Staging with Class

A Specification for Typed Template Haskell

NINGNING XIE, University of Cambridge, United Kingdom
MATTHEW PICKERING, Well-Typed LLP
ANDRES LÖH, Well-Typed LLP
NICOLAS WU, Imperial College London, United Kingdom
JEREMY YALLOP, University of Cambridge, United Kingdom
MENG WANG, University of Bristol, United Kingdom

Multi-stage programming using typed code quotation is an established technique for writing optimizing code generators with strong type-safety guarantees. Unfortunately, quotation in Haskell interacts poorly with type classes, making it difficult to write robust multi-stage programs.

We study this unsound interaction and propose a resolution, *staged type class constraints*, which we formalize in a source calculus $\lambda^{[\![\Rightarrow]\!]}$ that elaborates into an explicit core calculus $F^{[\![]\!]}$. We show type soundness of both calculi, establishing that well-typed, well-staged source programs always elaborate to well-typed, well-staged core programs, and prove beta and eta rules for code quotations.

Our design allows programmers to incorporate type classes into multi-stage programs with confidence. Although motivated by Haskell, it is also suitable as a foundation for other languages that support both overloading and quotation.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Semantics**; • **Theory of computation** → **Type theory**;

Additional Key Words and Phrases: Staging, Type Classes, Template Haskell

ACM Reference Format:
Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2021. Staging with Class: A Specification for Typed Template Haskell. 1, 1 (October 2021), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Producing code with predictable performance 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] and used to eliminate abstraction overhead in many domains [Jonnalagedda et al. 2014; Krishnaswami and Yallop 2019; Schuster et al. 2020; Willis et al. 2020; Yallop 2017]. These annotations give programmers fine control over performance by instructing the compiler to generate code in one stage of compilation that can be used in another.

Authors' addresses: Ningning Xie, University of Cambridge, United Kingdom, nx213@cl.cam.ac.uk; Matthew Pickering, Well-Typed LLP, matthew@well-typed.com; Andres Löh, Well-Typed LLP, andres@well-typed.com; Nicolas Wu, Imperial College London, United Kingdom, n.wu@imperial.ac.uk; Jeremy Yallop, University of Cambridge, United Kingdom, jeremy.yallop@cl.cam.ac.uk; Meng Wang, University of Bristol, United Kingdom, meng.wang@bristol.ac.uk.

XXXX-XXXX/2021/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The classic example of staging is *power n k*, 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 *qpower* function shows its corresponding staged version where *Code* annotates the types of values that will be present dynamically at run time. Since k is to be provided as a fixed value at compile time, it remains a value of type *Int*.

```
power :: Int → Int → Int                qpower :: Int → Code Int → Code Int
power 0 n = 1                           qpower 0 qn = ⟦ 1 ⟧
power k n = n * power (k − 1) n         qpower k qn = ⟦ $(qn) * $(qpower (k − 1) qn) ⟧
```

Then in the definition of *power5*, we can quote $n :: Int$ to create $⟦ n ⟧ :: Code\ Int$, and splice the expression $\$(qpower\ 5\ ⟦ n ⟧)$ to generate $n * (n * (n * (n * (n * 1))))$. By using the staged function, static information can be eliminated by partially evaluating the function at compile-time.

```
power5 :: Int → Int
power5 n = $(qpower 5 ⟦ n ⟧)    -- power5 n = n * n * n * n * n * 1
```

The code above is restricted to a fixed type *Int*, and it is natural to hope for a more generic version.

The incarnation of staged programming in Typed Template Haskell promises the benefits of *type classes*, one of the distinguishing features of Haskell [Hall et al. 1996; Peyton Jones et al. 1997], allowing a definition to be reused for any type that is qualified to be numeric:

```
npower :: Num a ⇒ Int → a → a           qnpower :: Num a ⇒ Int → Code a → Code a
npower 0 n = 1                          qnpower 0 qn = ⟦ 1 ⟧
npower k n = n * power (k − 1) n        qnpower k qn = ⟦ $(qn) * $(qnpower (k − 1) qn) ⟧
```

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 9.0.1:

```
npower5 :: Num a ⇒ a → a
npower5 n = $(qnpower 5 ⟦ n ⟧)    -- Error!
```

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 *qnpower*, the constraint is bound outside a quotation but is used inside. As we will see, this discrepancy leads to subtle inconsistencies, which can be used to show that the current implementation of type classes in Typed Template Haskell is *unsound*.

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, while preserving type soundness. In particular, inspired by Typed Template Haskell, we offer the following contributions:

- We formalize a source calculus $\lambda^{[\![\Rightarrow]\!]}$, which models two key features of Typed Template Haskell, type classes and multi-stage programming, and includes a novel construct, *staged type class constraints* that resolves the subtle interaction between the two (§3).
- We formalize a core calculus $F^{[\![]\!]}$, a polymorphic lambda calculus extended with quotations as a compilation target for multi-stage languages (§4). *Splice environments*, a key innovation in $F^{[\![]\!]}$, make evaluation order evident, avoiding the need for level-indexed evaluation, and support treating quotations opaquely, giving more implementation freedom about their form.

- We present a type-directed elaboration from $\lambda^{[\![\Rightarrow]\!]}$ to $F^{[\![]\!]}$, which combines our two key ideas: *dictionary-passing elaboration* of staged type class constraints, and elaboration of splices into splice environments (§5).
- We prove key properties of our formalism: (a) $F^{[\![]\!]}$ enjoys type soundness (§4.4), (b) *well-typed, well-staged source programs always elaborate to well-typed, well-staged core programs*, and thus $\lambda^{[\![\Rightarrow]\!]}$ also enjoys type soundness (§5.4) and (c) splices and quotations are *dual*, building on the *axiomatic semantics* of $F^{[\![]\!]}$ (§6).

§7 provides a detailed comparison of our work here to the current implementation of Template Haskell in GHC. The full proofs of the stated lemmas and theorems are available in the appendix included in the supplementary materials. While this work has been motivated by Typed Template Haskell, we believe our work will be useful to designers and implementors of other languages which combine similar features and share many of the same challenges.

2 OVERVIEW

This section gives an overview of our work. We start by introducing the fundamental concepts of multi-staged programming, in the context of Typed Template Haskell.

2.1 Multi-Stage Programming

Multi-stage programming provides two standard staging annotations that allow construction and combination of program fragments:

- A *quotation* expression $[\![\ e\ ]\!]$ is a representation of the expression e as program fragment in a future stage. If $e :: a$, then $[\![\ e\ ]\!] :: \mathit{Code}\ a$.
- A *splice* expression $\$(e)$ extracts the expression from its representation e. If $e :: \mathit{Code}\ a$, then $\$(e) :: a$. By splicing expressions inside quotations we can construct larger quotations from smaller ones.

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. Unfortunately, things are not so simple: *type soundness* in multi-staged programming also requires programs to be *well-staged*.

2.2 The Level Restriction

The definition of well-stagedness depends on the notion of a *level*. Levels indicate the evaluation order of expressions, and well-stagedness ensures that program can be evaluated in the order of their levels, so that an expression at a particular level can only be evaluated when all expressions it depends on at previous levels have been evaluated. Formally, the *level* of an expression is an integer given by the number of quotes that surround it, minus the number of splices. In other words, starting from level zero, quotation increases the level of an expression while splicing decreases it. The level of an expression indicates when the expression is evaluated: (1) programs of negative levels are evaluated at compile time; (2) programs of level 0 are evaluated at runtime; and (3) programs of positive levels are at future unevaluated stages.

In the simplest setting, a program is well-staged if each variable is used only at the level in which it is bound (hereafter referred to as *the level restriction*). Using a variable in a different stage may simply be impossible, or at least require special attention. The following three example programs are all well-typed, but only the first, *timely*, is well-staged:

$$\mathit{timely} :: \mathit{Code}\ (\mathit{Int} \to \mathit{Int}) \qquad \mathit{hasty} :: \mathit{Code}\ \mathit{Int} \to \mathit{Int} \qquad \mathit{tardy} :: \mathit{Int} \to \mathit{Code}\ \mathit{Int}$$
$$\mathit{timely} = [\![\ \lambda x \to x\ ]\!] \qquad \mathit{hasty} = \lambda y \to \$(y) \qquad \mathit{tardy} = \lambda z \to [\![\ z\ ]\!]$$

In *timely*, the variable x is both introduced and used at level 1. (Similarly, in the well-staged example, *qpower*, in the introduction, the variables *qpower*, k and qn are introduced and used at level 0.) In the second program, *hasty*, the variable y is introduced at level 0, but used at level -1. Evaluating the program would get stuck, because its value is not yet known at level -1. In the third program, *tardy*, the variable z is introduced at level 0, but used at level 1. Using a variable at a later stage in this way requires additional mechanisms to persist its value from one stage to another.

Relaxing the level restriction. Designers of multi-stage languages have developed several mechanisms for relaxing the level restriction to allow references to variables from previous stages [Hanada and Igarashi 2014; Taha and Sheard 1997]. *Lifting* makes a variable available to future stages by copying its value into a future-stage representation. Since lifting is akin to serialisation, it can be done easily for first-order types such as strings and integers, but not higher-order types. *Cross-stage persistence* (CSP) is more general than lifting: it supports embedding references to heap-resident values into quotations. Since it does not involve serialisation, CSP also supports persisting non-serialisable values such as functions and file handles. *Path-based persistence* is a restricted form of CSP for top-level[1] identifiers. Rather than persisting references to heap values, path-based persistence stores identifiers themselves, which can be resolved in the same top-level environment in future stages. For example, the top-level function *power* can be persisted in this way.

This work considers only path-based persistence. Fully-general CSP is limited to systems in which all stages are evaluated in the same process, since it requires sharing of heaps between stages; it is not available in systems such as Typed Template Haskell. Lifting is more broadly applicable, but it is straightforward to add separately as a local rewriting of programs. For example, GHC provides the *Lift* type class with a method *lift*, and instances of *Lift* for basic types like *Int*. Using these facilities, the ill-staged *tardy* can be rewritten into the well-staged *timelyLift*:

```
class Lift a where                timelyLift :: Int → Code Int
  lift :: a → Code a              timelyLift = λx → ⟦ $(lift x) ⟧
```

2.3 Type Classes and the Level Restriction

The examples in the previous section demonstrate the importance of levels in a well-staged program in the simplest setting. However, other features found in real-world languages sometimes interact in non-trivial ways with multi-stage programming support. One such feature is *type classes* [Wadler and Blott 1989], a structured approach to overloading. Unfortunately, naive integration of type classes and staging poses a threat to type soundness. This section presents the problem, after a brief introduction to type classes and their dictionary-passing elaboration.

Type classes and dictionary-passing elaboration. The following presents the elements of type classes: the *Show* class offers an interface with one method *show*, the *Show Int* instance provides an implementation of *Show* for the type *Int* with a primitive *primShowInt*, and the *print* function uses the class method *show*; its type indicates that it can be used at any type a that has a *Show* instance.

```
class Show a where          instance Show Int where      print :: Show a ⇒ a → String
  show :: a → String          show = primShowInt         print x = show x
```

Type classes do not have direct operational semantics; rather, they are implemented by *dictionary-passing elaboration* into a simpler language without type classes (e.g. System F). After elaboration, a type class definition becomes a *dictionary* (i.e. a record type with a field for each class member), an instance becomes a value of the dictionary type, and each function that uses class methods acquires an extra parameter for the corresponding dictionary:

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

$$\textbf{data}\ \mathit{ShowDict}\ a = \mathit{ShowDict}\ \{\mathit{show}' :: a \to \mathit{String}\} \qquad \mathit{showInt} = \mathit{ShowDict}\ \{\mathit{show}' = \mathit{primShowInt}\} \qquad \mathit{print}' :: \mathit{ShowDict}\ a \to a \to \mathit{String}\ \ \mathit{print}'\ \mathit{dShow}\ x = \mathit{show}\ \mathit{dShow}\ x$$

The problem of staging type class methods. Constraints introduced by type classes have the potential to break type soundness, as implicit dictionary passing may not adhere to the level restriction. For example, in the following program, the class method *show* appears inside a quotation. Note the change of the function return type from $a \to \mathit{String}$ to $\mathit{Code}\ (a \to \mathit{String})$[2].

$$\begin{aligned} &\mathit{print1} :: \mathit{Show}\ a \Rightarrow \mathit{Code}\ (a \to \mathit{String}) \\ &\mathit{print1} = [\![\ \mathit{show}\ ]\!] \end{aligned} \tag{C1}$$

Is *print1* well-staged? It appears so, since *print1* only uses the top-level class method *show*, which is path-based persisted. However, a subtle problem reveals itself after type class elaboration:

$$\begin{aligned} &\mathit{print1}' :: \mathit{ShowDict}\ a \to \mathit{Code}\ (a \to \mathit{String}) \\ &\mathit{print1}'\ \mathit{dShow} = [\![\ \mathit{show}\ \mathit{dShow}\ ]\!] \end{aligned}$$

After elaboration, $\mathit{print1}'$ takes an additional dictionary argument $\mathit{dShow} :: \mathit{ShowDict}\ a$. Notice that the dictionary variable *dShow* is introduced at level 0, but is used at level 1! Naively elaborating without considering the *levels of constraints* has introduced a cross-stage reference, making *print1* ill-staged. As §2.2 outlined, one possible remedy is to persist *dShow* between stages, a solution once advocated by [Pickering et al. 2019]. Although dictionaries are typically higher-order, they are ultimately constructed from path-persistable top-level values. However, the additional run-time overhead associated with this approach has led its erstwhile advocates to abandon it as impractical.

In contrast, the following monomorphic definition of *printInt* remains well-staged even after dictionary-passing elaboration into $\mathit{printInt}'$, since the constraint is resolved to a global instance *showInt* (which can be path-based persisted) rather than abstracted as a local variable. But of course this version does not enjoy all the benefits of type classes.

$$\begin{aligned} &\mathit{printInt} :: \mathit{Code}\ (\mathit{Int} \to \mathit{String}) \\ &\mathit{printInt} = [\![\ \mathit{show}\ ]\!] \end{aligned} \qquad \begin{aligned} &\mathit{printInt}' :: \mathit{Code}\ (\mathit{Int} \to \mathit{String}) \\ &\mathit{printInt}' = [\![\ \mathit{show}\ \mathit{showInt}\ ]\!] \end{aligned} \tag{C2}$$

The problem of splicing type class methods. The interaction of *splicing* and dictionary-passing elaboration can also be subtle. In particular, splices that appear in top-level definitions may require class constraints to be used at levels prior to the ones where they are introduced. Consider the definition of *topLift*:

$$\textbf{data}\ C = C \qquad \begin{aligned} &\mathit{topLift} :: \mathit{Lift}\ C \Rightarrow C \\ &\mathit{topLift} = \$(\mathit{lift}\ C) \end{aligned} \qquad \begin{aligned} &\mathit{topLift}' :: \mathit{LiftDict}\ C \to C \\ &\mathit{topLift}'\ \mathit{dLift} = \$(\mathit{lift}\ \mathit{dList}\ C) \end{aligned} \tag{TS1}$$

As with C1, although *topLift* appears to be well-staged, elaboration reveals that it is not, since it produces a future-stage reference inside the splice: the dictionary *dLift* is introduced at level 0 but is used at level −1. Unlike the case of C1, there is no remedy here, and the code should be rejected, as *dLift* is not known until runtime, and thus cannot be used in compile-time evaluation.

