Author:
Pickering, Matthew T

Title:
Understanding the interaction between elaboration and quotation

Understanding the Interaction Between Elaboration And Quotation

Matthew Pickering

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Engineering

School of Computer Science

June 13, 2021

Word Count:
60121

Abstract

Multi-stage programming languages have long promised programmers the means to program libraries with specific performance guarantees. Despite this, the adoption into mainstream high-level languages such as Haskell, Scala and OCaml has been very slow. In particular, our focus, Typed Template Haskell has been implemented for a number of years but the ecosystem has failed to adopt the multi-stage ideas. The situation hasn't been helped by a number of soundness and expressivity issues with the current implementation.

In this thesis we tackle the problem of soundly combining multi-stage features with other features as commonly found in modern programming languages. The main contribution is the design and implementation of a language which combines multiple stages with implicit arguments and an elaboration phase. The goal is to provide a firmer foundation for future implementations and to enable library writers to use multi-stage features with confidence in conjunction with the rest of the Haskell language.

Author's Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Matthew Pickering
June 13, 2021

Acknowledgments

I would firstly like to thank my supervisor Nick who has never failed to believe in me from the first day we met on a bench in Oxford during the summer of 2014. It is his belief and support which has enabled me to become an independent researcher and have confidence to follow my own research ideas even if they are not wholly aligned with his own. Despite this, he always has a suggestion for any problem and is a dabhand with the minipage.

My master's supervisor Jeremy also played a large role in me getting to grips with how to research and most importantly, how to write. Jeremy taught me that the best ideas are the simplest ones, and that a quality explanation is as valuable as a quality idea. Two other undergraduate tutors also deserve special mention: Mike Spivey and Alex Paseau.

I am also deeply indebted to Andres Löh who has been responsible for asking many of the questions which led to my interest in the topics of this thesis. At the start of my PhD Andres was interested in optimising GENERICS-SOP and told me about the specialisation trick with type classes. This led me to attempt to understand how the optimiser worked and to conclude it wasn't a good way to design a library. Then I became interested in multi-stage programming and ultimately my research made it possible to design an efficient version of GENERICS-SOP using Typed Template Haskell. Andres is also remarkably patient and constructs very nice small counter-examples when required.

All of the GHC developers and people who have made attending conferences and workshops such an enjoyable experience throughout my PhD. These trips are a highlight of my year, staying in the GHC house has led to some great patches. So thank you: Ben, Ryan, David, Csongor, Andreas and Jamie for the trips to Philadelphia, St Louis and Berlin.

Of course, once I gained some office-mates, they were also invaluable in coming up with ideas and listening to me go on about the partial evaluation textbook at any opportunity. So thank you: Jamie, Csongor, Eddie, Alessio for your invaluable help.

Personal thanks also extends to my family and parents for their support over the years. Also to Ashok and Hazel who have been supportive over many years.

When I moved to Bristol at the end of 2016, I knew no one in the city and decided to start running and orienteering as a new hobby. It turned out to be one of the best decisions I ever made and in the last four years have met a multitude of amazing friends and shared so many amazing experiences. So thank you to my friends from this time: Michael, Tom, Ben, Megan, Kit, Dan, Joel, Rob, Nick, Matt, Duncan and Tamsin for all your support and encouragement especially during the last few very difficult months.

Contents

Chapter 1

Introduction

Multi-stage programming languages offer the perfect high-level solution to writing performant libraries. A multi-stage language provides two constructs, quotes and splices, which can be used to separate a program into levels. Each level of a program is guaranteed to evaluate in order which means that by evaluating up-to a specific level then part of the computation can be performed ahead of the rest of the compilation. In Haskell, the multi-stage features are known as Typed Template Haskell (Mainland and Peyton Jones, 2010) and the separation between levels amounts to performing some evaluation at compile-time and deferring the rest until runtime. Multi-stage languages make it easy to write program generators in a natural style by refactoring existing programs.

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

The classic example of staging is the power function, where the value n^k can be efficiently computed for a fixed k by generating code where the required multiplications have been unrolled and inlined. The power function is a code generator which is indicated by the return type of the function being of type Code. The dynamic argument is indicated by the type of Code Int and the statically known argument by the type Int.

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

Any value $\mathsf{n} :: \mathsf{Int}$ can be quoted to create $[\![\ \mathsf{n}\ ]\!] :: \mathsf{Code\ Int}$, then spliced in the expression $\$(\mathsf{power}\ 5\ [\![\ \mathsf{n}\ ]\!])$ to generate $\mathsf{n} * (\mathsf{n} * (\mathsf{n} * (\mathsf{n} * (\mathsf{n} * 1))))$.

Not only this, but the power of a multi-stage language is amplified when coupled with other advanced type system features. The focus of this thesis is Typed Template Haskell and the recent advancements to Haskell's type system to elevate it towards a dependently typed language have meant that the type of a code generator can be made more precise. The extra precision can mean generating more straight-forward code with less runtime checks which results in a simpler and faster program (Pašalic et al., 2002). A staged language guarantees that any abstraction overhead from the more complex types is also removed by runtime.

