A Specification for Typed Template Haskell

ANONYMOUS AUTHOR(S)

Multi-stage programming is a proven technique that provides predictable performance characteristics by controlling code generation. We propose a core semantics for Typed Template Haskell, an extension of Haskell that supports multi staged programming that interacts well with polymorphism and qualified types. Our semantics relates a declarative source language with qualified types to a core language based on the the polymorphic lambda calculus augmented with multi-stage constructs.

Additional Key Words and Phrases: staging, polymorphism, constraints

1 INTRODUCTION

Producing optimal code is a difficult task that is greatly assisted by *staging annotations*, a technique which has been extensively studied and implemented in a variety of languages [Kiselyov 2014; Rompf and Odersky 2010; Taha and Sheard 2000]. These annotations give programmers fine control by instructing the compiler to generate code in one stage of compilation that can be used in another.

The classic example of staging is the *power* function, where the value n^k can be efficiently computed for a fixed k by generating code where the required multiplications have been unrolled and inlined. The incarnation of staged programming in Typed Template Haskell[1] benefits from type classes, one of the distinguishing features of Haskell [Hall et al. 1996; Peyton Jones et al. 1997], allowing a definition that can be reused for any type that is qualified to be numeric:

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

Any value n::*Int* can be quoted to create $[\![\ n\ ]\!]$::*Code Int*, then spliced in the expression $\$(power\ 5\ [\![\ n\ ]\!])$ to generate $n * (n * (n * (n * (n * 1))))$. Thanks to type class polymorphism, this works when n has any fixed type that satisfies the *Num* interface, such as *Integer*, *Double* and countless other types.

It is somewhat surprising, then, that the following function fails to compile in the latest implementation of Typed Template Haskell in GHC 8.10.1:

$$
\begin{aligned}
&power5 :: Num\ a \Rightarrow a \to a \\
&power5\ n = \$(power\ 5\ [\![\ n\ ]\!])
\end{aligned}
$$

Currently, GHC complains that there is no instance for *Num a* available, which is strange because the type signature explicitly states that *Num a* may be assumed. But this is not the only problem with this simple example: in the definition of *power*, the constraint is used inside a quotation but is bound outside. As we will see, this is an ad-hoc decision that then leads to subtle inconsistencies.

This paper sets out to formally answer the question of how a language with polymorphism and qualified types should interact with a multi-stage programming language. As well as the fact that all code generation happens at compile time, these are features that distinguish Typed Template Haskell, which until now has had no formalism even though it is an extension already fully integrated into GHC. We extend previous work that looked at cross-stage persistence in this

[1] Typed Template Haskell is a variation of the Template Haskell [Sheard and Jones 2002] that adds types in the style of MetaML [Taha and Sheard 2000]. The abstract datatype *Code a* in this paper is a newtype wrapper around *Q* (*TExp a*). In GHC, typed quotes are implemented by `[|| ||]` and typed splices by `$$( )`, rather than $[\![\ \cdot\ ]\!]$ and $\$(\cdot)$. It was implemented in 2013 by Geoffrey Mainland (who also used *power* as a motivating example) under a proposal by Simon Peyton Jones.

2020. 2475-1421/2020/1-ART44 $15.00
https://doi.org/

setting [Pickering et al. 2019a] by more closely modelling the reality of the type system implemented in GHC. In particular the implications of compile-time code generation are considered, as are the impredicativity restriction and the interaction with instance definitions.

Providing a formalism for the interaction of polymorphism, qualified types, and multi-stage programming has important consequences for Typed Template Haskell and other future implementations that combine these features. This specification resolves the uncertainty about how it should interact with new language features. It has also been unclear how to fix a large number of bugs in the implementation because there is no precise semantics it is supposed to operate under. Presently these deficiencies hampered the adoption of meta-programming in Haskell as a trusted means of producing code with predictable performance characteristics. This untapped potential has been demonstrated to be beneficial in other staged systems in MetaML [Taha and Sheard 2000], Scala's LMS [Rompf and Odersky 2010], and MetaOCaml [Kiselyov 2014].

At first glance it may appear that constructing programs using only quotations and splices is restrictive. However, multi-stage programming is widely applicable to any domain in which statically known abstractions should be eliminated. Some particular examples include removing the overhead of using parser combinators [Jonnalagedda et al. 2014; Krishnaswami and Yallop 2019; Willis et al. 2020], generic programming [Yallop 2017] and effect handlers [Schuster et al. 2020]. Furthermore, there are certain interesting techniques and tricks which are useful when constructing multi-stage programs that can vastly improve performance [Kiselyov 2018; Taha 2004].

Following a brief introduction to staging with Typed Template Haskell (Section 2), this paper makes the following contributions:

- We demonstrate the problems that arise from the subtle interaction between qualified constraints and staging, which highlights the need for a clear specification (Section 3).
- We introduce a type system for a source language which models Typed Template Haskell and introduces a new constraint form (Section 4).
- We introduce an explicitly typed core language which is System F extended with multi-stage features (Section 5).
- We give the elaboration semantics from the source language into the core language (Section 6).

Future directions including type inference, the latest developments in supporting impredicativity and notes about an implementation are then discussed (Section 7). This work is put into the context of related work (Section 8), before finally concluding (Section 9).

2 BACKGROUND: MULTI-STAGED PROGRAMMING

This section describes the fundamental concepts of Typed Template Haskell as currently implemented in GHC. In these aspects, the current implementation coincides with our proposed specification of Typed Template Haskell, as we are excluding polymorphism and type classes. This serves as preparation for more interesting examples that are discussed later (Section 3).

Typed Template Haskell is an extension to Haskell which implements the two standard staging annotations of multi-stage programming: *quotes* and *splices*. An expression $e :: a$ can be quoted to generate the expression $[\![e]\!] :: \mathit{Code}\ a$. Conversely, an expression $c :: \mathit{Code}\ a$ can be spliced to extract the expression $\$(c)$. An expression of type $\mathit{Code}\ a$ is a representation of an expression of type a. Given these definitions, it may seem that quotes and splices can be used freely so long as the types align: well-typed problems don't go wrong, as the old adage says; but things are not so simple for staged programs. As well as being well-typed, a program must be well-staged, whereby the level of variables is considered.

The concept of a level is of fundamental importance in multi-stage programming. The *level* of an expression is an integer given by the number of quotes that surround it, minus the number of

splices: quotation increases the level and splicing decreases the level. Negative levels are evaluated at compile time, level 0 is evaluated at runtime and positive levels are future unevaluated stages.[2] The goal of designing a staged calculus is to ensure that the levels of the program can be evaluated in order so that an expression at a particular level can only be evaluated when all expressions it depends on at previous levels have been first evaluated.

In the simplest setting, a program is *well-staged* if each variable it mentions is used only at the level in which it is bound. Doing so in any other stage may simply be impossible, or at least require special attention. The next three example programs, *timely*, *hasty*, and *tardy*, are all well-typed, but only the first is well-staged.

Using a variable that was defined in the *same* stage is permitted. A variable can be introduced locally using a lambda abstraction inside a quotation and then used freely within that context:

$$timely :: Code\ (Int \to Int)$$
$$timely = [\![\ \lambda x \to x\ ]\!]$$

Of course, the variable is still subject to the usual scoping rules of abstraction.

Using a variable at a level *before* it is bound is problematic because at the point we wish to evaluate the prior stage, we will not yet know the value of the future stage variable and so the evaluation will get stuck:

$$hasty :: Code\ Int \to Int$$
$$hasty\ c = \$(c)$$

Here, we cannot splice c without knowing the concrete representation that c will be instantiated to. There is no recovery from this situation without violating the fact that lower levels must be fully evaluated before higher ones.

Using a variable at a stage *after* it is bound is problematic because the variable will not generally be in the environment. It may, for instance, be bound to a certain known value at compile time but no longer present at runtime.

$$tardy :: Int \to Code\ Int$$
$$tardy\ x = [\![\ x\ ]\!]$$

In contrast to the *hasty* example, the situation is not hopeless here: there are ways to support referencing previous-stage variables, which is called *cross-stage persistence* [Taha and Sheard 1997].

One option is to interpret a variable from a previous stage using a *lifting* construct which copies the current value of the variable into a future-stage representation. As the process of lifting is akin to serialisation, this can be achieved quite easily for base types such as strings and integers, but more complex types such as functions are problematic.

Another option is to use *path-based persistence*: for example, a top-level identifier defined in another module can be persisted, because we can assume that the other module has been separately compiled, so the top-level identifier is still available at the same location in future stages.

GHC currently implements the restrictions described above. For cross-stage persistence, it employs both lifting and path-based persistence. Top-level variables defined in another module can be used at any level. Lifting is restricted to types that are instances of a type class *Lift* and witnessed by a method

$$lift :: Lift\ a \Rightarrow a \to Code\ a$$

[2] Unlike MetaML [Taha and Sheard 1997], code generation in Typed Template Haskell is only supported at compile time, and therefore there is no **run** operation.

which notably excludes function types and other abstract types such as *IO*. An example such as *tardy* can then be viewed as being implicitly rewritten to

$$tardy' :: Int \to Code\ Int$$
$$tardy'\ x = [\![\ \$(lift\ x)\ ]\!]$$

in which the reference of x occurs at the same level as it is bound. For this reason, our formal languages introduced in Section 4 and Section 5 will consider path-based persistence, but not implicit lifting, as this is easy to add separately in the same way as GHC currently implements it.

Let us consider the example from the introduction once again:

$$power\ 0\ cn = [\![\ 1\ ]\!]$$
$$power\ k\ cn = [\![\ \$(cn) * \$(power\ (k-1)\ cn)\ ]\!]$$

This example makes use of quotes and splices. The references to the variables *power*, k and *cn* are all at the same level as their binding occurrence, as they occur within one splice and one quote. The static information is the exponent k and the run-time information is the base *cn*. Therefore by using the staged function the static information can be eliminated by partially evaluating the function at compile-time by using a top-level splice. The generated code does not mention the static information.

When using Typed Template Haskell, it is common to just deal with two stages, a compile-time stage and a run-time stage. The compile-time stage contains all the statically known information about the domain, the typing rules ensure that information from the run-time stage is not needed to evaluate the compile-time stage and then the program is partially evaluated at compile-time to evaluate the compile-time fragment to a value. It is a common misconception that Typed Template Haskell only supports two stages, but there is no such restriction.[3]

3 STAGING WITH TYPE CLASSES AND POLYMORPHISM

The examples in the previous section were simple demonstrations of the importance of considering levels in a well-staged program. This section discusses more complicated cases which involve class constraints, top-level splices, instance definitions and type variables. It is here where our proposed version of Typed Template Haskell deviates from GHC's current implementation. Therefore, for each example we will report how the program *should* behave and contrast it with how the current implementation in GHC behaves.

3.1 Constraints

Constraints introduced by type classes have the potential to cause staging errors: type classes are implemented by passing dictionaries as evidence, and the implicit use of these dictionaries must adhere to the level restrictions discussed in Section 2. This requires a careful treatment of the interaction between constraints and staging, which GHC currently does not handle correctly.