2.4 Staging Type Classes: an Exploration of the Design Space

Up to this point we have focused on the problems of type unsoundness arising from the interaction between quotation/splicing and type classes. We now turn to an exploration of potential solutions. Since there is little formal work in this area, our remarks here focus on designs that have been implemented in GHC. This section discusses the problems with each of these designs, and §7 includes a more detailed comparison with GHC.

[2]This example is an eta-reduced version of $\mathit{print1} = [\![\ \lambda x \to \mathit{show}\ x\ ]\!]$. For simplicity, we omit the argument x.

Delaying type class elaboration until splicing. One approach to resolving Example C1 is to delay dictionary-passing elaboration until the program is spliced. With this approach, code values represent *source* programs rather than *elaborated* programs. For C1 this means that *print1* is not elaborated, and so the problem with its ill-staged elaboration *print1′* does not arise. Instead, splicing *print1* first inserts its source code and then performs dictionary-passing elaboration, at which point we can provide the dictionary as per normal.

$$\begin{array}{ll} universe :: String & universe' :: String \\ universe = \$(print1)\ 42 & universe' = show\ showInt\ 42 \end{array}$$

However, as Pickering et al. [2019] observe, not preserving dictionary information in quotations can also threaten soundness. For example, the *readInt* function below uses the built-in function $read :: Read\ a \Rightarrow String \to a$, which converts a *String* into some *Read* instance (e.g. *Int*).

$$\begin{array}{ll} printInt :: Code\ (Int \to String) & readInt :: Code\ (String \to Int) \\ printInt = [\![\ show\ ]\!] & readInt = [\![\ read\ ]\!] \end{array}$$

Like Example C2, we expect that the global instance *readIntPrim* can be used to resolve *Read Int* in *readInt*. If so, then the following function composition would have a clear meaning, which trims spacing around a string representing an integer by first reading it into an integer and then print it:

$$\begin{array}{l} trim :: Code\ (String \to String) \\ trim = [\![\ \$(printInt) \cdot \$(readInt)\ ]\!] \end{array} \tag{A1}$$

Unfortunately, if dictionary information is not preserved in quotations, and we only do dictionary-passing elaboration when splicing *trim*, i.e., in $\$(trim)$, then any use of $\$(trim)$ would be rejected, as its spliced result $print \cdot read$ is a typical example of an *ambiguous type scheme* [Jones 1993], i.e., $print \cdot read$ is of type $(Show\ a, Read\ a) \Rightarrow Code\ (String \to String)$, where the dictionary to be used cannot be decided deterministically. Moreover, even when there is no such ambiguity, this approach may still accidentally change the semantics of a program, for example when the definition site and the splicing site have different instances[3].

Excluding local constraints for top-level splices. One tempting solution to address the problem of splicing-type-class-methods mentioned above (Example TS1) is to exclude local constraints from the scope inside top-level splices. After all, top-level splices require compile time evaluation, and local constraints will not be available during compile time. While this approach can correctly reject TS1, it unfortunately cannot handle the combination of quotations and splices properly. In particular, programs like the following may be unnecessarily rejected.

$$\begin{array}{l} cancel :: Show\ a \Rightarrow a \to String \\ cancel = \$([\![\ show\ ]\!]) \end{array} \tag{A2}$$

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

Impredicativity. Forthcoming versions of GHC are expected to feature *impredicativity*, allowing type variables to be instantiated by polymorphic types [Serrano et al. 2020]. At a first glance, impredicativity appears to resolve the difficulty; furthermore, it naturally extends to include other features such as *quantified constraints* [Bottu et al. 2017].

[3]In GHC, this requires language pragmas for *overlapping instances*, which allows resolving class constraints using more specific instances, and is not uncommon in practice. For example, a module can have both **instance** *Eq* [*Int*] and **instance** *Eq* [*a*], and the former will be used to resolve *Eq* [*Int*] , and the latter can resolve, for example, *Eq* [*Bool*].

For our example, impredicativity allows *print* to be given the following type, indicating that the code returned is polymorphic in the *Show* instance:

$$printImp :: Code\ (Show\ a \Rightarrow a \rightarrow String)$$
$$printImp = [\![\ show\ ]\!]$$

At a small scale, this neatly solves the problem: the type indicates that the constraint *Show a* elaborates to a level 1 parameter, making the generated code well-staged. However, in larger examples, using impredicativity in this way severely limits the flexibility of staged functions. For example, here is an alternative definition of *qnpower* using impredicativity:

$$qnpowerImp :: Int \rightarrow Code\ (Num\ a \Rightarrow a) \rightarrow Code\ (Num\ a \Rightarrow a)$$
$$qnpowerImp\ 0\ qn = [\![\ 1\ ]\!]$$
$$qnpowerImp\ k\ qn = [\![\ \$(qn) * \$(qnpowerImp\ (k-1)\ qn)\ ]\!]$$

As with *printImp*, the types indicate that *qnpowerImp* is well-staged: the positions of the *Num a* constraints beneath *Code* indicate that they elaborate to level 1 parameters. Unfortunately, the type of the parameter *qn* now places additional demands on callers. The unstaged polymorphic *npower* function accepts an expression of any numeric type *a* as its second argument, and it would be convenient for its staged counterpart to accept an expression of any future-stage numeric type *Code a*. Instead, *qnpowerImp* demands an argument of type *Code* (*Num a* ⇒ *a*): even if it is called at a monomorphic type such as *Int*, the argument must still have type *Code* (*Num Int* ⇒ *Int*). This requirement has unfortunate effects on usability: such arguments cannot be of type *Code Int*, since *Code Int* is not a subtype of *Code* (*Num Int* ⇒ *Int*) (in the latest GHC). This is a significant loss of flexibility for callers. Further studies, beyond the scope of this paper, would be needed to support such subtyping while preserving impredicativity. Moreover, the requirement also leads to reduced control over generated code, which will be strewn with many additional dictionary abstractions and applications in generated code involving type classes. It may be possible to eliminate some of these in subsequent compiler passes but many of those passes are based on heuristics. Relying on compiler optimizations does not produce predictable program performance: it is almost impossible to tell by inspection how a program will be optimized.

2.5 Our Proposal: Staged Type Class Constraints

As we have seen, the interaction of staging and type-class elaboration is complicated, which cannot be managed by simply imposing additional restrictions on either one. A targeted solution that properly combines the two processes and restores type soundness is therefore needed.

Our proposal is to introduce *staged type class constraints*, a new constraint form *CodeC C* indicating that the constraint *C* has been *staged*. That is, we can use the staged constraint *CodeC C* to prove a constraint *C* in the next stage. With staged type class constraints, we can establish type soundness by enforcing well-stageness of constraints and dictionaries, and thus ill-staged use of constraints (e.g. *print1* and *topLift*) can be correctly rejected. To illustrate the idea, let us reconsider the problematic example *print1* in C1. We rewrite the example to *print2* with a staged type class constraint in its new type signature as follows.

$$print2 :: CodeC\ (Show\ a) \Rightarrow Code\ (a \rightarrow String) \quad \text{-- originally } Show\ a \Rightarrow Code\ (a \rightarrow String) \qquad \text{(S1)}$$
$$print2 = [\![\ show\ ]\!]$$

This example illustrates the key idea of staged type class constraints. First, during typing, we use the *CodeC* (*Show a*) constraint to resolve the *Show a* constraint raised by *show*. Notably, the *CodeC* (*Show a*) constraint is introduced at level 0 but the *Show a* constraint is resolved at level 1.

That means, staged type classes have the static semantics that *a constraint CodeC C at level n is equivalent to a constraint C at level n + 1*.

Second, in order to elaborate the expression with dictionary-passing, we need a dictionary representation of *CodeC C*. Fortunately, we already have all necessary machinery within the language – since dictionaries become regular data structures after elaboration, staging annotations can effectively convert between a dictionary for *CodeC C* and a dictionary for *C*. That means, staged type class constraints have the simple elaboration semantics that *a dictionary for a constraint CodeC C is a representation of the dictionary for a constraint C*.

Applying this elaboration semantics to *print2* produces the following code:

$$
\begin{aligned}
&print2' :: Code\ (ShowDict\ a) \to Code\ (a \to String)\\
&print2'\ cdShow = [\![\ show\ \$(cdShow)\ ]\!]
\end{aligned}
$$

The type $Code\ (ShowDict\ a)$ is the elaboration of the constraint $CodeC\ (Show\ a)$, and so *cdShow* is the representation of a dictionary, and can be spliced inside the quote as the dictionary argument to *show*. Crucially, the reference to *cdShow* is at the correct level, and so the program is type-safe.

The power function revisited. Recall the *qnpower* example in the introduction (§1):

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

Just as *print1* in Example C1, the definition had to be rejected because of the ill-stagedness of the constraints. Using staged class constraints, we argue that the function *power* should instead have the constraint $CodeC\ (Num\ a)$, which then gets elaborated to $Code\ (NumDict\ a)$:

$$
\begin{aligned}
&qnpower :: CodeC\ (Num\ a) \Rightarrow && qnpower' :: Code\ (NumDict\ a) \to\\
&\quad Int \to Code\ a \to Code\ a && \quad Int \to Code\ a \to Code\ a\\
&qnpower\ 0\ cn = [\![\ 1\ ]\!] && qnpower'\ cdNum\ 0\ cn = [\![\ 1\ ]\!] \qquad \text{(S2)}\\
&qnpower\ k\ cn = && qnpower'\ cdNum\ k\ cn =\\
&\quad [\![\ \$(cn) * \$(qnpower\ (k-1)\ cn)\ ]\!] && \quad [\![\ (*)\ \$(cdNum)\ \$(cn)\ \$(qnpower\ (k-1)\ cn)\ ]\!]
\end{aligned}
$$

The elaboration of *npower5* then shows how *C* can be converted into *CodeC C* by quoting:

$$
\begin{aligned}
&npower5 :: Num\ a \Rightarrow a \to a && npower5' :: NumDict\ a \to a \to a\\
&npower5\ n = \$(qnpower\ 5\ [\![\ n\ ]\!]) && npower5'\ dNum\ n = \$(qnpower'\ [\![\ dNum\ ]\!]\ 5\ [\![\ n\ ]\!])
\end{aligned}
$$

In this case, by quoting *dNum*, the argument to *qnpower′* is a representation of a dictionary (i.e., $[\![\ dNum\ ]\!] :: Code\ (NumDict\ a)$) as will be required by the elaborated type of $CodeC\ (Num\ a)$. Moreover, all variables in the definitions are well-staged.

2.6 Staging with Levels at Runtime

Besides formalizing staged type class constraints, our work also offers a guideline for implementation. In order to provide a robust basis for real-world languages such as Typed Template Haskell, we want our formalism to be easy to implement and to stay close to existing implementations.

One question, then, is how to evaluate staging programs. The *level* of an expression, described earlier, indicates when the expression is evaluated: expressions with negative levels are evaluated at compile time, those with level 0 at runtime, and those with positive levels in future stages. Ensuring a well-staged evaluation order involves access the level information during evaluation. For example, evaluating the following expression at runtime (level 0) involves evaluating e_1 and e_3, but not e_2:

$$(e_1, [\![\ e_2\ \$(e_3)\ ]\!])$$

This is often done by level-indexing the reduction relation [Calcagno et al. 2003; Taha and Sheard 1997]. For example, during evaluation, we can traverse the quotation $[\![\ e_2\ \$(e_3)\ ]\!]$, modifying the

level (initially 0) when quotations or splices are encountered, looking for expressions of level 0 to evaluate. This approach requires tracking of levels during runtime, adding complexity to implementations. Furthermore, as the above example illustrates, it requires inspecting and evaluating inside quotations. But in realistic implementations, quotations are compiled to a representation form for which implementing substitution can be difficult. In particular, previous implementations with low-level representations of quotations [Pickering et al. 2019; Roubinchtein 2015] maintain separate environments for free variables which can be substituted into without having to implement substitution in terms of the low-level representation.

2.7 Our Design: Splice Environments

We present a formalism that is easy to implement and reason about, by introducing quotations with *splice environments* in our core calculus $F^{[]}$. Splice environments capture *splices inside quotations*, avoiding the need to traverse quotations before splicing them into programs, and allowing quotations to be treated in an opaque manner that imposes few constraints on their representation. Splice environments also make the evaluation order of the core calculus evident, avoiding the need for level-indexed reduction. Using splice environments is reminiscent of the approach taken in logically inspired languages by Nanevski [2002] and Davies and Pfenning [2001].

A quotation with a splice environment is denoted $[\![e]\!]_\phi$, where e is a quoted expression and ϕ the splices it contains. ϕ consists of a list of *splice variables*, with each splice variable s represented as a *closure*. For example, our previous expression $(e_1, [\![\ e_2\ \$(e_3)\ ]\!])$ is represented as follows in $F^{[]}$ (assuming e_1, e_2 and e_3 contain no other splices).

$$(e_1, [\![e_2\ s]\!]_{\bullet \vdash^0 s:\tau=e_3})$$

There are several points to note. First, the splice $\$(e_3)$ is replaced by a fresh splice variable s, bound in the splice environment of the quotation. All splices in quotations will be similarly lifted, so that quotations no longer contain splices; in fact, $F^{[]}$ has no splices, only splice environments.

Second, the splice variable s captures four elements:

(1) the spliced expression (e_3).
(2) the type context ($\bullet$). Here the type context is empty, but in general the expression may contain free variables, which the type context tracks.
(3) the level of the expression. Here, e_3 is of level 0.
(4) the type (τ) after splicing. If e_3 is of type Code τ then $\$(e_3)$ is of type τ.

Those elements imply that the splice variable s, representing $\$(e_3)$, is at level 1 and of type τ.

Finally, the splice environment contains only expressions of level 0, and is itself bound to a quotation of the same level (i.e., the whole quotation $[\![e_2\ s]\!]_{\bullet \vdash^0 s:\tau=e_3}$ is at level 0). This is an invariant maintained in the core calculus: a splice is bound immediately to the innermost surrounding quotation at the same level.