The problem is that the combination of an advanced type system with multi-stage features has not yet been studied in detail. Early study into multi-stage languages focused on extending the languages to prevent scope extrusion (Taha and Nielsen, 2003) – modern implementations do not implement these suggested extensions (Kiselyov, 2014) and instead encounter problems combining together features such as implicit arguments and dependent types. Some recent work has tackled challenges related to dependent types (Kawata and Igarashi, 2019) but to our knowledge, the interaction of quotations and implicit arguments has not been deeply considered. This results in an unsound implementation of a multi-stage language. It is possible to produce a value of type $\mathsf{Code}\ \mathsf{T}$ that when spliced fails to produce a program which when executed will produce a value of type T. Here are two simple examples of this failure.

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

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

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

$$\mathsf{trim} :: \mathsf{Code}\ (\mathsf{String} \to \mathsf{String})$$
$$\mathsf{trim} = [\![\ \$(\mathsf{qshow}) \circ \$(\mathsf{qread})\ ]\!]$$

However, when trying to put the function to use by invoking $\$(\mathsf{trim})$ `"  1"`, GHC 8.10.1 rejects it with an error:

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

What happened? By the time GHC has stitched together trim, all that's left is show∘read, where the intermediate type is indeed ambiguous. This is bad, because the intended

meaning of trim is clearly to normalise Ints because of the types given to qshow and qread.

What this means is that the representation ought to contain the implicit information that the compiler holds *at the time of quotation*. This issue has previously been ignored, leading to unexpected behaviour in programs that make sophisticated use of implicit information and quotation.

As another example, consider the slight generalisation of the power function from before, this time so that the type of the numeric value is overloaded using the Num type class:

```
power :: Num a ⇒ a → Code a → Code a
power 0 cn = ⟦ 1 ⟧
power k cn = ⟦ $(cn) ∗ $(power (k − 1) cn) ⟧
```

You may hope to use this function as a polymorphic code generator which could be instantiated to many different numeric types. It is somewhat surprising, then, that the following function fails to compile in the latest implementation of Typed Template Haskell in GHC 8.10.1:

```
power5 :: Num a ⇒ a → a
power5 n = $(power 5 ⟦ n ⟧)
```

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

The structure of the thesis is as follows:

- Chapter 2 we will introduce the background material which explains Typed Template Haskell and a general introduction to multi-stage programming.
- In Chapter 3 many examples of how Typed Template Haskell is unsound will be presented. These are all of a similar flavour to the examples considered in the introduction and so we will identify the need for a typed internal representation and a more careful treatment of implicit arguments such as constraints. The main contribution of this chapter is a formalism of Typed Template Haskell

which introduces a new constraint form CodeC which can manipulate the level of constraints.

- Chapter 4 describes the technical details of the implementation of the formalism presented in the previous chapter. In particular the precise form the proposed representation will be discussed along with particular details about how the constraint solver and type checking of GHC was extended in order to support our proposed language changes.

- Chapter 5 gives a detailed explanation of how inlining and specialisation can be combined together to write optimised libraries in Haskell. These tranformations are fragile, which motivates why staged programming is necessary in a language with a sophisticated automatic optimiser. The prevailing wisdom in the Haskell community has been that the optimiser should be responsible for removing any abstraction overhead from programs. For simple examples this may be true but relying on the optimiser for any serious library is fraught with difficulties. The chapter concludes with an example of writing an efficient generic traversal library in both the automatic and staged styles.

- Chapter 6 explores an alternative language design which can support a more expressive form of cross stage persistence. This theoretical chapter extends Levy's CBPV calculus to make it multi-staged and expressive enough as a compilation target to lift function values. The motivation is to be able to provide a consistent user interface for a staged and unstaged library which uses higher-order functions.

Chapter 7 discusses the related implementations of multi-stage languages and other modal features currently implemented before we briefly conclude in Chapter 8.

The result of the thesis is both a theoretical and practical understanding of how to design and implement a multi-stage language which interacts correctly with a variety of common language features which require an elaboration phase.

The technical material in this thesis is based on a collection of papers and proposals published during the last four years.

- Kiss, C., Pickering, M., and Wu, N. (2018). Generic deriving of generic traversals. *Proc. ACM Program. Lang.*, 2(ICFP)

- Pickering, M., Wu, N., and Kiss, C. (2019a). Multi-stage programs in context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*,

Haskell 2019, page 71–84, New York, NY, USA. Association for Computing Machinery

- Pickering, M., Wu, N., and Németh, B. (2019b). Working with source plugins. In *Proceedings of the 2019 ACM SIGPLAN Symposium on Haskell*, Haskell '19, New York, NY, USA. Association for Computing Machinery
- Pickering, M. (2019). Overloaded quotations. GHC proposal 246
- Pickering, M. (2020). Make Q (TExp a) into a newtype. GHC proposal 195
- Willis, J., Wu, N., and Pickering, M. (2020). Staged selective parser combinators. *Proc. ACM Program. Lang.*, 4(ICFP)
- Pickering, M., Löh, A., and Wu, N. (2020). Staged sums of products. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, Haskell 2020, pages 122–135, New York, NY, USA. Association for Computing Machinery

Chapter 2

Background

In this chapter the necessary background material for the thesis is introduced. We will start by giving a general description of a multi-stage programming language in the style of MetaML (Taha and Sheard, 1997) before an in-depth description is given of the current implementation of Typed Template Haskell.

2.1 Multi-Stage Programming with Typed Template Haskell

We will start by giving the basic principles of a multi-stage programming language and the properties which you expect to hold. We will use the syntax of Typed Template Haskell as the language to talk about these general concepts as it is the main focus of this thesis.