Example C1. Consider the use of a type class method inside a quotation, similarly to how *power* is used in *power5* in the introduction:

$$c_1 :: Show\ a \Rightarrow Code\ (a \to String)$$
$$c_1 = [\![\ show\ ]\!]$$

Thinking carefully about the levels involved, the signature indicates that the evidence for *Show a*, which is available at stage 0, can be used to satisfy the evidence needed by *Show a* at stage 1.

[3]There is, however, an artificial restriction about nesting quotation brackets which makes writing programs with more than two stages difficult. It is ongoing work to lift this restriction.

In the normal dictionary passing implementation of type classes, type class constraints are elaborated to a function which accepts a dictionary which acts as evidence for the constraint. Therefore we can assume that the elaborated version of c_1 looks similar to the following:

$$d_1 :: ShowDict\ a \longrightarrow Code\ (a \longrightarrow String)$$
$$d_1\ dShow = [\![\ show\ dShow\ ]\!]$$

Now this reveals a subtle problem: naively elaborating without considering the *levels of constraints* has introduced a cross-stage reference where the dictionary variable *dShow* is introduced at level 0 but used at level 1. As we have learned in Section 2, one remedy of this situation is to try to make *dShow* cross-stage persistent by lifting. However, in general, lifting of dictionaries is not straightforward to implement. Recall that lifting in GHC is restricted to instances of the *Lift* class which excludes functions – but type class dictionaries are nearly always a record of functions, so automatic lifting given the current implementation is not possible.

GHC nevertheless accepts this program, with the underlying problem only being revealed when subsequently trying to splice the program. We instead argue that the program c_1 is ill-typed and should therefore be rejected at compilation time. Our solution is to introduce a new constraint form, *CodeC C*, which indicates that constraint *C* is available to be used in the next stage. Using this, the corrected type signature for example C1 is as follows:

$$c_1' :: CodeC\ (Show\ a) \Rightarrow Code\ (a \longrightarrow String)$$
$$c_1' = [\![\ show\ ]\!]$$

The *CodeC* (*Show a*) constraint is introduced at level 0 but indicates that the *Show a* constraint will be available to be used at level 1. Therefore the *Show a* constraint can be used to satisfy the *show* method used inside the quotation. The corresponding elaborated version is similar to the following:

$$d_1' :: Code\ (ShowDict\ a) \longrightarrow Code\ (a \longrightarrow String)$$
$$d_1'\ cdShow = [\![\ show\ \$(cdShow)\ ]\!]$$

As *cdShow* is now the representation of a dictionary, we can splice the representation inside the quote. The reference to *cdShow* is at the correct level and the program is well-staged.

Example C2. The example C1 uses a locally provided constraint which causes us some difficulty. We have a different situation if a constraint can be solved by a concrete global type class instance:

$$c_2 :: Code\ (Int \longrightarrow String)$$
$$c_2 = [\![\ show\ ]\!]$$

In c_2, the global *Show Int* instance is used to satisfy the *show* constraint inside the quotation. Since this elaborates to a reference to a top-level instance dictionary, the reference can be persisted using path-based persistence. Therefore it is permitted to use top-level instances to satisfy constraints inside a quotation, in the same way that it is permitted to refer to top-level variables.

GHC currently correctly accepts this program. However, it does not actually perform the dictionary translation until the program is spliced, which is harmless in this case, because the same *Show Int* instance is in scope at both points.

3.2 Top-level splices

A splice that appears in a definition at the top-level scope of a module[4] introduces new scoping challenges because it can potentially require class constraints to be used at levels prior to the ones where they are introduced.

[4]Do not confuse this use of "top-level" with the staging level.

Since a top-level splice is evaluated at compile time, it should be clear that no run-time information must be used in a top-level splice definition. In particular, the definition should not be permitted to access local variables or local constraints defined at a higher level. This is because local information which is available only at runtime cannot be used to influence the execution of the expression during compilation.

Example TS1. In the following example there are two modules. Module A defines the *Lift* class and contains the definition of a global instance *Lift Int*. The *lift* function from this instance is used in module B in the definition of ts_1, which is a top-level definition. The reference to *lift* occurs inside a top-level splice, thus at level -1, and so the *Lift Int* instance is needed a compile time.

```
module A where
class Lift a where
  lift :: a → Code a
instance Lift Int where
  ...
```

```
module B where
import A
ts1 :: Int
ts1 = $(lift 5)
```

This program is currently accepted by GHC, and it should be, because the evidence for a top-level instance is defined in a top-level variable and top-level definitions defined in other modules are permitted to appear in top-level splices.

The situation is subtly different to top-level quotations as in example C2, because the instance must be defined in another module which mirrors the restriction implemented in GHC that top-level definitions can only be used in top-level splices if they are defined in other modules.

Example TS2. On the other hand, constraints introduced locally by a type signature for a top-level definition must not be allowed. In the following example, the *Lift A* constraint is introduced at level 0 by the type signature of ts_2 but used at level -1 in the body:

```
data A = A

ts2 :: Lift A ⇒ A
ts2 = $(lift A)
```

We assume *lift* is imported from another module as in Example TS1 and therefore cross-stage persistent. However, the problem is that the dictionary that will be used to implement the *Lift A* constraint will not be known until runtime. Therefore the definition of *Lift A* is not available to be used at compile-time in the top-level splice, and GHC correctly rejects this program. It is evident why this program should be rejected considering the dictionary translation, in a similar vein to C1. The elaboration would amount to a future-stage reference inside the splice.

Example TS3. So far we have considered the interaction between constraints and quotations separately to the interaction between constraints and top-level splices. The combination of the two reveals further subtle issues. The following function ts_3 is at the top-level and uses a constrained type. This example is both well-typed and well-staged.

```
ts3 :: Ord a ⇒ a → a → Ordering
ts3 = $(⟦ compare ⟧)
```

In ts_3, the body of the top-level splice is a simple quotation of the *compare* method. This method requires an *Ord* constraint which is provided by the context on ts_3. The constraint is introduced at level 0 and also used at level 0, as the top-level splice and the quotation cancel each other out. It is therefore perfectly fine to use the dictionary passed to ts_3 to satisfy the requirements of *compare*.

Unfortunately, the current implementation in GHC rejects this program. It generally excludes local constraints from the scope inside top-level splices, in order to reject programs like Example TS2. Our specification accepts the example by tracking the levels of local constraints.

3.3 The *power* function revisited

Having discussed constraints and also their interaction with top-level splices, we can now fully explain why *power5* function discussed in Section 1 works in current GHC when spliced at a concrete type, but fails when spliced with an overloaded type, whereas it should work for both:

$$
\begin{aligned}
&power5 :: Int \to Int && \text{-- Option M}\\
&power5 :: Num\ a \Rightarrow a \to a && \text{-- Option P}\\
&power5\ n = \$(power\ 5\ [\![\ n\ ]\!])
\end{aligned}
$$

The two problems are that GHC accepts *power* at the incorrect type

$$power :: Num\ a \Rightarrow Int \to Code\ a \to Code\ a$$

(as in Example C1) and that it does not actually perform a dictionary translation for this until this is spliced. Option M works in GHC, because it finds the *Num Int* global instance when splicing, similarly to Example TS1, and everything is (accidentally) fine. However, Option P fails because the local constraint is not made available in the splice as in Example TS2, and even if it was, it would be a reference at the wrong level.

As with Example C1, we argue that the function *power* should instead only be accepted at type

$$power :: CodeC\ (Num\ a) \Rightarrow Int \to Code\ a \to Code\ a$$

Then, Option M is fine due to the cross-stage persistence of the global *Num Int* instance declaration; and Option P works as well, as the program will elaborate to code that is similar to:

$$
\begin{aligned}
&power5' :: NumDict\ a \to a \to a\\
&power5'\ dNum\ n = \$(power\ [\![\ dNum\ ]\!]\ 5\ [\![\ n\ ]\!])
\end{aligned}
$$

By quoting *dNum*, the argument to *power* is a representation of a dictionary as required, and reference is at the correct level.

3.4 Instance Definitions

The final challenge to do with constraints is dealing with instance definitions which use top-level splices. This situation is of particular interest as there are already special typing rules in GHC for instance methods which bring into scope the instance currently being defined in the body of the instance definition. This commonly happens in recursive datatypes where the instance declaration must be recursively defined.

Example I1. In the same manner as a top-level splice, the body of an instance method cannot use a local constraint, in particular the instance currently being defined or any of the instance head in order to influence the code generation in a top-level splice. On the other hand, the generated code should certainly be permitted to use the instances from the instance context and the currently defined instance. In fact, this is necessary in order to generate the majority of instances for recursive datatypes! Here is a instance definition that is defined recursively.

$$
\begin{aligned}
&\textbf{data}\ Stream = Cons\ \{hd :: Int, tl :: Stream\}\\
&\textbf{instance}\ Eq\ Stream\ \textbf{where}\\
&\quad s_1 == s_2 = hd\ s_1 == hd\ s_2\ \&\&\ tl\ s_1 == tl\ s_2
\end{aligned}
$$

The instance for *Eq Stream* compares the tails of the streams using the equality instance for *Eq Stream* that is currently being defined. This code works well, as it should, so there is reason to believe that a staged version should also be definable.

Example I2. The following is just a small variation of Example I1 where the body of the method is wrapped in a splice and a quote:

$$\textbf{instance}\ \mathit{Eq\ Stream}\ \textbf{where}$$
$$(==) = \$(\llbracket\ \lambda s_1\ s_2 \rightarrow \mathit{hd}\ s_1 == \mathit{hd}\ s_2\ \&\&\ \mathit{tl}\ s_1 == \mathit{tl}\ s_2\ \rrbracket)$$

This program should be accepted and equivalent to I1, because in general, $\$(\llbracket\ e\ \rrbracket) = e$. However, it is presently rejected by GHC, with the claim that the recursive use of (==) for comparing the tails is not stage-correct, when in principle it could and should be made cross-stage persistent.

Example I3. Yet another variant of Example I2 is to try to move the recursion out of the instance declaration, as follows:

$$\mathit{eqStream} :: \mathit{CodeC}\ (\mathit{Eq\ Stream}) \Rightarrow \mathit{Code}\ (\mathit{Stream} \rightarrow \mathit{Stream} \rightarrow \mathit{Bool})$$
$$\mathit{eqStream} = \llbracket\ \lambda s_1\ s_2 \rightarrow \mathit{hd}\ s_1 == \mathit{hd}\ s_2\ \&\&\ \mathit{tl}\ s_1 == \mathit{tl}\ s_2\ \rrbracket$$

Then, in a different module, we should be able to say:

$$\textbf{instance}\ \mathit{Eq\ Stream}\ \textbf{where}$$
$$(==) = \$(\mathit{eqStream})$$

It is important that the definition of *eqStream* uses the new constraint form *CodeC* (*Eq Stream*) so that the definition of *eqStream* is well-staged. As *Eq Stream* is the instance which is currently being defined, the *Eq Stream* constraint is introduced at level 0 when typing the instance definition. Therefore, in a top-level splice the local constraint can only be used by functions which require a *CodeC* (*Eq Stream*) constraint. This example is similar to *power* in Section 3.3 but the local constraint is introduced implicitly by the instance definition.