Now evaluation can be described straightforwardly, without the need to track levels or inspect quotations. Evaluation initially proceeds as if there is no staging. When it encounters a quotation $[\![e]\!]_\phi$, rather than inspecting e, it evaluates its splice environments ϕ, which are exactly those splices inside the quotation that should be evaluated in the current stage. In the above example, at level 0, evaluation starts with e_1, then proceeds to the quotation $[\![e_2\ s]\!]_{\bullet \vdash^0 s:\tau=e_3}$ and moves to its splice environment $\bullet \vdash^0 s : \tau = e_3$, which in turn evaluates e_3. As this description makes clear, evaluating the expression evaluates e_1 and e_3 as desired. In more complex examples, nested quotations and splices produce nested quotations and splice environments, but the evaluation principle is the same.

Compile-time evaluation and top-level splice definitions. As we have said, splice environments bind each splice to the innermost surrounding quotation at the same level. This scheme does not

account for the case of splices of negative levels which have no such enclosing quotation, such as top-level splices. Since splices of negative levels are exactly those expressions that are evaluated at compile-time, we lift the corresponding splice environments to top-level as *splice definitions*

$$\textbf{spdef} \bullet \vdash^{-1} s : \tau = e$$

and put them *before* the rest of the program. This also gives meaning to *compile-time evaluation* in our formalism, where it is modeled using top-level splice definitions, whose evaluation happens before the rest of the program. We might also imagine a post-elaboration process which partially evaluates a program to a residual by computing and removing these splice definitions. Such a process can be easily implemented separately, so we do not include it in the formalism.

3 $\lambda^{[\![\Rightarrow]\!]}$: MULTI-STAGE PROGRAMMING WITH TYPE CLASSES

We present $\lambda^{[\![\Rightarrow]\!]}$, which has been designed to incorporate the essential features of a language with staging and qualified types, with the key novelty in the formalism of staged type class constraints.

3.1 Syntax

Figure 1 presents the syntax of our source calculus $\lambda^{[\![\Rightarrow]\!]}$. The syntax of type classes follows closely that of Bottu et al. [2017]; Chakravarty et al. [2005]; Jones [1994].

A source program *pgm* is a sequence of top-level definitions $\mathcal{D}$, type class declarations C, and instance definitions $\mathcal{I}$, followed by an expression e. Top-level definitions $\mathcal{D}$ $(k = e)$ model path-based cross-stage persistence: only variables previously defined in a top-level definition can be referenced at arbitrary levels. The syntax of type class declarations C is largely simplified to avoid clutter in the presentation. In particular, type class definitions $\mathsf{TC}\ a\ \textbf{where}\ \{k : \rho\}$ have precisely one method and no superclasses. Instance definitions $\overline{C_i}^{\,i} \Rightarrow \mathsf{TC}\ \tau\ \textbf{where}\ \{k = e\}$ are permitted to have an instance context, which is interpreted that τ is an instance of the type class TC with the method implementation $k = e$, if $\overline{C_i}^{\,i}$ holds. The expression language e is a standard λ-calculus extended with multi-stage annotations, and includes literals i, top-level variables k, variables x, lambdas $\lambda x : \tau.e$, applications $e_1\ e_2$, as well as quotations $[\![e]\!]$ and splicing $\$e$.

Following Jones [1994], the type language distinguishes between monotypes τ, qualified types ρ, and polymorphic types σ. Monotypes τ include type variables a, the integer type Int, function types $\tau_1 \to \tau_2$ and code representation $\mathsf{Code}\ \tau$. Qualified types ρ qualify over monotypes with a list of constraints $(C \Rightarrow \rho)$. Polymorphic types σ are qualified types with universal quantifiers $(\forall a.\sigma)$. Finally, type class constraints are normal constraints $\mathsf{TC}\ \tau$, or staged constraints $\mathsf{CodeC}\ C$.

The program theory Θ is a context of type information for names introduced by top-level definitions $k : \sigma$, and the type class axioms introduced by instance declarations $\forall \overline{a_i}^{\,i}.\overline{C_j}^{\,j} \Rightarrow C$. The context Γ is used for locally introduced information, including value variables $x : (\tau, n)$, type variables a, and local type class axioms (C, n). The context keeps track of the (integer) level n that value and constraint variables are introduced at; the typing rules will ensure that the variables are only used at the current level.

3.2 Typing Expressions

Figure 1 also presents the typing rules for expressions. The judgment $\Theta; \Gamma \vdash^n e : \sigma$ says that under the program theory Θ, the context Γ, and the current level n, the expression e has type σ. The gray parts are for elaboration (§5) and can be ignored until then.

Most typing rules are standard [Bottu et al. 2017; Chakravarty et al. 2005], except that rules are indexed by a level. As emphasized before, level-indexed typing rules ensure that variables and constraint can only be used at the level they are introduced. Literals and top-level variables can

program	pgm	::=	$\textbf{def}\ \mathcal{D}; pgm \mid \textbf{class}\ C; pgm \mid \textbf{inst}\ \mathcal{I}; pgm \mid e$
definition	$\mathcal{D}$	::=	$k = e$
class	C	::=	$\mathsf{TC}\ a\ \textbf{where}\ \{k : \rho\}$
instance	$\mathcal{I}$	::=	$\overline{C_i}^{\,i} \Rightarrow \mathsf{TC}\ \tau\ \textbf{where}\ \{k = e\}$
expression	e	::=	$i \mid k \mid x \mid \lambda x : \tau.e \mid e_1\ e_2 \mid [\![e]\!] \mid \$e$
monotype	τ	::=	$a \mid \mathsf{Int} \mid \tau_1 \to \tau_2 \mid \mathsf{Code}\ \tau$
qualified type	ρ	::=	$C \Rightarrow \rho \mid \tau$
polymorphic type	σ	::=	$\forall a.\sigma \mid \rho$
constraint	C	::=	$\mathsf{TC}\ \tau \mid \mathsf{CodeC}\ C$
program context	Θ	::=	$\bullet \mid \Theta, k : \sigma \mid \Theta, \forall \overline{a_i}^{\,i}.\overline{C_j}^{\,j} \Rightarrow C$
context	Γ	::=	$\bullet \mid \Gamma, x : (\tau, n) \mid \Gamma, a \mid \Gamma, (C, n)$

$\boxed{\Theta; \Gamma \vdash^n e : \sigma \rightsquigarrow e \mid \phi}$ *(Typing expressions)*

S-LIT
$$\frac{}{\Theta; \Gamma \vdash^n i : \mathsf{Int} \rightsquigarrow i \mid \bullet}$$

S-KVAR
$$\frac{k : \sigma \in \Theta}{\Theta; \Gamma \vdash^n k : \sigma \rightsquigarrow k \mid \bullet}$$

S-VAR
$$\frac{x : (\tau, n) \in \Gamma}{\Theta; \Gamma \vdash^n x : \tau \rightsquigarrow x \mid \bullet}$$

S-ABS
$$\frac{\Theta; \Gamma, x : (\tau_1, n) \vdash^n e : \tau_2 \rightsquigarrow e \mid \phi_1 \qquad \Gamma \vdash \tau_1 \rightsquigarrow \tau_1' \qquad \phi_1 +\!\!+ x : (\tau_1', n) \rightsquigarrow \phi_2}{\Theta; \Gamma \vdash^n \lambda x : \tau_1.e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \tau_1'.e \mid \phi_2}$$

S-APP
$$\frac{\Theta; \Gamma \vdash^n e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1 \mid \phi_1 \qquad \Theta; \Gamma \vdash^n e_2 : \tau_1 \rightsquigarrow e_2 \mid \phi_2}{\Theta; \Gamma \vdash^n e_1\ e_2 : \tau_2 \rightsquigarrow e_1\ e_2 \mid \phi_1, \phi_2}$$

S-TABS
$$\frac{\Theta; \Gamma, a \vdash^n e : \sigma \rightsquigarrow e \mid \phi_1 \qquad \phi_1 +\!\!+ a \rightsquigarrow \phi_2}{\Theta; \Gamma \vdash^n e : \forall a.\sigma \rightsquigarrow \Lambda a.e \mid \phi_2}$$

S-TAPP
$$\frac{\Theta; \Gamma \vdash^n e : \forall a.\sigma \rightsquigarrow e \mid \phi \qquad \Gamma \vdash \tau \rightsquigarrow \tau'}{\Theta; \Gamma \vdash^n e : \sigma[a \mapsto \tau] \rightsquigarrow e\ \tau' \mid \phi}$$

S-CABS
$$\frac{\Theta; \Gamma, ev : (C, n) \vdash^n e : \rho \rightsquigarrow e \mid \phi_1 \qquad \Gamma \vdash C \rightsquigarrow \tau \qquad \phi_1 +\!\!+ ev : (\tau, n) \rightsquigarrow \phi_2 \qquad \mathsf{fresh}\ ev}{\Theta; \Gamma \vdash^n e : C \Rightarrow \rho \rightsquigarrow \lambda ev : \tau.e \mid \phi_2}$$

S-CAPP
$$\frac{\Theta; \Gamma \vdash^n e : C \Rightarrow \rho \rightsquigarrow e_1 \mid \phi_1 \qquad \Theta; \Gamma \models^n C \rightsquigarrow e_2 \mid \phi_2}{\Theta; \Gamma \vdash^n e : \rho \rightsquigarrow e_1\ e_2 \mid \phi_1, \phi_2}$$

S-QUOTE
$$\frac{\Theta; \Gamma \vdash^{n+1} e : \tau \rightsquigarrow e \mid \phi}{\Theta; \Gamma \vdash^n [\![e]\!] : \mathsf{Code}\ \tau \rightsquigarrow [\![e]\!]_{\phi.n} \mid \lfloor \phi \rfloor^n}$$

S-SPLICE
$$\frac{\Theta; \Gamma \vdash^{n-1} e : \mathsf{Code}\ \tau \rightsquigarrow e \mid \phi \qquad \Gamma \vdash \tau \rightsquigarrow \tau' \qquad \mathsf{fresh}\ s}{\Theta; \Gamma \vdash^n \$e : \tau \rightsquigarrow s \mid \phi, (\bullet \vdash^{n-1} s : \tau' = e)}$$

Fig. 1. Syntax and typing rules of $\lambda^{[\![\Rightarrow]\!]}$

be used at any level (rules S-LIT and S-KVAR), as they can be persisted. Importantly, rule S-VAR says that if a variable x is introduced at level n, then it is well-typed at level n. Rules S-CABS and S-CAPP handle generalization and instantiation of type class constraints. If an expression e can be type-checked under a local type class assumption C, then e has a qualified type $C \Rightarrow \rho$. Otherwise, if a constraint C can be resolved (§3.3), then an expression of type $C \Rightarrow \rho$ can be typed ρ.

$\boxed{\Gamma \vdash \sigma \rightsquigarrow \tau}$ *(Well-formed types)*

S-K-TVAR
$$\frac{a \in \Gamma}{\Gamma \vdash a \rightsquigarrow a}$$

S-K-ARROW
$$\frac{\Gamma \vdash \tau_1 \rightsquigarrow \tau_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \tau_2}{\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow \tau_1 \to \tau_2}$$

S-K-CARROW
$$\frac{\Gamma \vdash C \rightsquigarrow \tau_1 \qquad \Gamma \vdash \rho \rightsquigarrow \tau_2}{\Gamma \vdash C \Rightarrow \rho \rightsquigarrow \tau_1 \to \tau_2}$$

S-K-FORALL
$$\frac{\Gamma, a \vdash \sigma \rightsquigarrow \tau}{\Gamma \vdash \forall a.\sigma \rightsquigarrow \forall a.\tau}$$

S-K-CODE
$$\frac{\Gamma, a \vdash \tau \rightsquigarrow \tau'}{\Gamma \vdash \mathsf{Code}\ \tau \rightsquigarrow \mathsf{Code}\ \tau'}$$

$\boxed{\Gamma \vdash C \rightsquigarrow \tau}$ *(Well-formed constraints)*

S-K-TC
$$\frac{\mathsf{TC}\ a\ \mathbf{where}\ \{k : \rho\} \qquad \Gamma \vdash \rho[a \mapsto \tau] \rightsquigarrow \tau'}{\Gamma \vdash \mathsf{TC}\ \tau \rightsquigarrow \tau'}$$

S-K-CODEC
$$\frac{\Gamma \vdash C \rightsquigarrow \tau}{\Gamma \vdash \mathsf{CodeC}\ C \rightsquigarrow \mathsf{Code}\ \tau}$$

$\boxed{\Theta; \Gamma \models^n C \rightsquigarrow e \mid \phi}$ *(Constraint Resolution)*

S-SOLVE-GLOBAL
$$\frac{ev : \forall a.\overline{C_i}^{\,i} \Rightarrow C \in \Theta \qquad \Gamma \vdash \tau \rightsquigarrow \tau' \qquad \overline{\Theta; \Gamma \models^n C_i[a \mapsto \tau] \rightsquigarrow e_i \mid \phi_i}^{\,i}}{\Theta; \Gamma \models^n C[a \mapsto \tau] \rightsquigarrow ev\ \tau'\ \overline{e_i}^{\,i} \mid \overline{\phi_i}^{\,i}}$$

S-SOLVE-LOCAL
$$\frac{ev : (C, n) \in \Gamma}{\Theta; \Gamma \models^n C \rightsquigarrow ev \mid \bullet}$$

S-SOLVE-INCR
$$\frac{\Theta; \Gamma \models^{n-1} \mathsf{CodeC}\ C \rightsquigarrow e \mid \phi \qquad \Gamma \vdash C \rightsquigarrow \tau \qquad \mathsf{fresh}\ s}{\Theta; \Gamma \models^n C \rightsquigarrow s \mid \phi, (\bullet \vdash^{n-1} s : \tau = e)}$$

S-SOLVE-DECR
$$\frac{\Theta; \Gamma \models^{n+1} C \rightsquigarrow e \mid \phi}{\Theta; \Gamma \models^n \mathsf{CodeC}\ C \rightsquigarrow [\![e]\!]_{\phi.n} \mid \lfloor \phi \rfloor^n}$$

Fig. 2. Well-formed types, well-formed constraints and constraint resolution in $\lambda^{[\![\Rightarrow]\!]}$

Rules S-QUOTE and S-SPLICE type-check staging annotations. In particular, rule S-QUOTE increases the level by one and gives $[\![e]\!]$ type Code τ when e has type τ, while rule S-SPLICE decreases the level by one and gives e type τ when $\$e$ has type Code τ.