Typed Template Haskell is a feature of the Glasgow Haskell Compiler (GHC) (Marlow and Peyton Jones, 2012) which extends the Haskell language (Hudak et al., 1992) with support for multi-stage constructs. The thesis describes Typed Template Haskell as implemented in GHC-8.10.1.

The purpose of a multi-stage programming language is to provide the means to the programmer to separate their program into *stages*. Stages are usually labelled numerically, during executation of a staged language, parts of the program at stage k will be evaluated before parts which are at stage $k + 1$. In addition to this, the program executed at stage k generates part of the program which will run at stage $k+1$. Typically in a program there are two stages, compiling and running a program. Parser generators can be thought of as having three stages, generating a file from the grammar, compiling

and then running a program. Multi-stage programming gives the means to understand the interaction between different stages from within the same language.

The guarantee that a staged language provides is that the evaluation of stage k doesn't depend on any values calculated at later stages. The consequence of this guarantee is that a staged program can be partially evaluated up to a certain number of stages. One primary use of this property is in order to generate more efficient programs if all the information upto stage k is known ahead of time.

The soundness guarantee for a typed multi-stage language states that if a program type checks then it will produce a well-typed program when it is executed. Typed multi-stage programs are type-checked *once* before any stages are executed. Then when stage k is executed to produce the program to be executed at stage $k + 1$, the resulting code is not typechecked again, it is correct by construction. This concept forms a central part of this thesis and is discussed further in Section 3.1.

We will in particular care about multi-stage languages which provide explicit annotations for the user to dictate which stage an expression should be evaluated in. The first such language was MetaML (Taha and Sheard, 1997) which extended a polmorphic lambda calculus with staging annotations. In more recent years, MetaOCaml (Kiselyov, 2014) has developed at the most widely used implementation. Otherwise, the most notable implementations are in Scala 3 (Stucki et al., 2018) and our focus, Typed Template Haskell.

2.1.1 Syntax of Multi-Stage Programming

The two standard staging annotations of multi-stage programming are *quotes* and *splices*. An expression $\mathsf{e} :: \mathsf{a}$ can be quoted to generate the expression $[\![\ \mathsf{e}\ ]\!] :: \mathsf{Code\ a}$. Conversely, an expression $\mathsf{c} :: \mathsf{Code\ a}$ can be spliced to extract the expression $\$(\mathsf{c})$. An expression of type $\mathsf{Code\ a}$ is a representation of an expression of type a.

The standard example of a staged function which we saw in the introduction is the power function which computes n^k for a specific exponent k.

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

This example makes use of quotes and splices. The references to the variables power, k and cn are all at the same level as their binding occurrence, as they occur within one splice and one quote. The static information is the exponent k and the run-time

information is the base cn. Therefore by using the staged function the static information can be eliminated by partially evaluating the function at compile-time by using a top-level splice. The generated code does not mention the static information.

Once a code generator has been constructed by inserting quotes and splices in the correct places then it can be executed by using a *top-level splice*. In Template Haskell, top-level splices are run during compile-time and share the same syntax as nested splices inside quotations. In the following program the power code generator is executed using the statically known exponent 5 and unknown base n.

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

The power function is executed and the generated program is inserted in place of the splice. The result is that the definition of power is unfolded five times.

Code generators can get a lot more complicated than the simple example of unrolling a loop in the power case but the overall concepts do not. It is surprising how much can be achieved with a pair of simple language constructs and also perhaps an explanation as to why multi-stage languages have fallen out of research fashion.

2.1.2 Levels and Stages

Given these definitions, it may seem that quotes and splices can be used freely so long as the types align: well-typed problems don't go wrong, as the old adage says; but things are not so simple for staged programs. As well as being well-typed, a program must be well-levelled, whereby the level of variables is considered.

The concept of a level is of fundamental importance in multi-stage programming. The *level* of an expression is an integer given by the number of quotes that surround it, minus the number of splices: quotation increases the level and splicing decreases the level. Negative levels are evaluated at compile time, level 0 is evaluated at runtime and positive levels are future unevaluated stages. Unlike MetaML (Taha and Sheard, 1997), code generation in Typed Template Haskell is only supported at compile time, and therefore there is no run operation. The goal of designing a staged calculus is to ensure that the levels of the program can be evaluated in order so that an expression at a particular level can only be evaluated when all expressions it depends on at previous levels have been first evaluated.

It is also important to stress the difference between a level and a stage. A level is a semantic construct which is used during typechecking. The typing judgement for

expressions is indexed by a level to ensure that the resulting expression is well-levelled. A stage is an operational idea. In a program calculus with only expressions, the difference between a level and stage is sometimes unclear. Adding modules though for example, it is clear that a module A is evaluated completely before a module B is evaluated. Therefore it could be thought of that module A is evaluated at a stage prior to module B. On the other hand, each definition in A and B is level indexed in the same manner starting from 0, despite all the definitions in A being evaluated before those in module B. Therefore we attempt to be careful to correctly refer to levels and stages despite how informally the two ideas are often identified (Taha and Sheard, 2000).