It is somewhat ironic that while one of the major use cases of untyped Template Haskell is generating instance definitions such as this one, it seems impossible to use the current implementation of Typed Template Haskell for the same purpose.

3.5 Type Variables

The previous examples showed that type constraints require special attention in order to ensure correct staging. It is therefore natural to consider type variables as well, and indeed previous work by Pickering et al. [2019a] ensured that type variables are were level-aware in order to ensure a sound translation. However, our calculus will show that this is a conservative position in a language based on the polymorphic lambda calculus which enforces a phase distinction [Cardelli 1988] where typechecking happens entirely prior to running any stages of a program.

Example TV1. Consider the following example that turns a list of quoted values into a quoted list of those values:

$$\mathit{list} :: \forall a.[\mathit{Code}\ a] \rightarrow \mathit{Code}\ [a]$$
$$\mathit{list}\ [\,] \quad\ = \llbracket\ [\,]\ @a\ \rrbracket$$
$$\mathit{list}\ (x : xs) = \llbracket\ (:)\ @a\ \$(x)\ \$(\mathit{list}\ @a\ xs)\ \rrbracket$$

The type variable a is bound at level 0 and used both at at level 1 (in its application to the type constructors), and at level 0 (in its recursive application to the *list* function). If we took the lead from values then this program would be rejected by the typechecker. However, because of the in-built phase distinction, evaluation cannot get stuck on an unknown type variable as all the types

will be fixed in the program before execution starts. We just need to make sure that the substitution operation can substitute a type inside a quotation.

As it happens, this definition is correctly accepted by GHC, which is promising. However, it cannot actually be used since GHC produces compile error when given an expression such as $(*list* [⟦ *True* ⟧, ⟦ *False* ⟧]), since it complains that the type variable *a* is not in scope during type checking, and eventually generates an internal error in GHC itself.

3.6 Intermediate Summary

This section has uncovered the fact that the treatment of constraints is rather arbitrary in the current implementation of GHC. The examples we discussed have been motivated by three different ad-hoc restrictions that the current implementation exhibits. Firstly, prior stage constraint references (Section 3.1) are permitted without any checks. Secondly, top-level splices (Section 3.2) are typechecked in an environment isolated from the local constraint environment which is an overly conservative restriction. Thirdly, in an instance definition (Section 3.4), any reference to the instance currently being defined inside a top-level splice is rejected even if it is level-correct. Additionally we showed that although constraints have a role to play in understanding staging, type variables need no special treatment beyond the correct use of type application (Section 3.5).

Do the problems uncovered threaten the soundness of the language? In the case of prior-stage references, they do. The current representation form of quotations in GHC is untyped and therefore any evidence is erased from the internal representation of a quotation. The instance is therefore "persisted by inference", which operates under the assumption that enough information is present in an untyped representation to re-infer all the contextual type information erased after typechecking. This assumption leads to unsoundness as generated programs will fail to typecheck, as discussed already by Pickering et al. [2019a]. Future-stage references are forbidden by conservative checks, which we will aim to make more precise in our calculus in order to accept more programs.

The problems observed so far are the result of interactions between class constraints and staging. Our goal now will be to develop a formal calculus to model and resolve these problems. The approach will be to introduce a source language that includes type constraints in addition to quotes and splices (Section 4), and a core language that is a variant of the explicit polymorphic lambda calculus with multi-stage constructs (Section 5). We then show how the source language can be translated into a core language where constraints have been elaborated away into a representation form for a quotation that is typed and where all contextual type information is saved (Section 6).

4 SOURCE LANGUAGE

The source language we introduce has been designed to incorporate the essential features of a language with metaprogramming and qualified types that is able to correctly handle the situations discussed in Section 3. We try to stay faithful to GHC's current implementation of Typed Template Haskell where possible, with the addition of the quoted constraint form *CodeC* for the reasons discussed in Section 3. The key features of this language are:

(1) Values *and constraints* are always indexed by the level at which they are introduced to ensure that they are well-staged.
(2) The constraint form *CodeC* indicates that a constraint is available at the next stage.
(3) Types (including type variables) are not level-indexed and can be used at any level.
(4) There is no **run** operation in the language. All evaluation of code is performed at compile-time by mean of top-level splices which imply the existence of negative levels.
(5) Path-based cross-stage persistence is supported.

$$
\begin{array}{lll}
\mathit{pgm} & ::= & e \mid \mathit{def}\,;\mathit{pgm} \mid \mathit{cls}\,;\mathit{pgm} \mid \mathit{inst}\,;\mathit{pgm} \\
\mathit{def} & ::= & \textbf{def}\ k = e \\
\mathit{cls} & ::= & \textbf{class}\ \mathit{TC}\ a\ \textbf{where}\ \{k :: \sigma\} \\
\mathit{inst} & ::= & \textbf{instance}\ \overline{C} \Rightarrow \mathit{TC}\ \tau\ \textbf{where}\ \{k = e\} \\
e & ::= & \qquad\qquad\qquad\qquad\qquad\qquad \text{expressions} \\
 & & x \mid k \qquad\qquad\qquad\qquad\qquad\qquad \text{variables / globals} \\
 & \mid & \lambda x{:}\tau.e \mid e\ e \qquad\qquad\qquad\qquad \text{abstraction / application} \\
 & \mid & [\![\ e\ ]\!] \mid \$(e) \qquad\qquad\qquad\qquad\qquad \text{quotation / splice} \\
\tau & ::= & \qquad\qquad\qquad\qquad\qquad\qquad \text{types} \\
 & & a \mid H \qquad\qquad\qquad\qquad\qquad\qquad \text{variables / constants} \\
 & \mid & \tau \rightarrow \tau \qquad\qquad\qquad\qquad\qquad\qquad \text{functions} \\
 & \mid & \mathit{Code}\ \tau \qquad\qquad\qquad\qquad\qquad\qquad \text{representation} \\
\rho & ::= & \tau \mid C \Rightarrow \rho \qquad\qquad\qquad\qquad\qquad\qquad \text{qualified types} \\
\sigma & ::= & \rho \mid \forall a.\sigma \qquad\qquad\qquad\qquad\qquad\qquad \text{quantification} \\
C & ::= & \mathit{TC}\ \tau \mid \mathit{CodeC}\ C \qquad\qquad\qquad\qquad \text{constraint / representation} \\
\Gamma & ::= & \bullet \mid \Gamma, x{:}(\tau, n) \mid \Gamma, a \mid \Gamma, (C, n) \qquad\qquad \text{type environment} \\
P & ::= & \bullet \mid P, (\forall a.\overline{C} \Rightarrow C) \mid P, k{:}\sigma \qquad\qquad \text{program environment}
\end{array}
$$

Fig. 1. Source Syntax

These features have been carefully chosen to allow a specification of Typed Template Haskell that addresses all the shortcomings identified in Section 3 while staying close to the general flavour of meta-programming that is enabled in its current implementation in GHC.

4.1 Syntax

The syntax of the source language (Figure 1) models programs with type classes and metaprogramming features. This syntax closely follows the models of languages with type classes of Bottu et al. [2017] and Chakravarty et al. [2005], but with the addition of multi-stage features and without equality or quantified constraints.

A program *pgm* is a sequence of value definitions *def*, class definitions *cls*, and instance definitions *inst* followed by a top-level expression *e*.

Top-level definitions *def* are added to the calculus to model separate compilation and path-based cross-stage persistence in a way similar to what GHC currently implements: only variables previously defined in a top-level definition can be referenced at arbitrary levels.

Both type classes and instances are in the language, but simplified as far as possible: type class definitions *cls* have precisely one method and no superclasses; instance definitions *inst* are permitted to have an instance context.

The expression language e is a standard λ-calculus with the addition of the two multi-stage constructs, quotation $[\![\ e\ ]\!]$ and splicing $\$(e)$ which can be used to separate a program into stages.

Just as in Haskell, our language is *predicative*: type variables that appear as class parameters and in quantifiers range only over monotypes. This is modelled by the use of a Damas-Milner style type system which distinguishes between monotypes τ, qualified types ρ and polytypes σ. The type argument to the representation constructor $\mathit{Code}\ \tau$ must be a monotype.

A constraint C is either a type class constraint $\mathit{TC}\ \tau$, or a representation of a constraint $\mathit{CodeC}\ C$.

The environment Γ is used for locally introduced information, including value variables $x : (\tau, n)$, type variables a, and local type class axioms (C, n). The environment keeps track of the (integer)

level n that value and constraint variables are introduced at; the typing rules ensure that the variables are only used at the current level.

The program theory P is an environment of the type class axioms introduced by instance declarations and of type information for names introduced by top-level definitions. The axioms are used to dictate whether the usage of a type class method is allowed or not. The names indicate which variables can be used in a cross-stage persistent fashion.

4.2 Typing Rules

The typing rules proceed in a predicable way for anyone familiar with qualified type systems. The difference is that the source expression typing judgement (Figure 2) and constraint entailment judgement (Figure 3) are now indexed by a level. Furthermore, since these rules are the basis for elaboration into the core language, they have been combined with the elaboration rules, as highlighted by a grey box whose contents can be safely ignored until the discussion of elaboration (Section 6).

The typing rules also refer to type and constraint formation rules (Figures 8 and 9), which check just that type variables are well-scoped (our language does not include an advanced kind system).

4.2.1 Levels. A large part of the rules are indexed by their (integer) level. It is standard to index the expression judgement at a specific level but less standard to index the constraint entailment judgement. The purpose of the level index is to ensure that expression variables and constraints can only be used at the level they are introduced. The result is a program which is separated into stages in such a way that when imbued with operational semantics, the fragments at stage n will be evaluated before the fragments at stage $n + 1$.

The full program is checked at level 0. Quotation (E_Quote) increases the level by one and splicing (E_Splice) decreases the level. It is permitted to reach negative levels in this source language. The negative levels indicate a stage which should evaluate at compile-time.

In more detail, the consequence of the level-indexing is that:

- A variable can be introduced at any specific level n by a λ-abstraction (E_Abs) and used at precisely that level (E_Var).
- Variables introduced by top-level definitions can be used at any level (E_Var_TopLevel).
- Type variables can be introduced (E_TAbs) and used at any level (E_TApp).
- Constraints can be introduced at any level (E_C_Abs), but in practice, due to the system being predicative, this level will always be 0 (see Section 4.2.2). Constraints are also introduced into the program logic by instance declarations. Constraints are appropriately eliminated by application (E_C_App). The entailment relation (Figure 3) ensures that a constraint is available at the current stage.

Note that we do not allow implicit lifting as discussed in Section 2 and implemented in GHC. Explicit lifting via a *Lift* class can easily be expressed in this language, and implicit lifting, if desired, can be orthogonally added as an additional elaboration step.