Well-formed types and constraints. Typing rules (e.g., rule S-ABS) refer to well-formed rules for types and for constraints as given in Figure 2. The type well-formedness judgment $\Gamma \vdash \sigma$ simply checks that all type variables are well-scoped. The constraint well-formedness constraint $\Gamma \vdash C$ checks that the class method type is well-formed after substituting the variable a with τ.

3.3 Constraint Resolution

The typing rule (rule S-CAPP) also makes use of constraint resolution, whose rules are given at the bottom of Figure 2. The judgment $\Theta; \Gamma \models^n C$ reads that under the program theory Θ, the context Γ, and the current level C, the type class constraint C can be resolved. The definition of constraint resolution in $\lambda^{[\![\Rightarrow]\!]}$ has two key novelties: (1) *level-indexing*, which allows us to guarantee well-stagedness of constraints; (2) resolution of staged type class constraints.

Rule S-SOLVE-GLOBAL resolves a type class constraint using an instance definition. If Θ contains the instance definition $\forall a.\overline{C_i}^{\,i} \Rightarrow C$, we can resolve $C[a \mapsto \tau]$ by resolving $\overline{C_i[a \mapsto \tau]}^{\,i}$. Rule S-SOLVE-LOCAL resolves a constraint using the local type class axiom.

$\boxed{\Theta \vdash pgm : \sigma}$ *(Typing programs)*

S-PGM-DEF
$$\frac{\Theta_1 \vdash \mathcal{D} \dashv \Theta_2 \qquad \Theta_2 \vdash pgm : \sigma}{\Theta_1 \vdash \mathbf{def}\ \mathcal{D}; pgm : \sigma}$$

S-PGM-CLS
$$\frac{\Theta_1 \vdash C \dashv \Theta_2 \qquad \Theta_2 \vdash pgm : \sigma}{\Theta_1 \vdash \mathbf{class}\ C; pgm : \sigma}$$

S-PGM-INST
$$\frac{\Theta_1 \vdash \mathcal{I} \dashv \Theta_2 \qquad \Theta_2 \vdash pgm : \sigma}{\Theta_1 \vdash \mathbf{inst}\ \mathcal{I}; pgm : \sigma}$$

S-PGM-EXPR
$$\frac{\Theta; \bullet \vdash^0 e : \sigma \rightsquigarrow e \mid \phi \qquad \bullet \vdash \sigma \rightsquigarrow \tau \qquad e : \tau \vdash^{-1} \phi \rightsquigarrow \rho gm}{\Theta \vdash e : \sigma \rightsquigarrow \rho gm}$$

$\boxed{\Theta_1 \vdash \mathcal{D} \dashv \Theta_2}$ *(Typing definitions)* $\boxed{\Theta_1 \vdash C \dashv \Theta_2}$ *(Typing class declarations)*

S-DEF
$$\frac{\Theta; \bullet \vdash^0 e : \sigma}{\Theta \vdash k = e \dashv \Theta, k : \sigma}$$

S-CLS
$$\frac{a \vdash \rho}{\Theta \vdash \mathsf{TC}\ a\ \mathbf{where}\ \{k : \rho\} \dashv \Theta, k : \forall a.\mathsf{TC}\ a \Rightarrow \rho}$$

$\boxed{\Theta_1 \vdash \mathcal{I} \dashv \Theta_2}$ *(Typing class instances)*

S-INST
$$\frac{\mathsf{TC}\ a\ \mathbf{where}\ \{k : \rho\} \qquad \overline{\overline{b_j}^{\,j} \vdash C_i}^{\,i} \qquad \Theta; \overline{\overline{b_j}^{\,j}, (C_i, 0)}^{\,i} \vdash^0 e : \rho[a \mapsto \tau] \qquad \overline{b_j}^{\,j} = \mathsf{ftv}(\tau)}{\Theta \vdash \overline{C_i}^{\,i} \Rightarrow \mathsf{TC}\ \tau\ \mathbf{where}\ \{k = e\} \dashv \Theta, \forall \overline{b_j}^{\,j}.\overline{C_i}^{\,i} \Rightarrow \mathsf{TC}\ \tau}$$

Fig. 3. Program typing in $\lambda^{[\![\Rightarrow]\!]}$

Rules S-SOLVE-DECR and S-SOLVE-INCR are specific to our system. In particular, rule S-INCR says that a staged type class constraint CodeC C at level $n-1$ can be used to resolve C at level n, which is essentially what enables us to have constraint inside quotations. Similarly, rule S-DECR says that a normal type class constraint C at level $n+1$ can be used to resolve CodeC C at level n. We can thus use these two rules to convert back and forth between CodeC C and C.

Example 3.1 ($\lambda^{[\![\Rightarrow]\!]}$ typing). Let us illustrate the typing rules and the constraint resolution rules by revisiting the example $[\![\ show\ ]\!]$ (Example S1). Below we give its typing derivation. For this example we assume the primitive type String, and the program environment Θ to contain the type of *show*.

$\Theta = show : \forall a.\mathsf{Show}\ a \Rightarrow a \to \mathsf{String}$

$\Gamma = a, (\mathsf{CodeC}\ (\mathsf{Show}\ a), 0)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{show : \forall a.\mathsf{Show}\ a \Rightarrow a \to \mathsf{String} \in \Theta}{\Theta; \Gamma \vdash^1 show : \forall a.\mathsf{Show}\ a \Rightarrow a \to \mathsf{String}}\ \text{S-KVAR}}{\Theta; \Gamma \vdash^1 show : \mathsf{Show}\ a \Rightarrow a \to \mathsf{String}}\ \text{S-TAPP} \qquad \dfrac{(\mathsf{CodeC}\ (\mathsf{Show}\ a), 0) \in \Gamma}{\Theta; \Gamma \Vdash^0 \mathsf{CodeC}\ (\mathsf{Show}\ a)}\ \text{S-SOLVE-LOCAL}}{\Theta; \Gamma \Vdash \mathsf{Show}\ a}\ \text{S-SOLVE-INCR}}{\Theta; \Gamma \vdash^1 show : (a \to \mathsf{String})}\ \text{S-CAPP}}{\Theta; \Gamma \vdash^0 [\![show]\!] : \mathsf{Code}\ (a \to \mathsf{String})}\ \text{S-QUOTE}}{\Theta; a \vdash^0 [\![show]\!] : \mathsf{CodeC}\ (\mathsf{Show}\ a) \Rightarrow \mathsf{Code}\ (a \to \mathsf{String})}\ \text{S-CABS}}{\Theta; \bullet \vdash^0 [\![show]\!] : \forall a.\mathsf{CodeC}\ (\mathsf{Show}\ a) \Rightarrow \mathsf{Code}\ (a \to \mathsf{String})}\ \text{S-TABS}$$

Let us go through the derivation bottom-up. First, by applying rules S-TABS and S-CABS, we introduce the type variable a and the staged type class constraint CodeC (Show a) at level 0 into the context. Then by rule S-QUOTE, our goal becomes $\Theta; \Gamma \vdash^1 show : (a \to \mathsf{String})$ at level 1. At this point, rule S-KVAR allows us to use *show* from Θ at level 1, but we need to further apply rule S-TAPP and S-CAPP, and the latter requires us to prove Show a at level 1. To this end, rule S-SOLVE-LOCAL first gets CodeC Show a at level 0, and rule S-SOLVE-INCR then converts it into Show a at level 1.

3.4 Program Typing

As we have seen from the syntax, a program is a sequence of top-level definitions, class and instance declarations followed by an expression. Figure 3 presents the typing rules for programs. The judgment $\Theta \vdash pgm : \sigma$ reads that under the program theory Θ, the source program pgm has type σ. Most rules are standard. Top-level definitions (rule S-PGM-DEF) and declaration forms (rules S-PGM-CLS and S-PGM-INST) extend the program theory Θ which is used to type-check subsequent definitions. Rule S-PGM-EXPR makes it clear that the top-level of the program is level 0 and that the expression is checked in an empty local environment.

Rules S-DEF, S-CLS, and S-INST type-check top-level definitions, class and instance declarations, respectively. Rule S-DEF extends the list of top-level definitions available at all stages. Rule S-CLS extends the program theory with the qualified class method. Rule S-INST checks that the class method is of the type specified in the class definition.

4 $F^{[]}$: MULTI-STAGE CORE CALCULUS WITH SPLICE ENVIRONMENTS

We describe an explicitly typed core language $F^{[]}$, which extends System F with quotations, *splice environments* and *top-level splice definitions*. $F^{[]}$ does not contain splices themselves as they are modeled using the splice environments, which are attached to quotations, and top-level splice definitions. As such, quotations can be considered opaque until spliced, and $F^{[]}$ serves as a suitable compilation target for multi-staging programming.

4.1 Syntax

The syntax for $F^{[]}$ is presented at the top of Figure 4. To reduce notational clutter, we reuse notation from $\lambda^{[\Rightarrow]}$ for expressions and types, making it clear from the context which calculus we refer to.

A program (ρgm) is a sequence of top-level definitions ($\mathcal{D}$) and top-level splice definitions ($\mathcal{S}$) followed by an expression (e). Top-level definitions $k : \tau = e$ are the same as for $\lambda^{[\Rightarrow]}$, except that, since $F^{[]}$ is explicitly typed, k is associated with its type τ. There is no syntax for type classes or instances, which will be represented using top-level definitions after dictionary-passing elaborations. Top-level splice definitions $\Delta \vdash^n s : \tau = e$ are used to support compile-time evaluation, where the *splice variable* s captures the local type environment Δ, the level n, the type after splicing τ, and the expression to be spliced e. As we will see, the typing rules will ensure that that expression e has type Code τ at level n under type context Δ. The purpose of the environment Δ is to support open code representations which lose their lexical scoping when floated out from the quotation.

Expressions e include literals i, top-level variables k, splice variables s, variables x, lambdas $\lambda x : \tau.e$, applications $e_1\ e_2$, type abstractions $\Lambda a.e$ and type applications $e\ \tau$, and quotations with splice environment $[\![e]\!]_\phi$, which are quotations with an associated splice environment. The splice environment ϕ is essentially a list of splice definitions ($\Delta \vdash^n s : \tau = e$), which binds a splice variable s for each splice point within the quoted expression. A splice point is where the result of evaluating a splice will be inserted. One example we have seen from §2.7 is that the expression $[\![\ e_2\ \$(e_3)\ ]\!]$ can be represented in $F^{[]}$ as $[\![e_2\ s]\!]_{\bullet^0 s:\tau=e_3}$ which, when spliced, will insert the result of splicing e_3 in the place of the splice variable s.

The program context Θ records the type of top-level definitions $k : \tau$ and top-level splice definitions $s : (\Delta, \tau, n)$. We distinguish between two type contexts Δ and Γ, where Γ is Δ extended with types for splice variables. The syntax distinction makes it clear that splice definitions ($\mathcal{S}$) and environments (ϕ) only capture Δ, which are type contexts elaborated from the source language and so contain no splice variables.

program	ρgm	::=	$\mathbf{def}\ \mathcal{D}; \rho gm \mid \mathbf{spdef}\ \mathcal{S}; \rho gm \mid e : \tau$
definition	$\mathcal{D}$	::=	$k : \tau = e$
splice definition	$\mathcal{S}$	::=	$\Delta \vdash^n s : \tau = e$
expression	e	::=	$i \mid k \mid s \mid x \mid \lambda x : \tau.e \mid e_1\, e_2 \mid \Lambda a.e \mid e\, \tau \mid [\![e]\!]_\phi$
splice environment	ϕ	::=	$\bullet \mid \phi, \Delta \vdash^n s : \tau = e$
type	τ	::=	$a \mid \mathsf{Int} \mid \tau_1 \to \tau_2 \mid \forall a.\tau \mid \mathsf{Code}\ \tau$
program context	Θ	::=	$\bullet \mid \Theta, k : \tau \mid \Theta, s : (\Delta, \tau, n)$
context	Δ	::=	$\bullet \mid \Delta, x : (\tau, n) \mid \Delta, a$
	Γ	::=	$\bullet \mid \Gamma, x : (\tau, n) \mid \Gamma, a \mid \Gamma, s : (\Delta, \tau, n)$

$\boxed{\Theta \vdash \rho gm}$ *(Typing programs)*

C-PGM-DEF
$$\frac{\Theta_1 \vdash \mathcal{D} \dashv \Theta_2 \qquad \Theta_2 \vdash \rho gm}{\Theta_1 \vdash \mathbf{def}\ \mathcal{D}; \rho gm}$$

C-PGM-SPDEF
$$\frac{\Theta_1 \vdash \mathcal{S} \dashv \Theta_2 \qquad \Theta_2 \vdash \rho gm}{\Theta_1 \vdash \mathbf{spdef}\ \mathcal{S}; \rho gm}$$

C-PGM-EXPR
$$\frac{\Theta; \bullet \vdash^0 e : \tau}{\Theta \vdash e : \tau}$$

$\boxed{\Theta_1 \vdash \mathcal{D} \dashv \Theta_2}$ *(Typing definitions)* $\boxed{\Theta_1 \vdash \mathcal{S} \dashv \Theta_2}$ *(Typing splice definitions)*

C-DEF
$$\frac{\Theta; \bullet \vdash^0 e : \tau}{\Theta \vdash k : \tau = e \dashv \Theta, k : \tau}$$

C-SPDEF
$$\frac{\Theta; \Delta \vdash^n e : \mathsf{Code}\ \tau \qquad \Delta \dot{>} n}{\Theta \vdash \Delta \vdash^n s : \tau = e \dashv \Theta, s : (\Delta, \tau, n+1)}$$

$\boxed{\Theta; \Gamma \vdash^n e : \tau}$ *(Typing expressions)*

C-LIT
$$\frac{}{\Theta; \Gamma \vdash^n i : \mathsf{Int}}$$

C-VAR
$$\frac{x : (\tau, n) \in \Gamma}{\Theta; \Gamma \vdash^n x : \tau}$$

C-KVAR
$$\frac{k : \tau \in \Theta}{\Theta; \Gamma \vdash^n k : \tau}$$

C-SVAR
$$\frac{s : (\Delta, \tau, n) \in \Gamma \qquad \Delta \subseteq \Gamma}{\Theta; \Gamma \vdash^n s : \tau}$$

C-TOP-SVAR
$$\frac{s : (\Delta, \tau, n) \in \Theta \qquad \Delta \subseteq \Gamma}{\Theta; \Gamma \vdash^n s : \tau}$$

C-ABS
$$\frac{\Theta; \Gamma, x : (\tau_1, n) \vdash^n e : \tau_2}{\Theta; \Gamma \vdash^n \lambda x : \tau_1.e : \tau_1 \to \tau_2}$$