When using Typed Template Haskell, it is common to just deal with two stages, a compile-time stage and a run-time stage. The compile-time stage contains all the statically known information about the domain, the typing rules ensure that information from the run-time stage is not needed to evaluate the compile-time stage and then the program is partially evaluated at compile-time to evaluate the compile-time fragment to a value. It is a common misconception that Typed Template Haskell only supports two stages, but there is no such restriction.[1]

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

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

$$\begin{aligned}&\mathsf{timely} :: \mathsf{Code}\ (\mathsf{Int} \to \mathsf{Int})\\&\mathsf{timely} = [\![\ \lambda \mathsf{x} \to \mathsf{x}\ ]\!]\end{aligned}$$

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

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

$$\begin{aligned}&\mathsf{hasty} :: \mathsf{Code}\ \mathsf{Int} \to \mathsf{Int}\\&\mathsf{hasty}\ \mathsf{c} = \$(\mathsf{c})\end{aligned}$$

Here, we cannot splice c without knowing the concrete representation that c will be

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

instantiated to. There is no recovery from this situation without violating the fact that lower levels must be fully evaluated before higher ones.

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

```
tardy :: Int → Code Int
tardy x = ⟦ x ⟧
```

In general, the purpose of a type system of a multi-stage language is to ensure that variables are only used at the correct level. Therefore all typing judgements are augmented with a level (Taha and Sheard, 1997) and the level at which each variable is introduced is kept track of in the environment.

Lifting In contrast to the hasty example, the situation is not hopeless for tardy variables: there are ways to support referencing previous-stage variables, which is called *cross-stage persistence* (Taha and Sheard, 1997).

One option is to interpret a variable from a previous stage using a *lifting* construct which copies the current value of the variable into a future-stage representation. As the process of lifting is akin to serialisation, this can be achieved quite easily for base types such as strings and integers, but more complex types such as functions are problematic as they can not be serialised in most languages.

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

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

```
lift :: Lift a ⇒ a → Code a
```

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

```
tardy' :: Int → Code Int
tardy' x = ⟦ $(lift x) ⟧
```

in which the reference of x occurs at the same level as it is bound. For this reason, our formal languages introduced in Section 3.3 and Section 3.4 will consider path-based persistence, but not implicit lifting, as this is easy to add separately in the same way as GHC currently implements it.

The Lift Class The Lift class is used to implement cross-stage persistence by lifting. The class is polymorphic in the runtime representation (Eisenberg and Peyton Jones, 2017) since GHC 8.10 (Theriault, 2019) which allows Lift instances for primitive types such as Int#. There are two methods lift and liftUntyped which allow lifting to the typed and untyped representations respectively.

```
class Lift (t :: TYPE r) where
  liftUntyped :: t → Q Exp
  default liftUntyped :: (r ∼ 'LiftedRep) ⇒ t → Q Exp
  liftUntyped = unTypeQ ∘ lift
  lift :: t → Code t
```

For a datatype E then lift can be implemented by using quotes, splices and recursively calling lift.

```
data E = Num Int | Add E E

instance Lift E where
  lift (Num i) = ⟦ Num $(lift i) ⟧
  lift (Add e1 e2) = ⟦ Add $(lift e1) $(lift e2) ⟧
```

Deriving Lift Due to the mechanical nature of writing Lift instances, since GHC 8.0.1 support has been available to automatically derive the instance for most user-written data types using the `DeriveLift` extension. When the extension is enabled then the Lift class can be derived automatically using the normal deriving declaration.

```
data E = Num Int | Add E E deriving Lift
```

Which will generate the typical code you would write by hand using quotations and splices as implemented manually for E above.

For the majority of primitive types such as Int, Int#, Word# and so on Lift instances are provided by the TEMPLATE-HASKELL library[2]. It is possible to define these instances

[2]TEMPLATE-HASKELL is a library distributed with GHC which provides the data types to represent Haskell expressions and other useful functions for interacting with quotations.

in a user-library without relying on special compiler support because there are suitable combinators provided by TEMPLATE-HASKELL in order to construct these primitive values. For example the Int# primitive is represented using the IntPrimL constructor.

```
data Lit = ... | IntPrimL Integer | ...
instance Lift Int# where
  lift x = unsafeTExpCoerce (liftUntyped x)
  liftUntyped x = return (LitE (IntPrimL (fromIntegral (I# x))))
```

2.1.3 Differences to the Current Implementation

Throughout this thesis we will mostly be concerned with writing simple multi-stage programs as most of the material is formal or foundational in nature. There are certain simplifications which are taken in the presentation.

In the current implementation the syntax for quotation and splicing is different. Quotation is written using double barred quotation brackets (`[|| ||]`, rather than $[\![\ ]\!]$) and splices with a double dollar sign (`$$()`, rather than $\$(\cdot)$). Also importantly the type of a quotation is not simply `Code T` but `Q (TExp T)` which represents an expression which produces a typed expression `T` in a monadic context `Q`. Therefore the power example would instead look like the following in an actual Typed Template Haskell program:

```
power :: Int -> Q (TExp Int) -> Q (TExp Int)
power 0 _ = [|| 1 ||]
power k n = [|| $$n * $$(power (k - 1) n)  ||]
```

In the next version of GHC (9.0.1) the return type of the quotation has been modified again. Firstly due to the overloaded quotation proposal (Pickering, 2019), the return type of a quotation was generalised from Q to a monad supporting the necessary effects to generate the quotation representation. With this modification the type of quotation is now:

$$[\![\ 1\ ]\!] :: \mathsf{Quote}\ m \Rightarrow m\ (\mathsf{TExp}\ \mathsf{Int})$$

Overloaded quotations will be described in a bit more detail in Section 3.7.1

Then due to Proposal 195 (Pickering, 2020) the return type of quotations was modified again so that the result is wrapped in a newtype called Code, which brings the presentation closer to what is in this thesis.

```
⟦ 1 ⟧ :: Quote m ⇒ Code m Int
```

The motivation for this change is to make it easier to write instances for the $\mathsf{Q} \circ \mathsf{TExp}$ functor composition and store quoted fragments directly in type-indexed maps.