The rules furthermore implement the top-level splice restriction present in GHC. A top-level splice will require its body to be at level -1 and thereby rule out the use of any variables defined outside of the splice, with the exception of top-level variables.

4.2.2 Level of constraints. Inspecting the E_C_Abs rule independently it would appear that a local constraint can be introduced at any level and then be used at that level. In fact, the E_C_Abs rule can only ever be applied at level 0 because the E_Quote rule requires that the enclosed expression has a τ type. This forbids the ρ type as produced by E_C_Abs and more closely models the restriction to predicative types as present in GHC.

$$\boxed{P;\Gamma \vdash^n e : \sigma \rightsquigarrow t \mid TSP}$$

$$\frac{x{:}(\tau, n) \in \Gamma}{P;\Gamma \vdash^n x : \tau \rightsquigarrow x \mid \bullet}\ \text{E\_Var} \qquad \frac{k{:}\sigma \in P}{P;\Gamma \vdash^n k : \sigma \rightsquigarrow k \mid \bullet}\ \text{E\_Var\_TopLevel}$$

$$\frac{P;\Gamma, x{:}(\tau_1, n) \vdash^n e : \tau_2 \rightsquigarrow t \mid TSP \qquad \Gamma \vdash_{\text{ty}} \tau_1 \rightsquigarrow \tau_1'}{P;\Gamma \vdash^n \lambda x{:}\tau_1.e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x{:}\tau_1'.t \mid TSP}\ \text{E\_Abs}$$

$$\frac{P;\Gamma \vdash^n e_1 : \tau_1 \to \tau_2 \rightsquigarrow t_1 \mid TSP_1 \qquad P;\Gamma \vdash^n e_2 : \tau_1 \rightsquigarrow t_2 \mid TSP_2}{P;\Gamma \vdash^n e_1\ e_2 : \tau_2 \rightsquigarrow t_1\ t_2 \mid TSP_1 \cup TSP_2}\ \text{E\_App}$$

$$\frac{a \notin \Gamma \qquad P;\Gamma, a \vdash^n e : \sigma \rightsquigarrow t \mid TSP}{P;\Gamma \vdash^n e : \forall a.\sigma \rightsquigarrow \Lambda a.t \mid TSP}\ \text{E\_TAbs} \qquad \frac{P;\Gamma \vdash^n e : \forall a.\sigma \rightsquigarrow t \mid TSP \qquad \Gamma \vdash_{\text{ty}} \tau \rightsquigarrow \tau'}{P;\Gamma \vdash^n e : \sigma[\tau/a] \rightsquigarrow e\ \langle \tau' \rangle \mid TSP}\ \text{E\_TApp}$$

$$\frac{P;\Gamma \vdash^{n+1} e : \tau \rightsquigarrow t \mid TSP}{P;\Gamma \vdash^n [\![\ e\ ]\!] : Code\ \tau \rightsquigarrow [\![\ t\ ]\!]_{TSP_n} \mid \lfloor TSP \rfloor^n}\ \text{E\_Quote}$$

$$\frac{P;\Gamma \vdash^{n-1} e : Code\ \tau \rightsquigarrow t \mid TSP \qquad \text{fresh } sp \qquad \Gamma \vdash_{\text{ty}} \tau \rightsquigarrow \tau' \qquad \Gamma \rightsquigarrow \Delta}{P;\Gamma \vdash^n \$(e) : \tau \rightsquigarrow sp \mid \{\Delta \vdash sp{:}\tau' = t\} \cup_{n-1} TSP}\ \text{E\_Splice}$$

$$\frac{\Gamma \vdash_{\text{ct}} C \rightsquigarrow \tau \qquad P;\Gamma, ev{:}(C, n) \vdash^n e : \rho \rightsquigarrow t \mid TSP \qquad \text{fresh } ev}{P;\Gamma \vdash^n e : C \Rightarrow \rho \rightsquigarrow \lambda ev{:}\tau.t \mid TSP}\ \text{E\_C\_Abs}$$

$$\frac{P;\Gamma \vdash^n e : C \Rightarrow \rho \rightsquigarrow t_1 \mid TSP_1 \qquad P;\Gamma \vDash^n C \rightsquigarrow t_2 \mid TSP_2}{P;\Gamma \vdash^n e : \rho \rightsquigarrow t_1\ t_2 \mid TSP_1 \cup TSP_2}\ \text{E\_C\_App}$$

Fig. 2. Source Expression Typing with Elaboration Semantics

The new constraint form *CodeC C* can be used to allow local constraints to be used at positive levels in E_C_App. The rules C_Incr and C_Decr in constraint entailment can convert from *CodeC C* and *C* and vice versa. During elaboration, these rules will insert splices and quotes around the dictionary arguments as needed.

Constraints satisfied by instance declarations can be used at any level by means of C_Global.

Let us illustrate this by revisiting Example C1 and considering the expression $[\![\ show\ ]\!]$. In this case, we assume the program environment to contain the type of *show*:

$$P = \bullet, show{:}\forall a.Show\ a \Rightarrow a \to String$$

From this we can use E_Var_TopLevel to conclude that

$$P;\Gamma \vdash^1 show : \forall a.Show\ a \Rightarrow a \to String$$

Because $[\![\ show\ ]\!]$ must have type *Code* τ for some monotype τ, we cannot apply E_Quote yet, but must first apply E_TApp and E_C_App. We can conclude

$$\boxed{P; \Gamma \vDash^n C \rightsquigarrow t \mid TSP}$$

$$\frac{ev : (\forall a.\overline{C_i} \Rightarrow C) \in P \qquad \Gamma \vdash_{\mathsf{ty}} \tau \rightsquigarrow \tau' \qquad \overline{P; \Gamma \vDash^n C_i[\tau/a] \rightsquigarrow t_i \mid TSP_i}}{P; \Gamma \vDash^n C[\tau/a] \rightsquigarrow ev\ \langle \tau' \rangle\ \overline{t_i} \mid \bigcup_i TSP_i}\ \text{C\_Global}$$

$$\frac{ev : (C, n) \in \Gamma}{P; \Gamma \vDash^n C \rightsquigarrow ev \mid \bullet}\ \text{C\_Local} \qquad \frac{P; \Gamma \vDash^{n+1} C \rightsquigarrow t \mid TSP}{P; \Gamma \vDash^n CodeC\ C \rightsquigarrow [\![t]\!]_{TSP_n} \mid \lfloor TSP \rfloor^n}\ \text{C\_Decr}$$

$$\frac{P; \Gamma \vDash^{n-1} CodeC\ C \rightsquigarrow t \mid TSP \qquad \Gamma \vdash_{\mathsf{ct}} C \rightsquigarrow \tau \qquad \text{fresh } sp \qquad \Gamma \rightsquigarrow \Delta}{P; \Gamma \vDash^n C \rightsquigarrow sp \mid \{\Delta \vdash sp : \tau = t\} \cup_{n-1} TSP}\ \text{C\_Incr}$$

Fig. 3. Source Constraints with Elaboration

$$P; \Gamma \vdash^1 show : a \to String$$

if the entailment

$$P; \Gamma \vDash^1 Show\ a$$

holds. As we do not assume any instances for *Show* in this example, the constraint can only be justified via the local environment Γ. Because, as discussed above, any constraint can only have been introduced at level 0, the only way to move a local constraint from level 0 to level 1 is rule C_Incr, which requires us to show

$$P; \Gamma \vDash^0 CodeC\ (Show\ a)$$

This in turn follows from C_Local if we assume that Γ contains $CodeC\ (Show\ a)$:

$$\Gamma = \bullet, (CodeC\ (Show\ a), 0)$$

At this point, we know that

$$P; \Gamma \vdash^0 [\![show]\!] : Code\ (a \to String)$$

and by applying E_C_Abs and E_TAbs, we obtain

$$P; \bullet \vdash^0 [\![show]\!] : \forall a.CodeC\ (Show\ a) \Rightarrow Code\ (a \to String)$$

as desired.

4.2.3 Program Typing. A program (Figure 7) is a sequence of value, class and instances declarations followed by an expression. The declaration forms extend the program theory which is used to typecheck subsequent definitions. Value definitions extend the list of top-level definitions available at all stages. The judgement makes it clear that the top-level of the program is level 0 and that each expression is checked in an empty local environment.

Class definitions extend the program theory with the qualified class method. Instance definitions extend the environment with an axiom for the specific instance which is being defined. The rule checks that the class method is of the type specified in the class definition.

Notice that before the type class method is checked, the local environment Γ is extended with an axiom for the instance we are currently checking. This is important to allow recursive definitions (Example I1) to be defined in a natural fashion. The constraint is introduced at level 0,

$$\boxed{pgm \mid P \vdash_{\mathsf{def}} def : P \rightsquigarrow pgm}$$

$$\frac{P; \bullet \vdash^{0} e : \sigma \rightsquigarrow t \mid TSP \qquad \bullet \vdash_{\mathsf{ty}} \sigma \rightsquigarrow \tau \qquad d = \mathbf{def}\ k : \tau = t \qquad ds = collapse(-1, TSP, d; pgm)}{pgm \mid P \vdash_{\mathsf{def}} \mathbf{def}\ k = e : P, k : \sigma \rightsquigarrow ds}\ \textsc{Def}$$

Fig. 4. Source Definition Typing with Elaboration

$$\boxed{pgm \mid P \vdash_{\mathsf{cls}} cls : P \rightsquigarrow pgm}$$

$$\frac{\bullet, a \vdash_{\mathsf{ty}} \sigma}{pgm \mid P \vdash_{\mathsf{cls}} \mathbf{class}\ TC\ a\ \mathbf{where}\ \{k :: \sigma\} : P, k : \forall a.TC\ a \Rightarrow \sigma \rightsquigarrow pgm}\ \textsc{Cls}$$

Fig. 5. Source Class Typing with Elaboration

$$\boxed{pgm \mid P \vdash_{\mathsf{inst}} inst : P \rightsquigarrow pgm}$$

$$\frac{\begin{array}{c}\mathbf{class}\ TC\ a\ \mathbf{where}\ \{k :: \sigma\} \qquad \overline{b_j} = \mathsf{fv}(\tau) \qquad \bullet \vdash_{\mathsf{ty}} \sigma[\tau/a] \rightsquigarrow \tau' \qquad \overline{\bullet, \overline{b_j} \vdash_{\mathsf{ct}} C_i \rightsquigarrow \tau_i} \\ P; \bullet, \overline{b_j}, ev : (TC\ \tau, 0), \overline{(C_i, 0)} \vdash^{0} e : \sigma[\tau/a] \rightsquigarrow t \mid TSP \\ dI = \mathbf{def}\ ev : (\forall \overline{b_j}.\overline{\tau_i} \rightarrow \tau') = t \qquad ds = collapse(-1, TSP, dI; pgm) \qquad \text{fresh } ev\end{array}}{pgm \mid P \vdash_{\mathsf{inst}} \mathbf{instance}\ \overline{C_i} \Rightarrow TC\ \tau\ \mathbf{where}\ \{k = e\} : P, ev : (\forall \overline{b_j}.\overline{C_i} \Rightarrow TC\ \tau) \rightsquigarrow ds}\ \textsc{Inst}$$

Fig. 6. Source Instance Typing with Elaboration

the top-level of the program. It is important to introduce the constraint to the local environment rather than program theory because this constraint should not be available at negative levels. If the constraint was introduced to the program theory, then it could be used at negative levels, which would amount to attempting to use the instance in order to affect the definition of said instance. This nuance in the typing rules explains why Example I2 is accepted and the necessity of the *CodeC* constraint in Example I3.

