACM SIGPLAN Symposium on Haskell (Haskell '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3331545.3342599

[13] Evgeny Roubinchtein. 2015. *IR-MetaOCaml: (re)implementing MetaOCaml*. Master's thesis. University of British Columbia. https://doi.org/10.14288/1.0166800

[14] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language*

Design and Implementation (PLDI 2018). ACM, New York, NY, USA, 783–796. https://doi.org/10.1145/3192366.3192389

[15] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[16] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A Practical Unification of Multi-stage Programming and Macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2018)*. ACM, New York, NY, USA, 14–27. https://doi.org/10.1145/3278122.3278139

[17] Walid Taha. 2004. *A Gentle Introduction to Multi-stage Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

[18] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

[19] Alec Theriault. 2019. Levity polymorphic lift. GHC proposal. https://github.com/ghc-proposals/ghc-proposals/pull/209

[20] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

[21] Phillip Wadler and Stephen Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[22] Thomas Winant, Jesper Cockx, and Dominique Devriese. 2017. Expressive and Strongly Type-safe Code Generation. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP '17)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/3131851.3131872