For the majority of time we will not care about the ambient monadic context so will just write `Code Int` rather than also worrying about the monadic context used during code generation.

2.1.4 Untyped Template Haskell

Untyped Template Haskell (Sheard and Peyton Jones, 2002) follows a similar API to Typed Template Haskell but the quotations ($[| \cdot |]$) are not typechecked and the representation form is not indexed by a type. The result type of quoting an expression is `Q Exp`.

```
uquote :: Q Exp
uquote = [| 1 |]
```

It is even possible to use untyped quotations to quote unbound variables.

```
unbound :: Q Exp
unbound = [| a |]
```

Unlike Typed Template Haskell, representations can also be constructed directly using combinators. So `uquote` could instead be constructed directly using the `numE` combinator.

```
uquote :: Q Exp
uquote = numE 1
```

The final difference is that Untyped Template Haskell allows you to construct and splice in declarations, patterns and types rather than just expressions as in Typed Template Haskell.

It is uncommon to build compositional code generators using Untyped Template Haskell. The programmer has no means in order to ensure that their quotation, or generated program is correct before the metaprogram is executed. Users generally construct specific code generators for their particular domain on a case by case basis. The style of program generation is also quite different. Untyped Template

Haskell promotes intensional syntax analysis, programs should be constructed by pattern matching on the structure of an expression. Additional information about identifiers is queried by using "reification" functions. It is hard to construct program generators and easy to get wrong. The advantage of Untyped Template Haskell is its great versatility and flexibility.

There are some libraries which attempt to use Untyped Template Haskell to provide library functions which can be used by users to perform code generation but the interface is hard to use. For example GENIPLATE (Augustsson, 2018) provides combinators in order to construct traversal functions for specific data types. The signature of genUniverseBi gives little indication about what either the argument to the function is or what the type of the resulting generated expression is.

```
genUniverseBi :: Name → Q Exp
```

In fact, a documentation comment informs us that the Name should be of type $S \to [T]$ and the returned expression is also of type $S \to [T]$. Working out how to use this function and other functions in the untyped style is not very ergonomic so not many libraries provide functions in this style.

Interaction with Typed Template Haskell There are a pair of functions which can convert between the untyped and typed representations. unTypeQ converts from the typed to untyped representation and unsafeTExpCoerce from the untyped to typed representation.

```
unTypeQ :: Functor m ⇒ Code m a → m Exp
unsafeTExpCoerce :: Functor m ⇒ m Exp → Code m a
```

The unsafety of unsafeTExpCoerce is not in the sense of memory unsafety but in terms of the soundness guarantees of Typed Template Haskell. We will discuss this a lot more in Section 3.1. In particular, using an expression of type TExp a is supposed to represent a value of type a but unsafeTExpCoerce takes no precautions to ensure this is the case. The function can be used to coerce any Exp value into a type-indexed representation.

```
bad :: Code Q Char
bad = unsafeTExpCoerce [|| 1 ||]
```

The type of bad should actually be Code Q Int. Attempting to use bad will not result in a segfault but a failure to typecheck the resulting program when it is spliced.

The existence of unTypeQ and unsafeTExpCoerce is made possible by the fact that in the current implementation both typed and untyped quotations use exactly the same underlying representation. Hence unTypeQ and unsafeTExpCoerce have no computation content and just coerce a value of type Exp into onto type TExp a. In later sections when we discuss different choices for the representation then in order to support these conversion functions the interface will have to be modified because the underlying representations will be different.

The style of programming in Untyped Template Haskell is quite different to how you are supposed to program in Typed Template Haskell. In Typed Template Haskell you rely on using quotes and splices in order to construct expressions. It is unusual to want to use any of the functions from the Q monad. The style in Untyped Template Haskell is to rarely use quotation and splicing. Programs are generated usually by calling one of the reify$*$ functions to learn information about a specific name before directly generating a program using the combinators. This is at least partly due to how Untyped Template Haskell is often used to generate declarations

2.1.5 Current Implementation

For the purposes of this thesis, understanding how Typed Template Haskell is currently implemented is also important. The issues which will be focused on in later chapters are nearly all as a result of issues with the current implementation or new challenges posed by a more robust implementation strategy.

Quotation Representation When implementing a multi-stage language, one of the primary considerations is the chosen internal representation of a quotation. The approach that Untyped Template Haskell and MetaOCaml take is one based on generating code which when executed will produce an abstract syntax tree which mirrors the quoted expression. This is called the combinator approach because the functions used to build the representation are combinators which are executed to produce the syntax tree.

It is useful to keep this idea in mind, the elaboration of a quotation is a program which when run produces a runtime representation of the quoted expression.

The combinator approach makes code generation quite straightforward to think about as each construct in the internal compiler AST is translated to a call of a combinator which builds up the Template Haskell AST. For example, the typed quotation $[\![1 + 2]\!]$ is elaborated to the following program:

```
unsafeTExpCoerce (appE (appE (varE (+)) (numE 1)) (numE 2))
```

Applications are translated to appE, variables to varE, numbers to numE and so on. For typed quotations the untyped representation is first built by the combinators before being converted to the typed representation using unsafeTExpCoerce.