5 THE CORE LANGUAGE

In this section we describe an explicitly typed core language which is suitable as a compilation target for the declarative source language we described in the Section 4.

5.1 Syntax

The syntax for the core language is presented in Figure 10. It is a variant of the explicit polymorphic lambda calculus with multi-stage constructs, top-level definitions and top-level splice definitions.

Quotes and splices are represented by the syntactic form $[\![\ e\ ]\!]_{SP}$, which is a quotation with an associated splice environment which binds *splice variables* for each splice point within the quoted expression. A splice point is where the result of evaluating a splice will be inserted. Splice variables to represent the splice points are bound by splice environments and top-level splice definitions. The expression syntax contains no splices, splices are modelled using the splice environments which are attached to quotations and top-level splice definitions.

$$\boxed{P \vdash_{\text{pgm}} pgm : \sigma \rightsquigarrow pgm}$$

$$\frac{P; \bullet \vdash^{0} e : \sigma \rightsquigarrow t \mid TSP \qquad \bullet \vdash_{\text{ty}} \sigma \rightsquigarrow \tau \qquad p = t : \tau \qquad ds = collapse(-1, TSP, p)}{P \vdash_{\text{pgm}} e : \sigma \rightsquigarrow ds}\ \text{P\_Expr}$$

$$\frac{P_2 \vdash_{\text{pgm}} pgm : \sigma \rightsquigarrow pgm \qquad pgm \mid P_1 \vdash_{\text{def}} def : P_2 \rightsquigarrow pgm_1}{P_1 \vdash_{\text{pgm}} def; pgm : \sigma \rightsquigarrow pgm_1}\ \text{P\_Def}$$

$$\frac{P_2 \vdash_{\text{pgm}} pgm : \sigma \rightsquigarrow pgm \qquad pgm \mid P_1 \vdash_{\text{cls}} cls : P_2 \rightsquigarrow pgm_2}{P_1 \vdash_{\text{pgm}} cls; pgm : \sigma \rightsquigarrow pgm_1}\ \text{P\_Cls}$$

$$\frac{P_2 \vdash_{\text{pgm}} pgm : \sigma \rightsquigarrow pgm \qquad pgm \mid P_1 \vdash_{\text{inst}} inst : P_2 \rightsquigarrow pgm_1}{P_1 \vdash_{\text{pgm}} inst; pgm : \sigma \rightsquigarrow pgm_1}\ \text{P\_Inst}$$

Fig. 7. Source Program Typing with Elaboration

$$\boxed{\Gamma \vdash_{\text{ty}} \sigma \rightsquigarrow \tau}$$

$$\frac{a \in \Gamma}{\Gamma \vdash_{\text{ty}} a \rightsquigarrow a}\ \text{T\_Var} \qquad \frac{}{\Gamma \vdash_{\text{ty}} H \rightsquigarrow H}\ \text{T\_Const} \qquad \frac{\Gamma \vdash_{\text{ty}} \tau_1 \rightsquigarrow \tau_1' \qquad \Gamma \vdash_{\text{ty}} \tau_2 \rightsquigarrow \tau_2'}{\Gamma \vdash_{\text{ty}} \tau_1 \to \tau_2 \rightsquigarrow \tau_1' \to \tau_2'}\ \text{T\_Arrow}$$

$$\frac{\Gamma \vdash_{\text{ct}} C \rightsquigarrow \tau_1 \qquad \Gamma \vdash_{\text{ty}} \rho \rightsquigarrow \tau_2}{\Gamma \vdash_{\text{ty}} C \Rightarrow \rho \rightsquigarrow \tau_1 \to \tau_2}\ \text{T\_CArrow} \qquad \frac{\Gamma, a \vdash_{\text{ty}} \sigma \rightsquigarrow \tau}{\Gamma \vdash_{\text{ty}} \forall a.\sigma \rightsquigarrow \forall a.\tau}\ \text{T\_For\_All}$$

$$\frac{\Gamma \vdash_{\text{ty}} \tau \rightsquigarrow \tau'}{\Gamma \vdash_{\text{ty}} Code\ \tau \rightsquigarrow Code\ \tau'}\ \text{T\_Code}$$

Fig. 8. Source Type Formation with Elaboration

$$\boxed{\Gamma \vdash_{\text{ct}} C \rightsquigarrow \tau}$$

$$\frac{\textbf{class}\ TC\ a\ \textbf{where}\ \{k :: \sigma\} \qquad \Gamma \vdash_{\text{ty}} \sigma[\tau/a] \rightsquigarrow \tau'}{\Gamma \vdash_{\text{ct}} TC\ \tau \rightsquigarrow \tau'}\ \text{C\_TC} \qquad \frac{\Gamma \vdash_{\text{ct}} C \rightsquigarrow \tau}{\Gamma \vdash_{\text{ct}} CodeC\ C \rightsquigarrow Code\ \tau}\ \text{C\_CodeC}$$

Fig. 9. Source Constraint Formation with Elaboration

The splice environment maps a splice point sp to a local type environment Δ, a type τ and an expression e which we write as $\Delta \vdash sp : \tau = e$. The typing rules will ensure that the expression e has type $Code\ \tau$. The purpose of the environment Δ is to support open code representations which lose their lexical scoping when lifted from the quotation.

A top-level splice definition is used to support elaborating from negative levels in the source language. These top level declarations are of the form **spdef** $\Delta \vdash^{n} sp : \tau = e$ which indicates that

the expression e will have type $Code\ \tau$ at level n in environment Δ. Top-level splice definitions also explicitly record the level of the original splice so that the declaration can be typechecked at the correct level. The level index is not necessary for the quotation splice environments because the level of the whole environment is fixed by the level at which the quotation appears.

5.2 Typing Rules

The expression typing rules for the core language are for the most part the same as those in the source language. The rules that differ are shown in Figure 11.

The splice environment typing rules are given in Figure 12. A splice environment is well-typed if each of its definitions is well-typed. We check that the body of each definition has type $Code\ \tau$ in an environment extended by Δ. In the E_Quote rule, the body of the quotation is checked in an environment Γ extended with the contents of the splice environment. A splice variable sp is associated with an environment Δ, a type τ and a level n in Γ. When a splice variable is used, the E_Splice_Var rule ensures that the claimed local environment aligns with the actual environment, the type of the variable is correct and the level matches the surrounding context.

Top-level splice definitions are typed in a similar manner in Figure 13. The body of the definition is checked to have type $Code\ \tau$ at level n in environment Γ, the program theory is then extended with a splice variable $sp : (\Gamma, \tau)$ which can be used in the remainder of the program.

5.3 Dynamic Semantics

The introduction of splice environments makes the evaluation order of the core calculus evident and hence a suitable target for compilation.

Usually in order to ensure a well-staged evaluation order, the reduction relation must be level-indexed to evaluate splices inside quotations. In our calculus, there is no need to do this because the splices have already been lifted outside of the quotation during the elaboration process. This style is less convenient to program with, but easy to reason about and implement.

In realistic implementations, the quotations are compiled to a representation form for which implementing substitution can be difficult. By lifting the splices outside of the representation, the representation does not need to be inspected or traversed before it is spliced back into the program. The representation can be treated in an opaque manner which gives us more implementation freedom about its form. In our calculus this is evidenced by the fact there is no reduction rule which reduces inside a quotation.

The program evaluation semantics evaluate each declaration in turn from top to bottom. Top-level definitions are evaluated to values and substituted into the rest of the program. Top-level splice definitions are evaluated to a value of type $Code\ \tau$ which has the form $[\![\ e\ ]\!]_{SP}$. The splice variable is bound to the value e with the substitution SP applied and then substituted into the remainder of the program.

Using splice environments and top-level splice definitions is reminiscent of the approach taken in logically inspired languages by Nanevski [2002] and Davies and Pfenning [2001].

5.4 Module Restriction

In GHC, the module restriction dictates that only identifiers bound in other modules can be used inside top-level splices. This restriction is modelled in our calculus by the restriction that only identifiers previously bound in top-level definitions can be used inside a top-level splice. The intention is therefore to consider each top-level definition to be defined in its own "module" which is completely evaluated before moving onto the next definition.

This reflects the situation in a compiler such as GHC which supports separate compilation. When compiling a program which uses multiple modules, one module may contain a top-level splice

$$
\begin{array}{lll}
pgm & ::= e : \tau \mid def ; pgm \mid spdef ; pgm & \\
def & ::= \textbf{def}\ k : \tau = e & \\
spdef & ::= \textbf{spdef}\ \Delta \vdash^{n} sp : \tau = e & \\
e, t & ::= & \text{elaborated expressions} \\
 & \quad x \mid sp \mid k & \text{variables / splice variables / globals} \\
 & \mid \lambda x : \tau . e \mid e\ e & \text{abstraction / application} \\
 & \mid \Lambda a . e \mid e\ \langle \tau \rangle & \text{type abstraction / application} \\
 & \mid [\![e]\!]_{SP} & \text{quotation} \\
SP & ::= \bullet \mid SP, \Delta \vdash sp : \tau = e & \text{splice environment} \\
\tau & ::= & \text{core types} \\
 & \quad a \mid H & \text{variables / constants} \\
 & \mid \tau \rightarrow \tau & \text{functions} \\
 & \mid Code\ \tau & \text{representation} \\
 & \mid \forall a . \tau & \text{quantification} \\
\Gamma, \Delta & ::= \bullet \mid \Gamma, x : (\tau, n) \mid \Gamma, sp : (\Delta, \tau, n) \mid \Gamma, a & \text{type environment} \\
P & ::= \bullet \mid P, k : \tau \mid P, sp : (\Delta, \tau) & \text{program environment}
\end{array}
$$

Fig. 10. Core Language Syntax

which is then used inside another top-level splice in a different module. Although syntactically both top-level splices occur at level -1, they effectively occur at different stages. In the same way as different modules occur at different stages due to separate compilation, in our calculus, each top-level definition can be considered to be evaluated at a new stage.

At this point it is worthwhile to consider what exactly we mean by compile-time and run-time. So far we have stated that the intention is for splices at negative levels to represent compile-time evaluation so we should state how we intend this statement to be understood in our formalism. The elaboration procedure will elaborate each top-level definition to zero or more splice definitions (one for each top-level splice it contains) followed by a normal value definition. Then, during the evaluation of the core program the splice definitions will be evaluated prior to the top-level definition that originally contained them.