C-APP
$$\frac{\Theta; \Gamma \vdash^n e_1 : \tau_1 \to \tau_2 \qquad \Theta; \Gamma \vdash^n e_2 : \tau_1}{\Theta; \Gamma \vdash^n e_1\, e_2 : \tau_2}$$

C-TABS
$$\frac{\Theta; \Gamma, a \vdash^n e : \tau}{\Theta; \Gamma \vdash^n \Lambda a.e : \forall a.\tau}$$

C-TAPP
$$\frac{\Theta; \Gamma \vdash^n e : \forall a.\tau_2}{\Theta; \Gamma \vdash^n e\, \tau_1 : \tau_2[a \mapsto \tau_1]}$$

C-QUOTE
$$\frac{\Theta; \Gamma \vdash^n \phi \qquad \Theta; \Gamma, \phi^\Gamma \vdash^{n+1} e : \tau}{\Theta; \Gamma \vdash^n [\![e]\!]_\phi : \mathsf{Code}\ \tau}$$

ϕ^Γ converts ϕ into a context.

$$\bullet^\Gamma = \bullet$$
$$(\phi, \Delta \vdash^n s : \tau = e)^\Gamma = \phi^\Gamma, s : (\Delta, \tau, n+1)$$

$\boxed{\Theta; \Gamma \vdash^n \phi \triangleq \Theta; \Gamma \vdash \phi \wedge \phi \doteq n}$ $\boxed{\Theta; \Gamma \vdash \phi}$ *(Typing splice environments)*

C-S-EMPTY
$$\frac{}{\Theta; \Gamma \vdash \bullet}$$

C-S-CONS
$$\frac{\Theta; \Gamma \vdash \phi \qquad \Theta; \Gamma, \Delta \vdash^n e : \mathsf{Code}\ \tau \qquad \Delta \dot{>} n}{\Theta; \Gamma \vdash \phi, (\Delta \vdash^n s : \tau = e)}$$

Fig. 4. Syntax and typing in $F^{[\![]\!]}$

4.2 Typing Rules

Figure 4 presents the typing rules for $F^{[\![]\!]}$. The judgment $\Theta \vdash \rho gm$ type-checks a core program. As before, top-level definitions (rule C-PGM-DEF) and top-level splice definitions (rule C-PGM-SPDEF)

extend the program theory Θ which is used to type-check subsequent definitions. Rule C-PGM-EXPR type-checks the expression.

Rule C-SPDEF checks top-level splice definitions $\Delta \vdash^n s : \tau = e$ by checking that e has type Code τ at level n under the current program context Θ and the context Δ. Notice that the program context Θ is extended with $s : (\Delta, \tau, n+1)$. The level of s is $n+1$ as it represents the *spliced* expression. In the example of $[\![e_2\ s]\!]_{\bullet^0, s:\tau=e_3}$ which expresses $[\![\ e_2\ \$(e_3)\ ]\!]$, the splice variable s stands for $\$(e_3)$. The precondition $\Delta \dot{>} n$ ensures that all variables in Δ have levels greater than n (§4.4.1). We use *dotted binary operators* (e.g., $\dot{>}$, $\doteq$ etc) to indicate level comparison.

The expression typing rules for the core expressions are for the most part the same as those in the source language. One observation is that since the language does not contain splicings, the level during typing can only increase (when typing quotations in rule C-QUOTE) but never decrease.

Rules C-SVAR and C-TOP-SVAR retrieve the type of splice variables from the context. Note that, as with expression variables, splice variables must be used at the level where they are introduced. Moreover, the local type context Δ captured by s must be a subset of the current type context Γ so that all free variables in e remain well-typed after substituting s with e. Γ may contain more variables, including splice variables that are not in Δ.

Rule C-QUOTE, which type-checks quotations with splice environments, is of particular interest. First, it checks that a splice environment is well-typed by the judgment $\Theta; \Gamma \Vdash^n \phi$, which is based on the judgment $\Theta; \Gamma \vdash \phi$ but in addition requires ϕ to contain only splice variables of level n (§4.4.1). An empty splice environment is always well-typed (rule C-S-EMPTY). Otherwise the splice environment is well-typed if each of definition is well-typed (rule C-S-CONS), where the context Γ is extended with the local type context Δ to type-check e.

After type-checking ϕ, rule C-QUOTE converts the splice environment ϕ into a list of splice variables ϕ^Γ. The definition of ϕ^Γ is straightforward and is given in the same figure. Then, rule C-QUOTE adds all those splice variables ϕ^Γ into the context Γ, as they may be used inside e. One way to think about splice environments is that they attach splice variable *bindings* to the quotation whose body is e. And thus their concrete names do not matter and we can consider quotations equivalent up to alpha-renaming, e.g., $[\![s]\!]_{\Delta \Vdash^n s:\tau=e_1} =_\alpha [\![s']\!]_{\Delta \Vdash^n s':\tau=e_1}$. Finally, the rule type-checks e at level $n+1$, and concludes with the type Code τ.

4.3 Dynamic Semantics

Figure 5 presents the definition of values and dynamic semantics in $F^{[\![]\!]}$. Note that evaluation is not level-indexed, as splice environments make the evaluation order of the core calculus evident.

Values v include literals i, lambdas $\lambda x : \tau.e$, type abstractions $\Lambda a.e$, and quotations with splice environments $[\![e]\!]_{\phi_v}$. Notably, quotation values ($[\![e]\!]_{\phi_v}$) can quote arbitrary expressions (e), but require the splice environment to be a value (ϕ_v). A splice environment value ϕ_v simply requires all bindings to be values (i.e. $\Delta \vdash^n s : \tau = v$). As we will see from the dynamic semantics shortly, this avoids the need to look inside quotations, as the splice environment corresponds exactly to the splices inside quotations that need to be evaluated.

The program evaluation judgment ($\rho g m_1 \longrightarrow \rho g m_2$) evaluates declarations in turn from top to bottom. Top-level definitions are evaluated (rule CE-PGM-DEF) to values and substituted into the rest of the program (rule CE-PGM-DBETA). Similarly, rule CE-PGM-SPDEF evaluates a top-level splice definition to a value of the form $[\![e]\!]_\phi$. We must then insert splices back into the program, which is done in rule CE-PGM-SPBETA by substituting s with $[\phi_v]e$. The notation $[\phi_v]e$, defined at the top of the figure, further inserts splices in ϕ_v back into the expression e. To understand the process, let us first consider the case when ϕ_v is empty, giving us $[\bullet]e = e$, and suppose $n = -1$ then we have:

$$\textbf{spdef}\ \Delta \vdash^{-1} s : \tau = [\![e]\!]_\bullet; \rho g m \longrightarrow \rho g m[s \mapsto e]$$

$$
\begin{array}{lrcl}
\text{value} & v & ::= & i \mid \lambda x : \tau.e \mid \Lambda a.e \mid [\![e]\!]_{\phi_v} \\
\text{splice environment value} & \phi_v & ::= & \bullet \mid \phi_v, \Delta \vdash^n s : \tau = v
\end{array}
$$

$[\phi_v]e$ inserts splices in ϕ_v back into e.

$$
\begin{array}{lcl}
[\bullet]e & = & e \\
[\phi_v, \Delta \vdash^n s : \tau = [\![e']\!]_{\phi_{v'}}]e & = & [\phi_v](e[s \mapsto [\phi_v{}']e'])
\end{array}
$$

$\boxed{\rho gm_1 \longrightarrow \rho gm_2}$ *(Program reduction)*

CE-PGM-DEF
$$\frac{\mathcal{D} \longrightarrow \mathcal{D}'}{\textbf{def}\ \mathcal{D}; \rho gm \longrightarrow \textbf{def}\ \mathcal{D}'; \rho gm}$$

CE-PGM-DBETA
$$\frac{}{\textbf{def}\ k : \tau = v; \rho gm \longrightarrow \rho gm[k \mapsto v]}$$

CE-PGM-EXPR
$$\frac{e \longrightarrow e'}{e : \tau \longrightarrow e' : \tau}$$

CE-PGM-SPDEF
$$\frac{\mathcal{S} \longrightarrow \mathcal{S}'}{\textbf{spdef}\ \mathcal{S}; \rho gm \longrightarrow \textbf{spdef}\ \mathcal{S}'; \rho gm}$$

CE-PGM-SPBETA
$$\frac{}{\textbf{spdef}\ \Delta \vdash^n s : \tau = [\![e]\!]_{\phi_v}; \rho gm \longrightarrow \rho gm[s \mapsto ([\phi_v]e)]}$$

$\boxed{\mathcal{D}_1 \longrightarrow \mathcal{D}_2}$ *(Definition reduction)* $\boxed{\mathcal{S}_1 \longrightarrow \mathcal{S}_2}$ *(Splice definition reduction)*

CE-DEF
$$\frac{e \longrightarrow e'}{k : \tau = e \longrightarrow k : \tau = e'}$$

CE-SPDEF
$$\frac{e \longrightarrow e'}{\Delta \vdash^n s : \tau = e \longrightarrow \Delta \vdash^n s : \tau = e'}$$

$\boxed{e_1 \longrightarrow e_2}$ *(Reduction)*

CE-BETA
$$\frac{}{(\lambda x : \tau.e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]}$$

CE-TBETA
$$\frac{}{(\Lambda a.e)\ \tau \longrightarrow e[a \mapsto \tau]}$$

CE-APP
$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

CE-TAPP
$$\frac{e \longrightarrow e'}{e\ \tau \longrightarrow e'\ \tau}$$

CE-QUOTE
$$\frac{\phi \longrightarrow \phi'}{[\![e]\!]_\phi \longrightarrow [\![e]\!]_{\phi'}}$$

$\boxed{\phi_1 \longrightarrow \phi_2}$ *(Splice environment reduction)*

CE-S-HEAD
$$\frac{\phi \longrightarrow \phi'}{\phi, \Delta \vdash^n s : \tau = e \longrightarrow \phi', \Delta \vdash^n s : \tau = e}$$

CE-S-TAIL
$$\frac{e \longrightarrow e'}{\phi_v, \Delta \vdash^n s : \tau = e \longrightarrow \phi_v, \Delta \vdash^n s : \tau = e'}$$

Fig. 5. Values and dynamic semantics in $F^{[\![]\!]}$

Essentially, $\Delta \vdash^n s : \tau = [\![e]\!]_\bullet$ corresponds to the expression $\$[\![e]\!]$ in the source level, whose splicing result is bound to s. The position of s inside ρgm indicates where the source program $\$[\![e]\!]$ was originally found, and by substituting s with e we successfully insert the splicing result back into that position. Rule CE-PGM-SPBETA deals with the more general case where ϕ_v can be non-empty, which corresponds to *nested* splices, i.e., the source expression e (as in $\$[\![e]\!]$) may itself contain more splices, and those splices (of the corresponding level, in this case -1) are reflected as the splice environment ϕ_v associated to $[\![e]\!]_{\phi_v}$. In this case, we need to first insert those splice definitions back into the expression, i.e., as $[\phi_v]e$, and then we conclude by substituting s with $[\phi_v]e$.

After we evaluate all definitions and splice definitions, we can then start evaluating the expression (rule CE-PGM-EXPR). Expression reductions ($e_1 \longrightarrow e_2$) are mostly standard. Rule CE-BETA uses call-by-name, though the exact choice of the evaluation strategy does not matter. Of particular interest is rule CE-QUOTE, which says that to evaluate $[\![e]\!]_\phi$, we leave e as is, and all we need to do is to evaluate ϕ, which simply evaluates all expressions it binds (rules CE-S-HEAD and CE-S-TAIL).

Note that there is no reduction rule which reduces inside a quotation. Now the benefits of splice environments become clear: we can treat a quoted expression (the e part in $[\![e]\!]_\phi$) opaquely, giving the implementation freedom about its concrete form.

4.4 Well-stagedness and Type Soundness

In this section, we discuss the metatheory of $F^{[\![]\!]}$. Before we present the type soundness result, we first discuss well-stagedness of splice environments.

4.4.1 Well-staged Splice Definitions and Environments. Our typing rules are designed carefully to allow only well-staged programs. As splice definitions and environments are novel in this calculus, great care needs to be taken to guarantee their well-stagedness. To this end, the typing rules have imposed the following restrictions on levels of splice definitions and environments:

(1) A splice definition $\Delta \vdash^n s : \tau = e$ requires $\Delta \dot{>} n$ as in rule C-SPDEF (similarly, rule C-S-CONS). That is, all splice variables in the local type context captured by a splice variable must have a level greater than that of the expression captured by the splice variable.
(2) A well-staged quotation $\Theta; \Delta \vdash^n [\![e]\!]_\phi$ requires $\Theta; \Gamma \vdash^n \phi$, as in rule C-QUOTE, which implies $\phi \doteq n,$. That is, all splice variables that bind level n are introduced at level n.[4]

Example 4.1 (Counterexamples to well-staged splices). The following examples are rejected.

(a) $\bullet; \bullet \vdash^0 [\![e]\!]_{x:(\mathsf{Code\ Int},0)\vdash^\rho s:\mathsf{Int}=x}$: Code τ breaks (1) as $x : (\mathsf{Code\ Int}, 0) \not> 0$
(b) $\bullet; \bullet \vdash^0 [\![[\![e]\!]_{\bullet\vdash^\rho s:\mathsf{Int}=(\lambda y:\mathsf{Code\ Int}.\, y)\ ([\![2]\!]_\bullet)}]\!]_\bullet$: Code (Code τ) breaks (2) as $\bullet \vdash^0 s : \mathsf{Int} \not\equiv 1$

Essentially, the first restriction applies the *level restriction* of variables described in §2.1 to splice definition and environments; and the second lifts the level restriction to splice variables. In particular, consider the counterexample (a). What happens is that in the splice environment x is used at level 0, but inside e we can never introduce x at level 0 (recall that during typing the level monotonically increases)! So such an example is rejected because x is not well-staged.[5]

The level restriction to splice variables requires that a splice variable that binds level n is introduced at level n. The splice variable level restriction ensures that splice variables are evaluated at the right stage. Consider counterexample (b). If we evaluate the program at level 0, then because the splice environment is a value and we do not inspect inside the quotations, we will conclude that it is a value. But note that s is bound at level 0, which means the expression $(\lambda y : \mathsf{Code\ Int}.y)\ ([\![2]\!]_\bullet)$ is at level 0 and so should get reduced when the expression is evaluated at level 0! We thus reject this example as s is not well-staged.

4.4.2 Type Soundness. With well-staged splice definitions and environments, we can now prove that $F^{[\![]\!]}$ enjoys type soundness, by proving type preservation and progress.

First, we show that any reduction preserves the type information. For space reasons, we only present the theorem for expressions and programs, but the theorem holds for all other forms.