Another advantage of the combinator approach is that nested splices are easy to implement as the spliced expression can be directly used as an argument to the combinator. Because the representation is built by composing combinators together, the splices are inserted directly into the AST without explicitly performing substitution. For example, $[\![\$(x) + 2]\!]$ elaborates to:

```
unsafeTExpCoerce (appE (appE (varE (+)) (unTypeQ x) (numE 2)))
```

The spliced expression x is just inserted directly into the combinator calls rather than translating the variable x to varE "x".

Names The treatment of names in the elaboration process is also of interest in a language which supports quoting open terms. If we are to assume that all level errors have been corrected by inserting lifts, then a name appearing inside a quotation has two possibilities.

1. The name is bound in a normal Haskell program.
2. The name is bound inside the quotation.

In each of these two cases, a different implementation approach is needed.

Firstly, if the name is not bound inside the quotation, then it means we want to generate a program which contains precisely that unbound name without any modifications. As an example where this can arise, consider using the power function to statically compute the kth power of a number:

```
power5 :: Int → Int
power5 x = $(power 5 [| x |])
```

The usage of x is bound by x as an argument to power5. The typing rules ensure that the usage of x is correct and the intention here of the programmer is that the generated program will contain references to the variable x bound in the argument. Therefore, the precise information about the name needs to be stored in the representation so that

when the representation is converted back into the compiler's internal representation the reference is preserved.

The second case is when a name is bound inside a quotation:

$$\mathsf{qid} = [\![\ \lambda \mathsf{x} \rightarrow \mathsf{x}\ ]\!]$$

For any name which is bound inside a quotation, it is important that whenever the generating function is called that a fresh name is generated. This is why the Quote m type class contains a single method which is about generating fresh names as it is the only effect which is needed in order to elaborate a source expression into combinators. Whenever a binding construct appears inside a quotation a call to newName is inserted into the generated program so that each occurrence will be given a fresh name. qid will elaborate to something like:

$$\mathsf{newName}\ \texttt{"x"}\ \gg\!= \lambda \mathsf{x} \rightarrow \mathsf{lamE}\ [\mathsf{varP}\ \mathsf{x}]\ (\mathsf{varE}\ \mathsf{x})$$

For the simple case of qid, this dynamic renaming is not important, and you can get a long way without it but eventually there are situations where you need to dynamically rename variables to keep them distinct if a code generator is called recursively for instance (see Section 4.1.1 for an example).

During the elaboration process, the variable which will be bound to the result of newName is substituted for any occurrences of the variable inside the scope of the quotation. The result is that the generated program will have a fresh name for each time qid is called during the program generator.

Module Restriction for Top-Level Splices The module restriction is the limitation to top-level splices which states that only identifiers bound in the current splice or in another module are allowed in a top-level splice. Notably, this prevents the usage of top-level definitions from the same module being used inside a top-level splice.

$$\mathsf{qunit} = [\![\ ()\ ]\!]$$
$$\mathsf{unit} = \$(\mathsf{qunit})$$

This is a limitation due to the implementation as it has to ensure that any identifier used inside a top-level splice is compiled before the splice can be executed. For functions defined in other modules, they will already be compiled to object code, but definitions in the module currently being compiled will not be compiled themselves yet.

This limitation is one of the annoying paper cuts of Template Haskell, the program should "obviously work" according to the programmer. There are two workarounds which are employed to overcome the limitation, either inline the function `qunit` into the splice or move the definition of `qunit` into another module. For any definitions of reasonable size the inlining option is not appealing so for even small toy programs the programmer is immediately required to used at least two modules.

This limitation will be resolved in Section 4.1.5 where the new implementation strategy will make computing the necessary dependencies of top-level splices more straightforward.

Execution of Top-Level Splices The second aspect of implementing a multi-stage language is running the program generators. At the simplest level, there is no additional difficulty in executing these programs, as they are just normal Haskell programs. The main complication is that the evaluation happens at compile-time so the compiler needs to also include a complete interpreter for the Haskell language.

GHC uses the bytecode interpreter to perform this evaluation, the same evaluator as is used by GHCi. Using this interpreter causes some issues with evaluating Template Haskell programs on more unusual targets where the bytecode interpreter does not work but in general it provides an efficient and robust evaluation mechanism.

These days with the importantance of dependently typed languages it is very common for a compiler to be able to evaluate expressions in order to typecheck a program. When Template Haskell was implemented, building a simple evaluator into the compiler was less common. However, even though an internal evaluator is common. They are usually still very slow and inefficient. For this reason modern languages such as F* (Martínez et al., 2019) provide the option of evaluating with the built-in interpreter or using compiled code.

At some point in the future it is possible that Haskell will also get a similar symbolic interpreter built into the compiler but this will be a large engineering effort because of the large number of primitives that GHC implements with hard to specify semantics. For instance, a complete interpreter would need to provide an implementation of all the concurrency primitives and a large number of other very specific primops. It is clearly possible to achieve as evidenced by the Haskell-To-JavaScript compilers (GHCJS, Asterius, WebGHC) but a lot of engineering effort when there is already an interpreter which works.

2.1.6 Writing Staged Programs

The concern of this thesis is not in particular how to write staged programs so this section is only a brief refresher of some of the key insights needed in order to write in this style. There are only a couple of examples of staged programs in the thesis so it would be useful to consult (Kiselyov, 2018; Taha, 2004) for some more instruction about the necessary techniques.