The meaning of "compile-time" is therefore that the evaluation of the top-level splice happens before the top-level definition is evaluated. It is also possible to imagine a semantics which partially evaluates a program to a residual by computing and removing as many splice definitions as possible.

If we were to more precisely model a module as a collection of definitions then the typing rules could be modified to only allow definitions from previously defined modules to be used at the top-level and all splice definitions could be grouped together at the start of a module definition before any of the value definitions. Then it would be clearly possible to evaluate all of the splice definitions for a module before commencing to evaluate the module definitions.

6 ELABORATION

In this section we describe the process of elaboration from the source language to the core language. Elaboration starts from a well-typed term in the source language and hence is defined by induction over the typing derivation tree (Figure 2).

6.1 Elaboration procedure

There are three key aspects of the elaboration procedure:

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

$$\frac{\Delta \vdash sp{:}(\tau, n) \in \Gamma \qquad \Delta \subseteq \Gamma}{P;\Gamma \vdash^n sp : \tau}\ \text{E\_Splice\_Var} \qquad \frac{sp{:}(\Delta, \tau) \in P \qquad \Delta \subseteq \Gamma}{P;\Gamma \vdash^n sp : \tau}\ \text{E\_Top\_Splice\_Var}$$

$$\frac{P;\Gamma, a \vdash^n e : \tau}{P;\Gamma \vdash^n \Lambda a.e : \forall a.\tau}\ \text{E\_TAbs} \qquad \frac{P;\Gamma \vdash^n e : \forall a.\tau_2}{P;\Gamma \vdash^n e\ \langle \tau_1 \rangle : \tau_2[\tau_1/a]}\ \text{E\_TApp}$$

$$\frac{P;\Gamma, \overline{sp_i{:}(\Delta_i, \tau_i, n+1)} \vdash^{n+1} e : \tau \qquad P;\Gamma \vdash^n_{\text{SP}} SP \qquad SP = \overline{\Delta_i \vdash sp_i{:}\tau_i = e_i}}{P;\Gamma \vdash^n [\![\ e\ ]\!]_{SP} : \mathit{Code}\ \tau}\ \text{E\_Quote}$$

Fig. 11. Core Expression Typing

$$\boxed{P;\Gamma \vdash^n_{\text{SP}} SP}$$

$$\frac{}{P;\Gamma \vdash^n_{\text{SP}} \bullet}\ \text{SP\_Empty} \qquad \frac{P;\Gamma \vdash^n_{\text{SP}} SP \qquad P;\Gamma, \Delta \vdash^n e : \mathit{Code}\ \tau}{P;\Gamma \vdash^n_{\text{SP}} SP, \Delta \vdash sp{:}\tau = e}\ \text{SP\_Cons}$$

Fig. 12. Splice Environment Typing

$$\boxed{P \vdash_{\text{def}} def : P} \qquad \boxed{P \vdash_{\text{spdef}} spdef : P}$$

$$\frac{P;\bullet \vdash^0 e : \tau}{P \vdash_{\text{def}} \textbf{def}\ k{:}\tau = e : P, k{:}\tau}\ \text{C\_Def} \qquad \frac{P;\Gamma \vdash^n e : \mathit{Code}\ \tau}{P \vdash_{\text{spdef}} \textbf{spdef}\ \Gamma \vdash^n sp{:}\tau = e : P, sp{:}(\Gamma, \tau)}\ \text{C\_SpDef}$$

Fig. 13. Core Definition Typing

$$\boxed{P \vdash_{\text{pgm}} pgm}$$

$$\frac{P;\bullet \vdash^0 e : \tau}{P \vdash_{\text{pgm}} e{:}\tau}\ \text{CP\_Main} \qquad \frac{P_1 \vdash_{\text{def}} def : P_2 \qquad P_2 \vdash_{\text{pgm}} pgm}{P_1 \vdash_{\text{pgm}} def; pgm}\ \text{CP\_Def}$$

$$\frac{P_1 \vdash_{\text{spdef}} spdef : P_2 \qquad P_2 \vdash_{\text{pgm}} pgm}{P_1 \vdash_{\text{pgm}} spdef; pgm}\ \text{CP\_SpDef}$$

Fig. 14. Core Program Typing

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

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2}\ \text{D\_E\_App\_L} \qquad \frac{e_2 \rightsquigarrow e_2'}{e_1\ e_2 \rightsquigarrow e_1\ e_2'}\ \text{D\_E\_App\_R} \qquad \frac{}{(\lambda x{:}\tau.e)\ v \rightsquigarrow e[v/x]}\ \text{D\_E\_Beta}$$

$$\frac{e \rightsquigarrow e'}{e\ \langle \tau \rangle \rightsquigarrow e'\ \langle \tau \rangle}\ \text{D\_E\_TApp} \qquad \frac{}{(\Lambda a.e)\ \langle \tau \rangle \rightsquigarrow e[\tau/a]}\ \text{D\_E\_TBeta} \qquad \frac{SP \rightsquigarrow SP'}{[\![\ e\ ]\!]_{SP} \rightsquigarrow [\![\ e\ ]\!]_{SP'}}\ \text{D\_E\_Quote}$$

Fig. 15. Dynamic Semantics

$$\boxed{SP \rightsquigarrow SP}$$

$$\frac{SP \rightsquigarrow SP'}{SP, \Delta \vdash sp : \tau = e \rightsquigarrow SP', \Delta \vdash sp : \tau = e}\ \text{D\_SP\_Body} \qquad \frac{e \rightsquigarrow e'}{SP, \Delta \vdash sp : \tau = e \rightsquigarrow SP, \Delta \vdash sp : \tau = e'}\ \text{D\_SP\_Cons}$$

Fig. 16. Dynamic Semantics for Splice Environment

$$\boxed{def \rightsquigarrow def} \qquad \frac{e \rightsquigarrow e'}{\textbf{def}\ k : \tau = e \rightsquigarrow \textbf{def}\ k : \tau = e'}\ \text{D\_Def}$$

$$\boxed{spdef \rightsquigarrow spdef} \qquad \frac{e \rightsquigarrow e'}{\textbf{spdef}\ \Gamma \vdash^n sp : \tau = e \rightsquigarrow \textbf{spdef}\ \Gamma \vdash^n sp : \tau = e'}\ \text{D\_SpDef}$$

Fig. 17. Dynamic Semantics for Definitions

$$\boxed{pgm \rightsquigarrow pgm}$$

$$\frac{def \rightsquigarrow def'}{def; pgm \rightsquigarrow def'; pgm}\ \text{D\_P\_Def} \qquad \frac{spdef \rightsquigarrow spdef'}{spdef; pgm \rightsquigarrow spdef'; pgm}\ \text{D\_P\_SpDef} \qquad \frac{e \rightsquigarrow e'}{e : \tau \rightsquigarrow e' : \tau}\ \text{D\_P\_Main}$$

$$\frac{}{\textbf{def}\ k : \tau = v; pgm \rightsquigarrow pgm[v/k]}\ \text{D\_P\_Def\_Beta}$$

$$\frac{}{\textbf{spdef}\ \Delta \vdash^n sp : \tau = [\![\ e\ ]\!]_{SP}; pgm \rightsquigarrow pgm[e[SP]/sp]}\ \text{D\_P\_SpDef\_Beta}$$

Fig. 18. Dynamic Semantics for Program

(1) Splices at positive levels are removed in favour of a splice environment. The elaboration process returns a splice environment which is attached to the quotation form.
(2) Splices at non-positive levels are elaborated to top-level splice definitions, which are bound prior to the expression within which they were originally contained.
(3) Type class constraints are converted to explicit dictionary passing. We describe how to understand the new constraint form *CodeC C* in terms of quotation.

In order to support elaboration, all implicit evidence from the program theory and local environment are annotated with variables. The idea is that in the elaborated program, the evidence for each particular construct will be bound to a variable of that name. Top-level evidence by top-level variables and local evidence by λ-bound variables. The modifications to the typing rules for the source language syntax necessary in order to explain elaboration are highlighted by a grey box.

$$\Gamma ::= \bullet \mid \Gamma, x : (\tau, n) \mid \Gamma, a \mid \Gamma,\ ev : (C, n) \qquad \text{type environment}$$
$$P ::= \bullet \mid P,\ ev : (\forall a.\overline{C} \Rightarrow C) \mid P, k : \sigma \qquad \text{program environment}$$

The judgement $P; \Gamma \vdash^n e : \tau \rightsquigarrow t \mid TSP$ states that in program theory P and environment Γ, the expression e has type τ at level n and elaborates to term t whilst producing splices TSP.

6.2 Splice Elaboration

The *TSP* is a function from a level to a splice environment *SP*. In many rules, we perform level-pointwise union of produced splices, written $\cup$.

During elaboration, all splices are initially added to the *TSP* at the level of their contents, so a splice that occurs at level n is added at level $n-1$ by means of the operation $\cup_{n-1}$ as in rule E_Splice.

What happens with the splices contained in the *TSP* depends on their level. If they occur at a positive level, they will be bound by a surrounding quotation in rule E_Quote. The notation TSP_n denotes the projection of the splices contained in *TSP* at level n. They become part of the splice environment associated with the quotation. Via $\lfloor TSP \rfloor^n$, we then truncate *TSP* so that it is empty at level n and above.

If a splice occurs at non-positive level, it is a top-level splice and will become a top-level splice definition in rules Def or Inst, in such a way that the splice definitions are made prior to the value definition which yielded them. The splice definitions are created by the *collapse* judgement (Figure 19), which takes a splice environment returned by the expression elaboration judgement and creates top-level splice declarations for each negative splice which appeared inside a term. To guarantee a stage-correct execution, the splices are inserted in order of their level.

We maintain the invariant on the *TSP* where it only contains splices of levels prior to the current level of the judgement. Therefore, when the judgement level decreases as in E_Quote, the splices for that level are removed from the splice environment and bound at the quotation. As the top-level of the program is at level 0, the splice environment returned by an elaboration judgement at level 0 will only contain splices at negative levels, which is why the appeal to the *collapse* function starts from level -1.

6.3 Constraint Elaboration

The second point of interest are the constraint elaboration rules given in Figure 3. Since in our language type classes have just a single method, we use the function corresponding to the method itself as evidence for a class instance in rule Inst.

Constraints of the new constraint form $CodeC\ C$ are elaborated into values of type $Code\ \tau$. Therefore inspecting the entailment elaboration form C_Decr and C_Incr must be understood in terms of quotation. The C_Decr entailment rule is implemented by a simple quotation and thus similar to E_Quote. The C_Incr rule is conceptually implemented using a splice, but as the core language does not contain splices it is understood by adding a new definition to the splice environment, which mirrors E_Splice. These rules explains the necessity of level-indexing constraints in the source language; the elaboration would not be well-staged if the stage discipline was not enforced.

The remainder of the elaboration semantics which elaborate the simple terms and constraint forms are fairly routine.