Theorem 4.2 (Progress). (1) *If* $\bullet; \bullet \vdash^n e : \tau$, *then either* e *is a value, or* $e \longrightarrow e'$ *for some* e'.
(2) *If* $\bullet \vdash \rho gm$, *then either* ρgm *is* $v : \tau$, *or* $\rho gm \longrightarrow \rho gm'$ *for some* $\rho gm'$.

[4] An alternative is to represent a splice environment entry as $\Delta \vdash s : \tau = e$ (i.e. without levels), and then rule C-QUOTE, just like rule C-ABS, could directly take the current level from the typing judgment (which also means ϕ^Γ would need to take a level as input). However, that representation does not work for global splice variables (i.e. in rule C-SPDEF where typing is not level-indexed). Moreover, the representation of ϕ is also used during elaboration, where it is important to track the levels. Therefore, we prefer to have a consistent representation and preserve the level information in the core.

[5] It may seem like we can introduce x outside of the quotation, making x well-staged. However, if x is introduced outside of the quotation (and thus the splice environment), then it should *not* be captured by the splice variable, as it is in the scope of the splice environment (i.e. is not *local*). For example, the well-typed source program $\lambda x : \mathsf{Code\ Int}.[\![\$x]\!]$ elaborates to $\lambda x : \mathsf{Code\ Int}.[\![s]\!]_{\bullet\vdash^\rho s:\mathsf{Int}=x}$, while the source program $[\![\lambda x : \mathsf{Int}.\$[\![x]\!]]\!]$ elaborates to $[\![\lambda x : \mathsf{Code\ Int}.s]\!]_{x:(\mathsf{Int},1)\vdash^\rho s:\mathsf{Int}=[\![x]\!]_\bullet}$.

$\boxed{\phi_1 +\!\!+ \Delta \rightsquigarrow \phi_2}$ *(Injection)*

$$\frac{}{\bullet +\!\!+ \Delta \rightsquigarrow \bullet}\ \textsc{s-inj-empty} \qquad \frac{\phi_1 +\!\!+ \Delta_2 \rightsquigarrow \phi_2}{\phi_1, \Delta_1 \vdash^n s : \tau = e +\!\!+ \Delta_2 \rightsquigarrow \phi_2, (\Delta_2, \Delta_1 \vdash^n s : \tau = e)}\ \textsc{s-inj-cons}$$

$\boxed{\rho gm_1 \vdash^n \phi \rightsquigarrow \rho gm_2}$ *(Collapse)*

$$\frac{}{\rho gm \vdash^n \bullet \rightsquigarrow \rho gm}\ \textsc{s-clap-empty} \qquad \frac{\textbf{spdef}\ \phi.n; \rho gm_1 \vdash^{n-1} \lfloor \phi \rfloor^n \rightsquigarrow \rho gm_2}{\rho gm_1 \vdash^n \phi \rightsquigarrow \rho gm_2}\ \textsc{s-clap-rec}$$

Fig. 6. Auxiliary definitions used in elaboration: injection used in Figure 1, and collapse used in Figure 3

Now we show that well-typed programs cannot go wrong, by proving that a well-typed expression (and definition / program respectively) is either a value, or can take a step.

Theorem 4.3 (Type Preservation). (1) *If* $\Theta; \Delta \vdash^n e : \tau$, *and* $e \longrightarrow e'$, *then* $\Theta; \Delta \vdash^n e' : \tau$.
(2) *If* $\Theta \vdash \rho gm$, *and* $\rho gm \longrightarrow \rho gm'$, *then* $\Theta \vdash \rho gm'$.

5 ELABORATION FROM $\lambda^{[\![\Rightarrow]\!]}$ TO $F^{[\![]\!]}$

In this section we describe the process of type-directed elaboration from the source language $\lambda^{[\![\Rightarrow]\!]}$ into the core language $F^{[\![]\!]}$. There are three key aspects of the elaboration procedure :

(1) Splices are removed in favour of a splice environment. The elaboration process returns a splice environment which is attached to the quotation form (§5.1).
(2) Type class constraints are converted to explicit dictionary passing. We describe how to understand staged type class constraints *CodeC C* in terms of quotation (§5.2).
(3) Splices at non-positive levels that are not attached to a corresponding quotation are elaborated to top-level splice definitions, which are put before the rest of the program (§5.3).

5.1 Elaborating Expressions with Splice Environments

The elaboration of expressions appears in gray with the source typing rules in Figure 1. The judgment $\Theta; \Gamma \vdash^n e : \sigma \rightsquigarrow e \mid \phi$ states that, under the program context Θ and the context Γ, the source expression e at level n with type σ is elaborated into a core expression e whilst producing the splice environment ϕ. As we will see, since splices at level n create splice variables at level $n-1$, and quotations at level n capture all inner splice variables at level n, we maintain the invariant on the judgment that $\phi \stackrel{.}{<} n$ (§5.4.1).

At a high level, all splice variables are initially added to the splice environment when elaborating splices (rule S-SPLICE), and then propagated through the rules, until captured by quotations (rule S-QUOTE); uncaptured splice variables are discussed in §5.3. Let us first take a look at rule S-SPLICE. To elaborate a source splice $\$e$, rule S-SPLICE generates a fresh splice variable s which is returned as the elaboration result. It then extends the splice environment ϕ with s that binds an empty local context (as every variable is still in the scope of the splice at this moment), the level of the expression $n-1$, the core type τ', and the core expression e. This way we effectively insert s as a splice point, with the expression to be spliced bound to s in the splice environment. Splice environments are captured by quotations in rule S-QUOTE. In particular, a quotation at level n captures only the splices at level n; the notation $\phi.n$ denotes the projection of the splices contained in ϕ at level n. We then truncate ϕ by removing $\phi.n$ from it using the notation $\lfloor \phi \rfloor^n$.

Importantly, we need to ensure well-scopedness of splice environments during this process. When a splice variable gets out of a scope, e.g. in rule S-ABS, we cannot directly return ϕ_1, as ϕ_1 may refer to x and directly returning ϕ_1 would cause it to be ill-typed! To this end, whenever a splice variable gets out of a scope, it captures the scope in its local context. In other words, *a splice variable captures the local context from its introduction point up to the point where it is bound by a quotation.* This is done by the *injection* judgment $\phi_1 +\!\!+ \Delta \rightsquigarrow \phi_2$, defined at the top of Figure 6, and is used in for example rule S-ABS. Specifically, the judgment $\phi_1 +\!\!+ \Delta \rightsquigarrow \phi_2$ inserts Δ into the local context of each splice variable in ϕ_1, producing a new splice environment ϕ_2. As we will prove, the injection process is crucial to establish elaboration soundness.

The remaining rules elaborate source expressions in an expected way, while propagating splice environments, e.g. rule S-APP elaborates a source application into a core application, and collects splice environments from preconditions. We talk more about elaborating type classes (rules S-CABS and S-CAPP) in the next section.

5.2 Dictionary-passing Elaboration of Constraints

Figure 2 presents the elaboration of types and constraints. Well-formed source types elaborate to well-formed core types ($\Gamma \vdash \sigma \rightsquigarrow \tau$).

Type classes are translated away by dictionary-passing elaboration [Jones 1994]. In particular, well-formed constraints elaborate to well-formed core types ($\Gamma \vdash C \rightsquigarrow \tau$). Note that a class constraint $\mathsf{TC}\ \tau$ elaborates to its method type, as an instance of the constraint provides an implementation of the method.[6] Accordingly, rule S-CABS elaborates an expression with a constraint into a dictionary-taking function, and rule S-CAPP elaborates class resolution as function applications.

The last judgment $\Theta; \Gamma \models^{p} C \rightsquigarrow e \mid \phi$ is of particular interest: resolving a type class constraint C returns an expression e as evidence for the constraint, with a splice environment ϕ. Rules S-SOLVE-GLOBAL and S-SOLVE-LOCAL are standard elaboration rules of normal type class resolution, where the former uses an instance declaration in the program context, and the latter uses a local instance (as introduced in rule S-CABS).

Rules S-SOLVE-INCR and S-SOLVE-DECR concern staged type class constraints. Rule S-SOLVE-DECR elaborates staged type class constraints into values of type $\mathsf{Code}\ \tau$. Therefore resolution elaboration of staged type class constraints must be understood in terms of quotations. Rule S-SOLVE-DECR is implemented by a simple quotation and thus similar to typing quotations (i.e., rule S-QUOTE). Rule S-SOLVE-INCR conceptually introduces a splice; as in rule S-SPLICE, it achieves this by extending the splice environment, since the core language does not have splices. These rules explain the necessity of level-indexing constraints in the source language: the elaboration would not be well-staged if the stage discipline was not enforced.

5.3 Elaborating Programs with Top-level Splice Definitions

We elaborate programs as shown in gray in Figure 3. For space reasons, we only present the elaboration for programs of the form $e : \tau$ (rule S-PGM-EXPR); elaborations of other forms apply the same idea to the standard elaboration of type class and instance declarations [Bottu et al. 2017; Jones 1994]. The full rules can be found in the appendix.

If a splice occurs at a non-positive level without corresponding surrounding quotations, then it should be evaluated at compile time, and in our formalism, it becomes a top-level splice definition.[7]

[6] This is a simplification of elaboration for multi-method type classes, which produces a *record* with a field for each method.

[7] In general, non-positive splices can still have surrounding quotations. There are two cases. (1) The quotation is not at the corresponding level, then the splice is lifted to top-level splice definition. For example, $[\![\$(\$e)]\!]$ elaborates to **spdef** $\bullet \vdash^{-1} s_2 : \mathsf{Code\ Int} = e; [\![s_1]\!]_{\bullet \emptyset_{s_1:\mathsf{Int}=s_2}} : \mathsf{Code\ Int}$, where s_2 has a surrounding quotation but becomes a **spdef**. (2) The quotation is at the corresponding level, then the splice will be attached to a quotation even if it is non-positive. For example,

This process can be seen from rule S-PGM-EXPR, where we start by elaborating the source expression e at the default level 0, which returns the core expression e and the splice environment ϕ. As we have mentioned in §5.1, elaborating expression at level n maintains the invariant $\phi \doteq n$ (§5.4.1). Since in this case the expression is elaborated at level 0, we have $\phi \doteq 0$; namely, the result ϕ returned from elaborating the expression contains non-positive splice variables that should be evaluated at compile time. Hence, we turn those splice environments into top-level splice definitions and put them before $e : \tau$, using the collapse judgment $\rho gm_1 \vdash^n \phi \rightsquigarrow \rho gm_2$, given in Figure 6. The collapse process takes the current program ρgm_1, and creates top-level splice declarations for each splice in ϕ, generating ρgm_2. To guarantee a stage-correct execution, the splices are inserted in order of their levels, decreasing from n; for rule S-PGM-EXPR, we have $n = -1$. Now ρgm returned from rule S-PGM-EXPR contains exactly what we want: a sequence of top-level splice definitions, followed by the elaborated core expression.

Example 5.1 (Elaboration). The derivation below shows the elaboration of a source program $\$(k)$, where k is a top-level definition defined as $[\![show]\!]$ whose typing derivation has been given in Example 3.1. This illustrates two particular points of interest: CodeC (Show a) is elaborated into quoted evidence using rule C-SOLVE-DECR, and the injection ensures the splices are well-typed.

$\Theta = k : \forall a.\mathsf{CodeC}\ (\mathsf{Show}\ a) \Rightarrow \mathsf{Code}\ (a \to \mathsf{String})$
$\Gamma = a, \boxed{ev :}\ (\mathsf{Show}\ a, 0)$

$\phi_1 = \bullet \vdash^{-1} s : a \to \mathsf{String} = k\ a\ [\![ev]\!]_\bullet$
$\phi_2 = ev : (a \to \mathsf{String}, 0) \vdash^{-1} s : a \to \mathsf{String} = k\ a\ [\![ev]\!]_\bullet$
$\phi_3 = a, ev : (a \to \mathsf{String}, 0) \vdash^{-1} s : a \to \mathsf{String} = k\ a\ [\![ev]\!]_\bullet$

$$\dfrac{\dfrac{\dfrac{\dfrac{k : \forall a.\mathsf{CodeC}\ (\mathsf{Show}\ a) \Rightarrow \mathsf{Code}\ (a \to \mathsf{String}) \in \Theta}{\Theta;\Gamma \vdash^{-1} k : \forall a.\mathsf{CodeC}\ (\mathsf{Show}\ a) \Rightarrow \mathsf{Code}\ (a \to \mathsf{String}) \rightsquigarrow k \mid \bullet}\ \text{S-KVAR}}{\Theta;\Gamma \vdash^{-1} k : \mathsf{CodeC}\ (\mathsf{Show}\ a) \Rightarrow \mathsf{Code}\ (a \to \mathsf{String}) \rightsquigarrow k\ a \mid \bullet}\ \text{S-TAPP} \qquad \dfrac{\dfrac{\boxed{ev :}\ (\mathsf{Show}\ a, 0) \in \Gamma}{\Theta;\Gamma \models^0 \mathsf{Show}\ a \rightsquigarrow ev \mid \bullet}\ \text{S-SOLVE-LOCAL}}{\Theta;\Gamma \models^{-1} \mathsf{CodeC}\ (\mathsf{Show}\ a) \rightsquigarrow [\![ev]\!]_\bullet \mid \bullet}\ \text{S-SOLVE-DECR}}{\Theta;\Gamma \vdash^{-1} k : \mathsf{Code}\ (a \to \mathsf{String}) \rightsquigarrow k\ a\ [\![ev]\!]_\bullet \mid \bullet}\ \text{S-CAPP}}{\Theta;\Gamma \vdash^0 \$(k) : a \to \mathsf{String} \rightsquigarrow s \mid \phi_1}\ \text{S-SPLICE} \qquad \dfrac{\phi_1 +\!\!+ ev : (a \to \mathsf{String}, 0) \rightsquigarrow \phi_2}{}\ \text{S-INJ-CONS}$$

$$\dfrac{\Theta; a \vdash^0 \$(k) : \mathsf{Show}\ a \Rightarrow a \to \mathsf{String} \rightsquigarrow \lambda ev : a \to \mathsf{String}.s \mid \phi_2}{}\ \text{S-CABS} \qquad \dfrac{\phi_2 +\!\!+ a \rightsquigarrow \phi_3}{}\ \text{S-INJ-CONS}$$

$$\dfrac{}{\Theta;\bullet \vdash^0 \$(k) : \forall a.\mathsf{Show}\ a \Rightarrow a \to \mathsf{String} \rightsquigarrow \Lambda a.\lambda ev : a \to \mathsf{String}.s \mid \phi_3}\ \text{S-TABS}$$

Having obtained the main expression, we can apply rule S-PGM-EXPR and use collapse to turn ϕ_3 into a top-level splice definition and form the resulting program:

$$(\Lambda a.\lambda ev : a \to \mathsf{String}.s) : \forall a.(a \to \mathsf{String}) \to a \to \mathsf{String} \vdash^{-1} \phi_3 \quad \rightsquigarrow \quad \mathbf{spdef}\ a, ev : (a \to \mathsf{String}, 0) \vdash^{-1} s : a \to \mathsf{String} = k\ a\ [\![ev]\!]_\bullet;$$
$$(\Lambda a.\lambda ev : a \to \mathsf{String}.s) : \forall a.(a \to \mathsf{String}) \to a \to \mathsf{String}$$

5.4 Elaboration Soundness

In this section, we prove that elaboration preserves types, which, together with type soundness of $F^{[\![]\!]}$, establishes type soundness of $\lambda^{[\![\Rightarrow]\!]}$. To this end, we first need to show how the well-stagedness restrictions in $F^{[\![]\!]}$ (§4.4.1) are satisfied during elaboration.