Binding-Time Analysis The rules of multi-stage programming ensure that the program you construct can be evaluated level by level but getting to this point often requires care in structuring your program. Not every program can be staged, before any staging commences the programmer needs to perform their own *binding-time analysis* of their program in order to decide which arguments are statically known and which are dynamically known. For example, if you are staging a standard interpreter than you can consider the input program to be statically known but the evaluation environment to be dynamic information. Dynamic arguments are indicated with the Code type constructor.

$$\mathsf{eval} :: \mathsf{Program} \to \mathsf{Code\ Env} \to \mathsf{Code\ Result}$$

Once the programmer has performed the binding-time analysis and annotated the function correctly with the binding-time information, it is sometimes a mechanical process of inserting quotations and splices in order to construct the staged program. However, for any realistic application, it will not be this straightforward. Reasoning about binding-times takes experience to get right, and even if you are correct an unstaged program often contains static and dynamic information interacting together in ways which are forbidden in the multi-stage setting. Further program transformation is necessary in order to enforce the separation. An incorrect binding-time analysis will manifest in type errors as you attempt to stage the program. Some of these will require the additional insertion of quotes and splices but some will either require resolution by lifting or others may indicate that your original analysis was incorrect and it is impossible to stage the program as you desired.

Writing program generators in a multi-stage language is very much a type guided process. This is what distinguishes program generators written in Typed Template Haskell from Untyped Template Haskell.

The difficulty in writing a staged program is for good reason: unless the binding-time separation is enforced by the compiler then it is easy to write programs which have non-obvious interaction between static and dynamic arguments. It is in fact a great blessing

that the compiler can help us write stage-correct programs and one of the main reasons why staged programming is of value compared to untyped techniques. An automatic optimiser would also likely get stuck in the same place unless care is taken to enforce the correct binding structure, which is obviously much harder without the support of a type system.

Binding-Time Improvement In order to prove to the compiler that your binding-time analysis is correct then it is often necessary to perform some program transformations in order to remove places where dynamic and static information interact with each other. There are a number of techniques to achieve this and we will briefly describe the two most relevant methods to improve binding times, partially static data types and conversion to continuation-passing style.

Continuation-Passing Style Using continuation-passing style has long been known in the partial evaluation community to improve the binding times of programs (Consel and Danvy, 1991) and the same is true for multi-stage programs. In particular, if a code generator has a return type of $\mathsf{Code\ T}$ then the returned value of type T has become dynamic during the course of program generation. It may have always been dynamic but it also may have once been static information which was lifted from T to $\mathsf{Code\ T}$ during execution. Once a value has been made dynamic then any subsequent generation can not made use of its prior static nature. By adding a continuation argument $\mathsf{T} \rightarrow \mathsf{Code\ r}$ and modifying the return type to $\mathsf{Code\ r}$ then the caller of the function has control about how to use the static knowledge of T.

Partially-Static Data Structures A *partially-static data structure* is a data structure which has both static and dynamic properties. For example, a pair of type $(\mathsf{Code\ Int}, \mathsf{Code\ Bool})$ has the static property of being a tuple but with dynamically known values. It is often useful to convert data types into partially static structures when staging a program so that more static information is evident to the compiler. The need for such structures often arises when refactoring when a specific record contains some information which is known statically and some which is known dynamically. By using a static record, which contains one field which is dynamic, the static information can be used during the first stage to improve the generated code.

Partially static structures can also be used to partially normalise expressions before code generation. For example, a monoid with dynamic holes can be represented as:

```
data Monoid a = Empty | Sta a | Dyn (Code a) | Append (Monoid a) (Monoid a)
```

Then the interpretation function can perform the monoidal append operation at compile-time for statically known values before lifting the result.

```
eval :: (Monoid a, Lift a) ⇒ Monoid a → Code a
eval Empty = ⟦ mempty ⟧
eval (Sta a) = lift a
eval (Dyn a) = a
eval (Append (Sta a) (Sta b)) = lift (a ◇ b)
eval (Append a b) = ⟦ $(eval a) ◇ $(eval b) ⟧
```

Yallop et al. (2018) develops a more general theory of building partially static structures which perform more complicated normalisation of algebraic structures containing free variables.

In general, being aware that when staging a program it may also be necessary to modify the definition of data types involved can save you a lot of time and greatly improve the quality of the generated code. Staging is not as simple as modifying some types and adding some quotations.

2.2 Type Classes

One of the primary problems which this thesis considers is the interaction of type classes and quotations. Type classes (Wadler and Blott, 1989; Jones, 1992) are a means of overloading functions for a specific set of types. A type class definition declares which function can be overloaded, and a type class instance indicates that the function in the definition can be used at a specific type.

The Eq class defines an overloaded equality operator $(==)$.

```
class Eq a where
  (==) :: a → a → Bool
```

The definition indicates that we want to introduce a new class of types, called Eq. The definition has one method $(==)$ which can be overloaded by each instance of Eq. There is an instance of Eq defined for the Bool type:

```
instance Eq Bool where
  False == False = True
  True == True = True
  _ == _ = False
```

The Eq Bool instance provides a definition for $(==)$ at the specific type Bool and now $(==)$ can be used to compare boolean values. When $(==)$ is used at an unknown type, the use generates a constraint on the type of that variable.

```
check :: Eq a ⇒ a → a → a
check a1 a2 = if a1 == a2
                then a1
                else a2
```

The check function compares two values, using $(==)$ and if they are equal, returns the first, otherwise returns the second. The definition uses $(==)$, using $(==)$ requires the type to be an instance of Eq, this is reflected by the constraint Eq a $\Rightarrow$ in the type signature for check. It constrains the type variable a, so that it can only be instantiated to types which implement Eq.