6.4 Example

As an example of elaboration, let us consider the expression $\$(c_1')$ where $c_1' = [\![\ show\ ]\!]$ from Example C1. There are two points of interest here: there is a top-level splice which will be floated to a top-level splice definition, and the $CodeC\ (Show\ a)$ constraint of c_1' must be elaborated into quoted evidence using rule C_Decr.

We assume that the program environment contains c_1':

$$P = \bullet, c_1' : \forall a.CodeC\ (Show\ a) \Rightarrow Code\ (a \to String)$$

As our program comprises a single main expression, we have to use rule P_Expr. This rule requires us to first elaborate the expression itself at level 0 in an empty type environment. We obtain

$$P; \bullet \vdash^0 \$(c_1') : \forall a.Show\ a \Rightarrow a \to String$$
$$\rightsquigarrow \Lambda a.\lambda ev : a \to String.sp \mid \{\Delta \vdash sp : a \to String = c_1'\ \langle a \rangle\ [\![\ ev\ ]\!]_\bullet\} \cup_{-1} \bullet$$

$$\boxed{collapse(n, TSP, pgm) = ds}$$

$$\frac{}{collapse(n, \bullet, pgm) = pgm}\ \text{C\_Empty}$$

$$\frac{ds = collapse(n-1, \lfloor TSP \rfloor^n, \overline{spdef_i}; pgm) \qquad \overline{spdef_i = \textbf{spdef}\ \Delta_i \vdash^n sp_i : \tau_i = e_i} \qquad TSP_n = \overline{\Delta_i \vdash sp_i : \tau_i = e_i}}{collapse(n, TSP, pgm) = ds}\ \text{C\_Strip}$$

Fig. 19. Definition of *collapse*

$$\boxed{\Gamma \rightsquigarrow \Delta}$$

$$\frac{}{\bullet \rightsquigarrow \bullet}\ \text{TE\_Empty} \qquad \frac{\Gamma \rightsquigarrow \Delta \qquad \Gamma \vdash_{\text{ty}} \tau \rightsquigarrow \tau'}{\Gamma, x:(\tau, n) \rightsquigarrow \Delta, x:(\tau', n)}\ \text{TE\_Var} \qquad \frac{\Gamma \rightsquigarrow \Delta}{\Gamma, a \rightsquigarrow \Delta, a}\ \text{TE\_TyVar}$$

$$\frac{\Gamma \rightsquigarrow \Delta \qquad \Gamma \vdash_{\text{ct}} C \rightsquigarrow \tau}{\Gamma, ev:(C, n) \rightsquigarrow \Delta, ev:(\tau, n)}\ \text{TE\_Ctx}$$

Fig. 20. Elaboration of Type Environments

where

$$\Delta = \bullet, a, ev:(a \to String, 0)$$

The splice point *sp* has been introduced for the splice via rule E_Splice. Evidence for the *Show* constraint is introduced into the type environment at level 0 via rule E_C_Abs. It is captured in Δ for use in the splice. Because the use of the evidence occurs at level -1 and the required constraint is *CodeC Show*, rule C_Decr is used to quote the evidence. The splice environment attached to the quote is empty, because there are no further splices.

Back to rule P_Expr, we furthermore obtain that the resulting main expression is of the form

$$p = \Lambda a.\lambda ev{:}\, a \to String.sp{:}\forall a.(a \to String) \to a \to String$$

The type of p results from elaborating the original *Show a* constraint to the type of its method $a \to String$ via the rule C_TC. We can now look at *collapse* and observe that it will extract the one splice at level -1 passed to it into a top-level splice definition that ends up before d, the result being

$$\textbf{spdef}\ \Delta \vdash^{-1} sp{:}\, a \to String = c_1'\ \langle a \rangle\ [\![\ ev\ ]\!]_\bullet; p$$

The operational semantics for the language evaluates the definition of *sp* first to a quotation before the quoted expression is substituted into the remainder of the program so that evaluation can continue.

7 PRAGMATIC CONSIDERATIONS

Now that the formal developments are complete and the relationship between the source language and core language has been established by the elaboration semantics, it is time to consider how our formalism interacts with other language extensions, and to discuss implementation issues.

7.1 Integration into GHC

The implementation and integration of this specification into GHC's sophisticated architecture naturally requires certain design decisions to be made, which we now discuss.

7.1.1 Type Inference. Type inference for our new constraint form should be straightforward to integrate into the constraint solving algorithm used in GHC [Vytiniotis et al. 2011]. The key modification is to keep track of the level of constraints and only solve goals with evidence at the right level. If there is no evidence available for a constraint at the correct level, then either the C_Incr or C_Decr rule can be invoked in order to correct the level of necessary evidence.

7.1.2 Substituting Types. Substituting inside quotations poses some implementation challenges depending on the quotation representation. In previous implementations which used low-level representations to represent quotations [Pickering et al. 2019a; Roubinchtein 2015] the solution was to maintain a separate environment for free variables which can be substituted into without having to implement substitution in terms of the low-level representation. The idea is then that by the time the low-level representation is evaluated all the variables bound in the environment are already added to the environment and so computation can proceed as normal when it encounters what was previously a free variable.

Therefore we should treat every free type variable in a quotation as a splice point, create an environment to attach to the quotation which maps the splice point to the variable of the name and then when the quotation is interpreted substitute the type into the quotation.

7.1.3 Erasing Types. The operational semantics of System F do not depend on the type information and so the types can be erased before evaluation. It is this observation that leads us to accept Example TV1. However, there is another point in the design space – we could have elaborated to an erased version of System F where types were replaced by placeholders. This would save us the complications of having to substitute types inside quotations. However, the option of maintaining the type information is important for practical purposes. GHC has an optional internal typechecking phase called core lint which verifies the correctness of code generation and optimisation, it would be a shame to lose this pass in any program which used metaprogramming. In a language where the type information dwarfs the runtime content of terms, it would be desirable to also explore the option to store erased or partially erased terms [Brady et al. 2003; Jay and Peyton Jones 2008].

7.1.4 Cross-Stage Persistence. Earlier work on cross-stage persistence [Pickering et al. 2019a] has suggested that implementing cross-stage persistence for instance dictionaries should be possible because a dictionary was ultimately a collection of top-level functions so some special logic could be implemented in the compiler to lift a dictionary.

Additional complexity arises when functions with local constraints are passed an arbitrary dictionary which could have been constructed from other dictionaries, which in turn come from dictionaries. At the point the dictionary is passed, the required information about its structure has been lost so it is impossible to interpret into a future stage. A solution to this could be to use a different evidence form which passes the derivation tree for a constraint to a function as evidence before the function constructs the required dictionary at the required stage. This would increase the runtime overhead of using type classes and would not be practical to implement.

7.2 Interaction with Existing Features

So far we have considered how metaprogramming interacts with qualified types, but of course there are other features that are specific to GHC that need to be considered.

7.2.1 GADTs. Local constraints can be introduced by pattern matching on a GADT. For simplicity our calculus did not include GADTs or local constraints but they require similar treatment to other constraints introduced locally. The constraint solver needs to keep track of the level that a GADT pattern match introduces a constraint and ensure that the constraint is only used at that level.

Note that this notion of a "level" is the stage of program execution where the constraint is introduced and not the same idea of a level the constraint solver uses to prevent existentially quantified type variables escaping their scope. Each nested implication constraint increases the level so type variables introduced in an inner scope are forbidden from unifying with type variables which are introduced at a previous level.

7.2.2 Quantified Constraints. The quantified constraints extension [Bottu et al. 2017] relaxes the form of the constraint schemes allowed in method contexts to also allow the quantified and implication forms which in our calculus are restricted to top-level axioms. Under this restriction there are some questions about how the *CodeC* constraint form should interact especially with implication constraints. In particular, whether constraint entailment should deduce that $CodeC\ (C_1 \Rightarrow C_2)$ entails $CodeC\ C_1 \Rightarrow CodeC\ C_2$ or the inverse and what consequences this has for type inference involving these more complicated constraint forms.

7.3 Interaction with Future Features

GHC is constantly being improved and extended to encompass new and ambitious features, and so we consider how metaprogramming should interact with features that are currently in the pipeline.

7.3.1 Impredicativity. For a number of the examples that we have discussed in this paper, an alternative would be to use impredicative instantiation to more precisely express the binding position of a constraint. For instance, the function $c_1 :: Show\ a \Rightarrow Code\ (a \rightarrow String)$ from Example C1 might instead have been expressed in the following manner:

$$c_1' :: Code\ (\forall a.Show\ a \Rightarrow a \rightarrow String)$$
$$c_1' = [\![\ show\ ]\!]$$

The type of c_1' now binds and uses the *Show a* constraint at level 1 without the use of *CodeC*.

However, despite many attempts [Peyton Jones et al. 2007], GHC has never properly supported impredicative instantiation due to complications with type inference. Recent work [Serrano et al. 2020, 2018] has proposed the inclusion of restricted impredicative instantiations, and these would accept c_1'. In any case, impredicativity is not a silver bullet, and there is still a need for the *CodeC* constraint form. Without the *CodeC* constraint form there is no way to manipulate "open" constraints. That is, constraints which elaborate to quotations containing free variables.

Experience has taught us that writing code generators which manipulate open terms is a lot more convenient than working with only closed terms. We predict that the same will be true of working with the delayed constraint form as well. In particular, features such as super classes, instance contexts and type families can be used naturally with *CodeC* constraints. So whilst relaxing the impredicativity restriction will have some positive consequences to the users of Typed Template Haskell, it does not supersede our design but rather acts as a supplement.

7.3.2 Dependent Haskell. Our treatment of type variables is inspired by the in-built phase distinction of System F. As Haskell barrels at an impressive rate to a dependently typed language [Eisenberg 2016; Gundry 2013], the guarantees of the phase distinction will be lost in some cases. At this point it will be necessary to revise the specification in order to account for the richer phase structure.

The specification for Dependent Haskell [Weirich et al. 2017] introduces the so-called relevance quantifier in order to distinguish between relevant and irrelevant variables. The irrelevant quantifier is intended to model a form of parametric polymorphism like the $\forall$ in System F, the relevant quantifier is written as Π after the dependent quantifier from dependent type theory.

Perhaps it is sound to modify their system in order to enforce the stage discipline for relevant type variables not irrelevant ones. It may be that the concept of relevance should be framed in

terms of stages, where the irrelevant stage proceeds all relevant and computation stages – in which case it might be desirable to separate the irrelevant stage itself into multiple stages which can be evaluated in turn. The exact nature of this interaction is left as a question for future work.

8 RELATED WORK

Multi-stage languages with explicit staging annotations were first suggested by Taha and Sheard [1997, 2000]. Since then there has been a reasonable amount of theoretical interest in the topic which has renewed in recent years. There are several practical implementations of multi-stage constructs in mainstream programming languages such as BER MetaOCaml [Kiselyov 2014], Typed Template Haskell and Dotty [Stucki et al. 2018].