5.4.1 Well-staged Splice Environments. The first restriction says that every $\Delta \vdash^n s : \tau = e$ has $\Delta \doteq n$ (rules C-SPDEF and C-S-CONS). During elaboration, we have seen that a splice variable captures the local context from its introduction point up to the point where it is bound by a quotation. The restriction holds trivially when a splice variable is created with an empty local context, but since the local context can later be extended by injection we must prove that injection respects the

$\$[\![\$e]\!]$ elaborates to $\mathbf{spdef}\ \bullet \vdash^{-1} s_4 : \mathsf{Int} = [\![s_3]\!]_{\bullet \vdash^{-1} s_3 : \mathsf{Int} = e}; s_4 : \mathsf{Int}$, where s_3 appears at non-positive level but is attached to a quotation. Note that the evaluation order is still correct: since s_4 is evaluated at level -1, its splice environment is evaluated at -1, and thus s_3 is evaluated at -1.

restriction. This can be shown by first proving the invariant that the splice environment produced from typing has level smaller than the current typing level:

Lemma 5.2 (Level Correctness of ϕ). *If* $\Theta; \Gamma \vdash^n e : \tau \rightsquigarrow e \mid \phi$, *then* $\phi \dot{<} n$.

This can be easily seen from rule S-SPLICE that produces only splice variables with smaller levels; and rule S-QUOTE captures all splices at the current level.

We then use Lemma 5.2 to show that injection produces well-staged splice environments. Consider rule S-ABS as an example. By Lemma 5.2 we have $\phi_1 \dot{<} n$, and therefore $\phi_1 \dot{<} x : (\tau, n)$, so injection as in $\phi_1 +\!\!+ x : (\tau, n) \rightsquigarrow \phi_2$ preserves the restriction. Formally, we can prove

Lemma 5.3 (Context Injection). *If* $\Theta; \Delta_1, \Delta_2 \vdash \phi_1$, *and* $\phi_1 \dot{<} \Delta_2$, *and* $\phi_1 +\!\!+ \Delta_2 \rightsquigarrow \phi_2$, *then* $\Theta; \Delta_1 \vdash \phi_2$.

The second restriction requires that an elaborated quotation $\Theta; \Delta \vdash^n [\![e]\!]_\phi$ has $\Theta; \Gamma \vdash^n \phi$. We generate quotations at rule S-QUOTE. As the rule binds $\phi.n$ which by construction has level n, we only need to show $\Theta; \Delta \vdash \phi$, which can be proved making use of Lemma 5.3. In the following lemma statement, the notations $\Theta \rightsquigarrow \Theta$ and $\Gamma \rightsquigarrow \Delta$ elaborate contexts in a unsurprising way; their definitions can be found in the appendix.

Lemma 5.4 (Well-staged ϕ). *If* $\Theta; \Gamma \vdash^n e : \tau \rightsquigarrow e \mid \phi$, *and* $\Theta \rightsquigarrow \Theta$, *and* $\Gamma \rightsquigarrow \Delta$, *then* $\Theta; \Delta \vdash \phi$.

5.4.2 Elaboration Soundness. Now that we have established the key well-stagedness properties of splice environments, we are ready to prove that $\lambda^{[\![\Rightarrow]\!]}$ is type-safe by proving elaboration soundness, which formally establishes our goal: well-typed, well-staged source programs always elaborate to well-typed, well-staged core programs.

Theorem 5.5 (Elaboration Soundness).

(1) *If* $\Theta; \Gamma \vdash^n e : \tau \rightsquigarrow e \mid \phi$, *and* $\Theta \rightsquigarrow \Theta$, *and* $\Gamma \rightsquigarrow \Delta$, *and* $\Gamma \vdash \tau \rightsquigarrow \tau$, *then* $\Theta; \Delta, \phi^\Gamma \vdash^n e : \tau$.

(2) *If* $\Theta \vdash pgm : \sigma \rightsquigarrow \rho gm$, *and* $\Theta \rightsquigarrow \Theta$, *then* $\Theta \vdash \rho gm$.

6 AXIOMATIC SEMANTICS

Our goal in designing $\lambda^{[\![\Rightarrow]\!]}$ and $F^{[\![]\!]}$ is to provide a theoretical foundation for multi-stage programming. It is thus important to show that our formalism enjoys desirable properties. One such property is that splices and quotations are dual to each other, which provides a simple reasoning principle for multi-stage programming, and allows programmers to *cancel splices and quotations out* without worrying about changing the semantics of programs.

In this section, we prove this crucial property by establishing axioms and axiomatic semantics of $\lambda^{[\![\Rightarrow]\!]}$ and $F^{[\![]\!]}$ respectively, and show that canceling out splices and quotations leads to *contextually equivalent* programs. The definitions of axiomatic semantics and the proofs in this section follow Taha et al. [1998] and Taha [1999], with key novelties in that (1) $\lambda^{[\![\Rightarrow]\!]}$ has elaboration-based semantics, and thus the correctness of its axioms are built on that of $F^{[\![]\!]}$, and this indirection poses extra complexities in the proofs; and (2) for $F^{[\![]\!]}$, we define the axiomatic semantics and extend the proofs for our novel splice environments and top-level splice definitions.

6.1 Duality of Splices and Quotations in $\lambda^{[\![\Rightarrow]\!]}$

The property we seek to establish is captured by the two axioms of $\lambda^{[\![\Rightarrow]\!]}$ given in Figure 7a, which state that splicing a quotation or quoting a splice is equivalent to the original expression: they respectively represent eta and beta laws for *Code*. These axioms form part of the equational theory of $\lambda^{[\![\Rightarrow]\!]}$; they can be thought of as context-independent pattern-based rewriting rules.

Consider an axiomatic equivalence relation between $\lambda^{[\![\Rightarrow]\!]}$ programs that is the contextual and equivalence closure of the axioms, which we denote as $pgm_1 =_{ax} pgm_2$. Our goal now is to prove

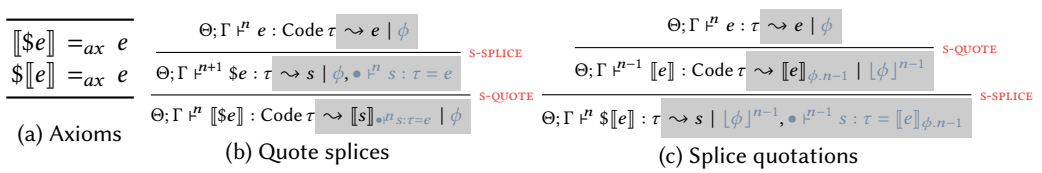

Fig. 7. Axioms and elaboration derivations in $\lambda^{[\![\Rightarrow]\!]}$

axiomatically equivalent source programs are *contextually equivalent*, i.e. they always produce the same result and thus can be used in an interchangeable way. As the dynamic semantics of $\lambda^{[\![\Rightarrow]\!]}$ is defined based on elaboration to $F^{[\![]\!]}$, we build the proofs based on the axiomatic semantics of $F^{[\![]\!]}$.

6.2 Axiomatic Semantics of $F^{[\![]\!]}$

The axiomatic semantics of $F^{[\![]\!]}$ is guided by the elaboration of the $\lambda^{[\![\Rightarrow]\!]}$ axioms. Supposing source e elaborates to core e with ϕ, Figures 7b and 7c present elaboration derivations of $[\![\$e]\!]$ and $\$[\![e]\!]$ respectively. Looking first at Figure 7b, what is needed to show the first $\lambda^{[\![\Rightarrow]\!]}$ axiom is a $F^{[\![]\!]}$ axiom that models the equivalence between expression $[\![s]\!]_{\bullet \vdash^n s:\tau=e}$ with ϕ (the elaboration result of $[\![\$e]\!]$) and e with ϕ (the elaboration result of e). Since the two ϕs are the same, it is sufficient to introduce a core axiom $[\![s]\!]_{\bullet \vdash^n s:\tau=e} =_{ax} e$.

The case for splicing quotations (Figure 7c) is more challenging: in this case we cannot directly compare the elaborated expressions, as the generated splice environments are different. Instead, we need to consider equivalence between two core quotations where the splice environments are bound. To derive the axiom, let us first consider the case where both expressions are bound immediately to a quotation. That leads to $[\![s]\!]_{\lfloor\phi\rfloor^{n-1}, \bullet \vdash^{n-1} s:\tau=[\![e]\!]_{\phi.n-1}} =_{ax} [\![e]\!]_{\lfloor\phi\rfloor^{n-1}, \phi.n-1}$. Abstracting over the specific shape of splice environments gives us $[\![s]\!]_{\phi_1, \bullet \vdash^n s:\tau=[\![e]\!]_\phi} =_{ax} [\![e]\!]_{\phi_1,\phi}$. In the case when s is not immediately bound, we then have $[\![e_1]\!]_{\phi_1, \bullet \vdash^n s:\tau=[\![e]\!]_\phi} =_{ax} [\![e_1[s \mapsto e]]\!]_{\phi_1,\phi}$. However, there are still some wrinkles to this axiom. First, s could have a non-empty splice environment ϕ_2 to its right, as until s is bound there can be more splices. Second, s could have a non-empty local context Δ, as until s is bound it may have got out of some scopes and so have applied the injection process. Finally, if s has a non-empty local context, then after it is substituted away on the right hand side, we cannot directly discard its local context Δ and leave ϕ, since ϕ now becomes ill-typed as it loses the scope of the variables from Δ. Therefore, we need to inject Δ into ϕ.

Summarizing our discussion, we end up with the axiomatic semantic of $F^{[\![]\!]}$ as defined below. Note that splicing quotations also leads to the equivalence axiom between **spdef**.

Definition 6.1 (Axiomatic Semantics of $F^{[\![]\!]}$). *Axiomatic semantics of $F^{[\![]\!]}$ models β-equivalence, as well as the following axioms.*

$[\![s]\!]_{\bullet \vdash^n s:\tau=e}$	$=_{ax}$	e	
$[\![e_1]\!]_{\phi_1, \Delta \vdash^n s:\tau=[\![e]\!]_\phi, \phi_2}$	$=_{ax}$	$[\![e_1[s \mapsto e]]\!]_{\phi_1,\phi',\phi_2}$	*where* $\phi +\!\!+ \Delta \rightsquigarrow \phi'$
spdef $\Delta \vdash^n s : \tau = [\![e]\!]_\phi; \rho gm$	$=_{ax}$	**spdef** $\phi'; \rho gm[s \mapsto e]$	*where* $\phi +\!\!+ \Delta \rightsquigarrow \phi'$

Now consider an axiomatic equivalence relation between $F^{[\![]\!]}$ programs that is the contextual and equivalence closure of the axioms, denoted as:

$$\Theta \vdash \rho gm_1 \backsimeq_{ax} \rho gm_2 \triangleq \Theta \vdash \rho gm_1 \wedge \Theta \vdash \rho gm_2 \wedge \rho gm_1 =_{ax} \rho gm_2$$

To show that our definition of axiomatic semantics of $F^{[\![]\!]}$ indeed captures the desirable duality between splices and quotations, we prove that axiomatically equivalent source programs elaborate to axiomatically equivalent core programs.

Lemma 6.2 ($\lambda^{[\![\Rightarrow]\!]}$ $=_{ax}$ to $F^{[\![]\!]}$ $\backsimeq_{ax}$). *If* $pgm_1 =_{ax} pgm_2$, *where* $\Theta \vdash pgm_1 : \sigma \rightsquigarrow \rho gm_1$, *and* $\Theta \vdash pgm_2 : \sigma \rightsquigarrow \rho gm_2$, *and* $\Theta \rightsquigarrow \Theta$, *then* $\Theta \vdash \rho gm_1 \backsimeq_{ax} \rho gm_2$.

With this lemma, now we can use core axiomatic equivalence as an intermediate step to show that source axiomatic equivalence derives core contextual equivalence.

6.3 Contextual Equivalence

We define contextual equivalence in $F^{[\![]\!]}$ as below.

Definition 6.3 (Contextual Equivalence in $F^{[\![]\!]}$).

$$\begin{aligned}
\bullet;\Gamma \Vdash^n e_1 \backsimeq_{ctx} e_2 : \tau \quad &\triangleq \quad \bullet;\Gamma \Vdash^n e_1 : \tau \wedge \bullet;\Gamma \Vdash^n e_2 : \tau \\
&\quad \wedge\ (\forall \mathbb{C} : \bullet;\Gamma \Vdash^n \tau \rightsquigarrow \bullet;\bullet \Vdash^0 \mathsf{Int},\ \mathbb{C}[e_1] \longrightarrow^* i \Longleftrightarrow \mathbb{C}[e_2] \longrightarrow^* i) \\
\Theta \vdash \rho gm_1 \backsimeq_{ctx} \rho gm_2 : \tau \quad &\triangleq \quad \Theta \vdash \rho gm_1 \wedge \Theta \vdash \rho gm_2 \ \wedge\ (\forall \overline{\mathcal{S}_i, \mathcal{D}_j}^{\,i,j} : \Theta \vdash \tau \longrightarrow \bullet \vdash \tau, \\
&\quad (\overline{\mathbf{spdef}\,\mathcal{S}_i; \mathbf{def}\,\mathcal{D}_j}^{\,i,j}; \rho gm_1 \longrightarrow^* e_1 : \tau \Longleftrightarrow \overline{\mathbf{spdef}\,\mathcal{S}_i; \mathbf{def}\,\mathcal{D}_j}^{\,i,j}; \rho gm_2 \longrightarrow^* e_2 : \tau) \\
&\quad \wedge\ (\bullet;\bullet \Vdash^0 e_1 \backsimeq_{ctx} e_2 : \tau))
\end{aligned}$$

Expression contextual equivalence says that two core expressions e_1 and e_2 are contextually equivalent, if for any *computation context* $\mathbb{C}$, $\mathbb{C}[e_1]$ and $\mathbb{C}[e_2]$ evaluate to the same value. A computation context $\mathbb{C}$ is a core expression with a hole in it, and we use the notation $\mathbb{C}[e]$ to plug in the expression e into the hole of $\mathbb{C}$. The notation $\mathbb{C} : \bullet;\Gamma \Vdash^n \tau \rightsquigarrow \bullet;\bullet \Vdash^0 \mathsf{Int}$ means that if $\bullet;\Gamma \Vdash^n e : \tau$ then $\bullet;\bullet \Vdash^n \mathbb{C}[e] : \mathsf{Int}$. Program contextual equivalence is defined in a similar manner and is built using expression contextual equivalence.