At a first glance there is surprisingly little work which attempts to reconcile multi-stage programming with language features which include polymorphism. Most presented multi-staged calculi are simply typed despite the fact that all the languages which practically implement these features support polymorphism. The closest formalism is by Kokaji and Kameyama [2011] who consider a language with polymorphism and control effects. Their calculus is presented without explicit type abstraction and application. There is no discussion about cross-stage type variable references or qualified types and their primary concern, similar to Kiselyov [2017] is the interaction of the value restriction and staging. Our calculus in contrast, as it models Haskell, does not contain any effects so we have concentrated on qualified types. Calcagno et al. [2003] present a similar ML-like language with let-generalisation.

Combining together dependent types and multi-stage features is a more common combination. The phase distinction is lost in most dependently typed languages as the typechecking phase involves evaluating expressions. Therefore in order to ensure a staged evaluation, type variables must also obey the same stage discipline as value variables. This is the approach taken by Kawata and Igarashi [2019]. Pašalic [2004] defines the dependently-typed multi-stage language Meta-D but doesn't consider constraints or parametric polymorphism. Concoqtion [Fogarty et al. 2007] is an extension to MetaOCaml where Coq terms appear in types. The languge is based on $\lambda_{H\bigcirc}$ [Pašalic et al. 2002] which includes dependent types but is motivated by removing tags in the generated program. Brady and Hammond [2006] observe similarly that it is worthwhile to combine together depedent types and multi-stage programming to turn a well-typed interpreter into a verified compiler. The language presented does not consider parametric polymorphism nor constraints.

We are not aware of any prior work which considers the implications of relevant implicit arguments formally, although there is an informal characterization by Pickering et al. [2019a].

Formalising Template Haskell. With regards to formalisms of Template Haskell, a brief description of Untyped Template Haskell is given by Sheard and Jones [2002]. The language is simply-typed and does not account for multiple levels. The language has also diverged since their formalism as untyped quotations are no longer typechecked before being converted into their representation. Their formalism does account for the Q monad which provides operations that allow a programmer to "reify" types, declarations and so on in order to inspect the internal structure. These are typically used in untyped Template Haskell in order to generically define instances or other operations.

It is less common to use the reification functions in Typed Template Haskell programs and so we have avoided their inclusion in our formalism for the sake of simplicity. In our calculus we wanted to precisely understand the basic interaction between constraints and quotations, and the existence of the Q monad is orthogonal to this.

Code generators are typically effectful in order to support operations such as let insertion or report errors so it is an important question to define a calculus with effects. From GHC 8.12, the type of quotations will be generalised [Pickering 2019] from Q ($TExp\ a$) to a minimal interface

$\forall m.Quote\ m \Rightarrow m\ (TExp\ a)$ so a user will have more control over which effects they are allowed to use in their code generators. We leave formalising this extension open to future work.

Metaprogramming In GHC. GHC implements many different forms of metaprogramming from the principled to the ad-hoc. At the principled end of the spectrum in a similar vein as Typed Template Haskell there is Cloud Haskell [Epstein et al. 2011], which implements a modality for distributed computing. Another popular principled style is generic programming [Magalhães et al. 2010; Rodriguez et al. 2008] which allows the representation of datatypes to be inspected and interpreted at runtime. Untyped Template Haskell is used for untyped code generation in the combinator style. As well as generating expressions it can be used to generate patterns, declarations and types. Programs are typechecked after being generated rather than typechecking the generators as in Typed Template Haskell. Untyped Template Haskell has a limited interface into the typechecker but Source Plugins [Pickering et al. 2019b] allow unfettered access to the internal state and operations of the typechecker and other compiler phases.

Modal Type Systems. Type systems motivated by modal logics have more commonly contemplated the interaction of modal operators and polymorphism. In particular attention has turned recently to investigating dependent modal type theories and the complex interaction of modal operators in such theories [Gratzer et al. 2020]. It seems probable that ideas from this line of research can give a formal account of the interaction of the code modality [Davies and Pfenning 2001] and the parametric quantification from System F which can also be regarded as a modality [Nuyts and Devriese 2018; Pfenning 2001].

In recent times, Fitch-Style Modal calculi [Clouston 2018; Gratzer et al. 2019] have become a popular way of specifying a modal type system due to their good computational properties. It would be interesting future work to attempt to modify our core calculus to a Fitch-Style system which was not level indexed. In particular the calculus for Simple RaTT [Bahr et al. 2019] looks like a good starting point.

In short, we are sure there is a lot to learn from the vast amount of literature on modal type systems but we are not experts in this field and the literature does not deal with our practical concerns regarding implementing and writing programs in staged programming languages. The goal of this paper was not to uncover a logically inspired programming language but to give a practical and understandable practical specification which can be understood by people without extensive background in modal type theories.

9 CONCLUSION

Now that we have presented the first formalism of Typed Template Haskell, the way is clear for future researchers to understand and extend the basic system. We envisage that the system will be useful for two different communities. Firstly, researchers into extensions such as Linear Haskell [Bernardy et al. 2017] and Dependent Haskell [Weirich et al. 2017] now have the possibility to consider how their features interact with stages so that the language remains sound with respect to staged evaluation. Secondly, users interested in multi-stage programming now have a firmer foundation to base further extensions to the multi-stage features in GHC.

In the long-term our hope is to make multi-stage programming a more popular and accessible paradigm for functional programmers. Even with the present implementation of Typed Template Haskell, Yallop et al. [2018] have shown how different features implemented in GHC can be used together in order to express elegant code generators. We anticipate that a firm foundation for Typed Template Haskell that supports finer control over the type of qualified constraints will enable more practitioners to explore this unexplored territory, and begin to reach for staging as a useful tool in their toolbox of techniques to enhance the predictability and performance of their code.

REFERENCES

Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: A Fitch-Style Modal Calculus for Reactive Programming without Space Leaks. *Proc. ACM Program. Lang.* 3, ICFP, Article 109 (July 2019), 27 pages. https://doi.org/10.1145/3341713

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Oxford, UK) *(Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 148–161. https://doi.org/10.1145/3122955.3122967

Edwin Brady and Kevin Hammond. 2006. A Verified Staged Interpreter is a Verified Compiler. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering* (Portland, Oregon, USA) *(GPCE '06)*. Association for Computing Machinery, New York, NY, USA, 111–120. https://doi.org/10.1145/1173706.1173724

Edwin Brady, Conor McBride, and James McKinna. 2003. Inductive families need not store their indices. In *International Workshop on Types for Proofs and Programs*. Springer, 115–129. https://doi.org/10.1007/978-3-540-24849-1_8

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* (Erfurt Germany) *(GPCE03)*. Association for Computing Machinery, 57–76. https://doi.org/10.5555/954186.954190

Luca Cardelli. 1988. Phase distinctions in type theory. Unpublish Manuscript.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. *SIGPLAN Not.* 40, 9 (Sept. 2005), 241–253. https://doi.org/10.1145/1090189.1086397

Ranald Clouston. 2018. Fitch-style modal lambda calculi. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 258–275. https://doi.org/10.1007/978-3-319-89366-2_14

Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (May 2001), 555–604. https://doi.org/10.1145/382780.382785

Richard A Eisenberg. 2016. *Dependent types in Haskell: Theory and practice*. University of Pennsylvania.

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

Seth Fogarty, Emir Pasalic, Jeremy Siek, and Walid Taha. 2007. Concoqtion: Indexed Types Now!. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Nice, France) *(PEPM '07)*. Association for Computing Machinery, New York, NY, USA, 112–121. https://doi.org/10.1145/1244381.1244400

Daniel Gratzer, GA Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. (2020). In submission.

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a Modal Dependent Type Theory. *Proc. ACM Program. Lang.* 3, ICFP, Article 107 (July 2019), 29 pages. https://doi.org/10.1145/3341711

Adam Gundry. 2013. *Type inference, Haskell and dependent types*. Ph.D. Dissertation. University of Strathclyde.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (1996), 109–138. https://doi.org/10.1145/227699.227700

Barry Jay and Simon Peyton Jones. 2008. Scrap your type applications. In *International Conference on Mathematics of Program Construction*. Springer, 2–27. https://doi.org/10.1007/978-3-540-70594-9_2

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 637–653. https://doi.org/10.1145/2660193.2660241

Akira Kawata and Atsushi Igarashi. 2019. A Dependently Typed Multi-stage Calculus. In *Asian Symposium on Programming Languages and Systems*. Springer, 53–72. https://doi.org/10.1007/978-3-030-34175-6_4

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. https://doi.org/10.1007/978-3-319-07151-0_6

Oleg Kiselyov. 2017. Generating Code with Polymorphic let: A Ballad of Value Restriction, Copying and Sharing. *Electronic Proceedings in Theoretical Computer Science* 241 (Feb 2017), 1–22. https://doi.org/10.4204/eptcs.241.1

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

Yuichiro Kokaji and Yukiyoshi Kameyama. 2011. Polymorphic multi-stage language with control effects. In *Asian Symposium on Programming Languages and Systems*. Springer, 105–120. https://doi.org/10.1007/978-3-642-25318-8_11

Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A Typed, Algebraic Approach to Parsing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 379–393. https://doi.org/10.1145/3314221.3314625

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A Generic Deriving Mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell* (Baltimore, Maryland, USA) *(Haskell '10)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1863523.1863529

Aleksandar Nanevski. 2002. Meta-Programming with Names and Necessity. (2002), 206–217. https://doi.org/10.1145/581478.581498

Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 779–788. https://doi.org/10.1145/3209108.3209119

Emir Pašalic. 2004. *The role of type equality in meta-programming*. Ph.D. Dissertation. OGI School of Science & Engineering at OHSU.

Emir Pašalic, Walid Taha, and Tim Sheard. 2002. Tagless Staged Interpreters for Typed Languages. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP '02)*. Association for Computing Machinery, New York, NY, USA, 218–229. https://doi.org/10.1145/581478.581499

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

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *J. Funct. Program.* 17, 1 (Jan. 2007), 1–82. https://doi.org/10.1017/S0956796806006034

Frank Pfenning. 2001. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 221–230.

Matthew Pickering. 2019. Overloaded Quotations. GHC proposal. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0246-overloaded-bracket.rst

Matthew Pickering, Nicolas Wu, and Csongor Kiss. 2019a. Multi-Stage Programs in Context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/3331545.3342597

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

Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. 2008. Comparing Libraries for Generic Programming in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/1411286.1411301

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

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

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Zero-cost Effect Handlers by Staging. (2020). In submission.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. (January 2020). https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/ In submission.

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* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 783–796. https://doi.org/10.1145/3192366.3192389

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

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* (Boston, MA, USA) *(GPCE 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3278122.3278139

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

Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (Amsterdam, The Netherlands) *(PEPM '97)*. ACM, New York, NY, USA, 203–217. https://doi.org/10.1145/258993.259019

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

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

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged Selective Parser Combinators. (2020). In submission.

Jeremy Yallop. 2017. Staged Generic Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110273

Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially-Static Data as Free Extension of Algebras. *Proc. ACM Program. Lang.* 2, ICFP, Article 100 (July 2018), 30 pages. https://doi.org/10.1145/3236795