The final piece in our proof is to show that axiomatically equivalent core programs are contextually equivalent, then with Lemma 6.2 we can prove that axiomatically equivalent source programs elaborate to contextually equivalent core programs. The proofs follow those of Taha et al. [1998] and Taha [1999], which are omitted for space reasons. At a high level, this lemma requires us to build *parallel reduction* of $F^{[\![]\!]}$ to prove the Church-Rosser property, which is then used to prove equivalence between $F^{[\![]\!]}$ axiomatic semantics and operational semantics.

Lemma 6.4 ($F^{[\![]\!]}$ $\backsimeq_{ax}$ to $F^{[\![]\!]}$ $\backsimeq_{ctx}$). *If* $\Theta \vdash \rho gm_1 \backsimeq_{ax} \rho gm_2$, *then* $\Theta \vdash \rho gm_1 \backsimeq_{ctx} \rho gm_2 : \tau$.

Combining Lemma 6.2 and Lemma 6.4 yields our final goal:

Theorem 6.5 ($\lambda^{[\![\Rightarrow]\!]}$ $=_{ax}$ to $F^{[\![]\!]}$ $\backsimeq_{ctx}$). *If* $pgm_1 =_{ax} pgm_2$, *where* $\Theta \vdash pgm_1 : \sigma \rightsquigarrow \rho gm_1$, *and* $\Theta \vdash pgm_2 : \sigma \rightsquigarrow \rho gm_2$, *and* $\Theta \rightsquigarrow \Theta$, *and* $\bullet \vdash \sigma \rightsquigarrow \tau$, *then* $\Theta \vdash \rho gm_1 \backsimeq_{ctx} \rho gm_2 : \tau$.

7 TODAY'S TYPED TEMPLATE HASKELL

The behavior of Typed Template Haskell in GHC differs from our calculus. Figure 8 summarizes the key examples from §2, comparing the results from the latest GHC (9.0.1) to $\lambda^{[\![\Rightarrow]\!]}$. The Haskell code examples are in the appendix.

At a high level, GHC's implementation is close to the description in §2.4: it delays type class elaboration until splicing, and excludes local constraints for top-level splices. This is sufficient to accept the definitions of *print1* (and *qnpower*) and *trim*, but it restricts their use: *print1* can only be spliced with a monomorphic type signature, and *trim* can never be spliced, despite its semantics being clear. The central guarantee of typed code quotations is that well-typed code values represent well-typed programs; we view GHC's behavior, in which splicing a well-typed code value can

	print1	*printInt*	*print2*	*topLift*	*trim*	*cancel*	*qnpower*/*npower5*	
	C1	C2	S1	TS1	A1	A2	§1	S2
Well-staged?	✗	✓	✓	✗	✓	✓	✗/ ✗	✓/ ✓
$\lambda^{[\![\Rightarrow]\!]}$	✗	✓	✓	✗	✓	✓	✗/ ✗	✓/ ✓
GHC 9.0.1	✗	✓	*O*	✗	✗	✗	✗/ ✗	*O*

Fig. 8. Examples comparison. Well-staged? indicates well-stagedness after dictionary-passing elaboration. ✗ means the definition itself is accepted but its use is restricted; and *O* means not applicable.

raise a type error, as unsound. Even where it does not lead to unexpected splice-time type errors, delaying type class elaboration can unexpectedly change the semantics of a program when the definition site and the splicing site have different instances in scope.

Finally, because GHC excludes local constraints for top-level splices, it (accidentally) correctly rejects *topLift* (and *npower5*) but wrongly rejects *cancel*. We argue that *topLift* should be rejected because it is ill-staged, and *cancel* should be accepted both because it is well-staged, and because canceling a splice-quotation pair should preserve semantics.

8 INTEGRATION INTO GHC

The goal of this work is to formally study the interaction of type classes and staging, along with the formalism of splice environments, and so we have focused on a source calculus that captures their essence. Integrating our solution into GHC will require additional steps, which we touch on briefly here.

Type inference. We anticipate that type inference for staged constraints will be straightforward to integrate into existing constraint solving algorithms (e.g. Vytiniotis et al. [2011]). The key modification is to track the level of constraints and only solve goals with evidence at the right level. In our formalism, constraints can be solved either by rule S-SOLVE-INCR or by rule S-SOLVE-DECR. In practice, the implementation only needs to keep track of the level of normal constraints (e.g. when given *CodeC C* at level 0, the context can record the spliced evidence for *C* at level 1) so that constraint solving only needs to consider rule S-SOLVE-DECR.

Local constraints. Local constraints can be introduced by (for example) pattern matching on GADTs [Peyton Jones et al. 2006], and we anticipate that they can be treated similarly to type class constraints: the constraint solver needs to keep track of the level at which a constraint is introduced and ensure that the constraint is only used at that level.

Quantified constraints. The full Haskell language supports more elaborate forms of type classes than the essence modeled in $\lambda^{[\![\Rightarrow]\!]}$. For example, GHC supports *quantified constraints* [Bottu et al. 2017], which include forms such as $\forall x.\mathit{Show}\ x \Rightarrow \mathit{Show}\ (f\ x)$, a constraint that converts *Show* instances for x into *Show* instances for $f\ x$. Future work is required to study more formally the interaction between staged constraints and implication constraints; we envisage that constraint entailment should deduce that $\mathit{CodeC}\ (C_1 \Rightarrow C_2)$ entails $\mathit{CodeC}\ C_1 \Rightarrow \mathit{CodeC}\ C_2$.

Representation of quotations. In today's GHC implementation, untyped code representations are built compositionally using combinators, and type-checked at splice sites. With our development, code representations contain type information, especially dictionaries, and must therefore correspond to one of GHC's post-typechecking term representations. One option is GHC Core terms, which is the simplest representation that retains type information and has existing serialization support (for inlining definitions across modules).

Our development also requires changing the implementation of splicing to support performing substitution at splices inside quotations. In today's GHC, substitution is performed implicitly during translation from expressions to combinators. With the new representation of quotations, the substitution needs to be represented explicitly and performed explicitly during deserialization of the quotation body. Substituting splices takes two steps. First, a quotation body is traversed and each splice is replaced by a splice variable where the evaluated splice term needs to be inserted. The splice variable is maintained in the splice environment. Second, the splicing operation itself involves checking the splice environment for each splice variable and performing the substitution.

9 RELATED WORK

Since its introduction [Taha and Sheard 1997, 2000] multi-stage programming with quotation has attracted both theoretical and practical interest. Several languages, including MetaOCaml [Kiselyov 2014], Haskell and Scala [Stucki et al. 2018], include implementations of typed quotations.

Considering that implementations of multi-stage languages have supported polymorphism from the very beginning, there is surprisingly little work that formally combines multi-stage programming with polymorphism: most multi-staged calculi are simply-typed. An exception, by Kokaji and Kameyama [2011], involves a language with polymorphism and control effects; their primary concern is the interaction of the value restriction and staging. Another, by Kiselyov [2017], considers the tripartite interaction of polymorphism, cross-stage persistence and mutable cells.

Several works examine the interaction of quotation with individual language features, particularly with various forms of effects, such as control operators [Oishi and Kameyama 2017] and mutable cells [Kiselyov et al. 2016]. Work by Yallop and White [2015] is more closely related to the present work, since there is a well-known correspondence between ML modules and type classes [Wehr and Chakravarty 2008]; it examines the interaction between typed compile-time staging and modules. However, since modules are written explicitly rather than introduced by elaboration, the dictionary level problem does not arise. In a similar vein, Radanne [2017] studies the interaction of ML modules with a different modality, client-server programming, where the distinction between client and server functors corresponds to our distinction between unstaged and staged type class constraints.

Several researchers have combined multi-stage programming and dependent types. Kawata and Igarashi [2019] impose a stage discipline on type variables as on term variables, reflecting the fact that checking types involves evaluating expressions. Pašalic [2004] defines a dependently-typed multi-stage language Meta-D but doesn't consider constraints or parametric polymorphism. Concoqtion [Fogarty et al. 2007] extends MetaOCaml to support Coq terms within types; it is based on the dependently-typed $\lambda_{H\bigcirc}$ [Pašalic et al. 2002], which is motivated by removing tags in generated programs. Brady and Hammond [2006] combine dependent types and multi-stage programming to turn a well-typed interpreter into a verified compiler, but do not consider either parametric polymorphism or constraints.

We are not aware of any work that considers the implications of relevant implicit arguments formally, but there is an informal characterization by Pickering et al. [2019], who also advocated persisting dictionaries between stages, using the fact that dictionary values have top-level names. Unfortunately, that scheme, based on extending the constraint solver to select dictionary representations using both type and level, does not readily extend to local constraints. An alternative approach that the authors later considered, passing constraint derivation trees to allow local construction of future-stage dictionaries, was judged to carry too much run-time overhead to be practical.

Formalising Template Haskell. Sheard and Jones [2002] give a brief description of Untyped Template Haskell. The language is simply-typed and does not account for multiple levels. The language has since diverged: untyped quotations are no longer typechecked before conversion

into their representation. Some aspects of their formalism, notably the Q monad which supports reification of types and declarations, are more suited to the untyped metaprogramming than the typed multi-stage programming we consider here. [Berger et al. 2017] give a more formal study of a core calculus that models some aspects of Untyped Template Haskell, focusing on levels and evaluation rather than these additional features.

Code generators often make use of effects such as let insertion or error reporting so it is useful for to consider the interaction of quotation with effects. In GHC releases since 8.12, the type of quotations is generalised [Pickering 2019] from Q ($TExp\ a$) to a minimal interface $\forall m.Quote\ m \Rightarrow m\ (TExp\ a)$ giving users more control over which effects are allowed in code generators. We leave formalising this extension to future work.

Modal Type Systems. Several type systems motivated by modal logics have modeled the interaction of modal operators and polymorphism. 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 likely that ideas from this 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].

10 CONCLUSION

We have proposed a resolution to a longstanding problem in Typed Template Haskell arising from the interaction beteen two key features, code quotation and type classes. In our view, the mysterious failures that can arise when writing large-scale multi-stage programs are one reason for the limited adoption of Typed Template Haskell. Although it is used in a few developments (e.g. Pickering et al. [2020]; Willis et al. [2020]; Yallop et al. [2018]), take-up is low, despite the many use cases for type-safe optimizing code generators. We hope that the resolution of the shortcomings we have described and the reasoning principles we have established will encourage broader adoption.

Although our work is inspired by Haskell, there is reason to believe that it has wider applications. The recent release of Scala 3 added support for typed code quotations to the language [Stucki et al. 2018]. Preliminary experiments suggest that these quotations suffer from surprising interactions with implicit arguments: implicit resolution within quotations sometimes fails mysteriously. Similarly, it is anticipated that OCaml will soon acquire support both for typed code quotations [Yallop and White 2015] and for implicit arguments [White et al. 2014]. We hope that our work will help to guide the integration of these features and avoid problems with unsoundness from the outset.

ACKNOWLEDGMENTS

We thank Dimitrios Vytiniotis, and the anonymous reviewers for their insightful comments. The work is partly supported by EPSRC Grant *SCOPE: Scoped Contextual Programming with Effects* (EP/S028129/1) and Grant *EXHIBIT: Expressive High-Level Languages for Bidirectional Transformations* (EP/T008911/1).

REFERENCES

Martin Berger, Laurence Tratt, and Christian Urban. 2017. Modelling Homogeneous Generative Meta-Programming. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:23. https://doi.org/10.4230/LIPIcs.ECOOP.2017.5

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

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

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

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

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.

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

Yuichiro Hanada and Atsushi Igarashi. 2014. On Cross-Stage Persistence in Multi-Stage Programming. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 103–118. https://doi.org/10.1007/978-3-319-07151-0_7

M.P. Jones. 1993. *Coherence for qualified types.* Research Report YALEU/DCS/RR-989. Yale University, Dept. of Computer Science.

Mark P. Jones. 1994. *Qualified Types: Theory and Practice.* Cambridge University Press.

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, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 271–291. https://doi.org/10.1007/978-3-319-47958-3_15

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

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

Junpei Oishi and Yukiyoshi Kameyama. 2017. Staging with control: type-safe multi-stage programming with control operators. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 29–40. https://doi.org/10.1145/3136040.3136049

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 Geoffrey Washburn. 2006. Simple Unification-based Type Inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/1159803.1159811

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, Andres Löh, and Nicolas Wu. 2020. Staged sums of products. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, Tom Schrijvers (Ed.). ACM, 122–135. https://doi.org/10.1145/3406088.3409021

Matthew Pickering, Nicolas Wu, and Csongor Kiss. 2019. 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

Gabriel Radanne. 2017. *Tierless Web programming in ML. (Programmation Web sans-étages en ML).* Ph.D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-01788885

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. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408971

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 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

Walid Taha, Tim Sheard, et al. 1998. Multi-stage programming: Axiomatization and type safety. In *International Colloquium on Automata, Languages, and Programming*. Springer, 918–929.

Walid Mohamed Taha. 1999. *Multistage programming: its theory and applications.* Oregon Graduate Institute of Science and Technology.

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

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

Stefan Wehr and Manuel M. T. Chakravarty. 2008. ML Modules and Haskell Type Classes: A Constructive Comparison. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5356)*, G. Ramalingam (Ed.). Springer, 188–204. https://doi.org/10.1007/978-3-540-89330-1_14

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014 (EPTCS, Vol. 198)*, Oleg Kiselyov and Jacques Garrigue (Eds.). 22–63. https://doi.org/10.4204/EPTCS.198.2

Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged selective parser combinators. *Proc. ACM Program. Lang.* 4, ICFP (2020), 120:1–120:30. https://doi.org/10.1145/3409002

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

Jeremy Yallop and Leo White. 2015. Modular Macros. OCaml Users and Developers Workshop.