2.2.1 Dictionary Translation

GHC compiles type classes using the dictionary translation. For each type class definition, a corresponding data type is defined which acts as the evidence for that type class. The data type has one field for each method of the type class. The evidence for the Eq class would look like the following:

```
data DEq a = DEq {dEq :: a → a → Bool}
```

An instance creates a value which is constructed using the relevant data type constructor. This is the evidence that a certain type is an instance of that type class. For the Eq Bool instance, the definition of the evidence would look as follows.

```
eqInt :: Bool → Bool → Bool
eqInt False False = True
eqInt True True   = True
eqInt _ _         = False

evEqBool :: DEq Bool
evEqBool = DEq eqInt
```

Then finally, the evidence for Eq is passed to the check function by translating the Eq a constraint into a normal argument.

```
check :: DEq a → a → a → a
check dEqEv a1 a2 = if (dEq dEqEv) a1 a2 then a1 else a2
```

When check is called at a specific type, such as Bool, then the Eq Bool evidence is passed to check.

A large part of this thesis is about how the dictionary translation interacts with quotations. Problems arise because the translation introduces new arguments and uses of those arguments, therefore the uses need to be level-consistent in order to result in a well-staged program. In the current implementation of Typed Template Haskell using type classes causes level-incorrect programs to be generated. This is discussed in detail in Section 3.2.

2.3 Conclusion

In this chapter we introduced the necessary concepts from multi-stage programming and provided a brief introduction to type classes as an example of one form of implicit evidence.

Chapter 3

A Specification for Typed Template Haskell

In this chapter we will formally specify the design of Typed Template Haskell which fixes a number of unsoundness issues present in the current design. Firstly in Section 3.1 the soundness condition will be defined before a large number of current soundness issues are identified. In Section 3.2, class constraints are considered in more detail before the main technical contribution, an idealised formalism of Typed Template Haskell with sound treatment of class constraints is presented in Sections 3.3, 3.4 and 3.5. How the proposed ideas would interact with other language features is discussed in Section 3.6. Finally, in Section 3.7, the other modifications to the Template Haskell API are considered and discussed. The technical content in this chapter is the contents of an as-of-yet unpublished paper coauthored with Andres Löh and Nicolas Wu. Other material has previously appeared in:

> Pickering, M., Wu, N., and Kiss, C. (2019a). Multi-stage programs in context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 71–84, New York, NY, USA. Association for Computing Machinery

3.1 Typed Template Haskell is Unsound

In this section we reflect on the main problem tackled by this thesis, making Typed Template Haskell sound. Firstly we will explicitly say what we mean by soundness and why it is an important property for a multi-stage language before demonstrating a

variety of different unsound examples which we would like to fix. The common cause of all the issues will be due to the choice of representation for quotations, but changing the representation leads to a variety of other interesting research directions which we will first deal with formally before moving onto discussing the realistic challenges of an implementation.

3.1.1 Soundness for a Multi-Stage Language

Soundness is the most important property of a programming language. Informally stated it is the property that a "well-typed" program can't "go wrong". However, it is up to the language designer to specify what a well-typed program is and what "go wrong" means.

For Typed Template Haskell, and multi-stage languages in general the contract states that if a code generator type checks then once executed, the generated program will also be well-typed.

This has a number of important consequences.

Firstly, the whole program can be typechecked without executing any code. It is only necessary to type check the code generator in order to determine whether the whole program will succeed. In contrast, Untyped Template Haskell requires all the top-level splices to be executed in order to determine whether a program is well-typed or not. In Untyped Template Haskell splices can perform complicated actions such as introduce new top-level definitions, define instances and so on, therefore it the result of running one splice can affect whether another part of the program type checks.

Secondly, any errors about code generation are directly reported at the quotation site rather than at some place in the generated program. It is usually quite difficult to trace back program generation errors from a generated program to the program generator. It is possible to include provenance information in the generated program but Untyped Template Haskell does not currently implement this. Usually code generators using Untyped Template Haskell are not very sophisticated or performing very complicated code manipulations due to the untyped nature of the interface. Untyped code generation can be very brittle and therefore in practice working out what is wrong in your code generator is a hard problem. However, when it is possible to build typed compositional generators using Typed Template Haskell it can be highly non-obvious how a certain piece of code was constructed. Therefore the soundness guarantee becomes even more critical in the typed setting.

Finally, there is no need to typecheck the generated program again which should speed

up the execution of meta programs, in practice, when executed Template Haskell splices can generate very large programs, which contain no type information and the type checker can spend a long time attempting to infer types for these very big expressions. If your multi-stage language is sound, then there should be no need to do this as you have already guaranteed that the program you have generated is correct by the definition of the language.

It is the soundness criteria which distinguishes Typed and Untyped Template Haskell and what makes Typed Template Haskell a more appealing way to write code generators. Unfortunately, Typed Template Haskell is unsound in a number of subtle ways to do with implicit arguments.

3.1.2 Types

The most common form of implicit argument are types. Most functional programming languages have sophisticated type inference which makes specifying the precise types of terms unnecessary. The process of elaboration makes all types explicit during type checking. If this implicit information is not taken into account when deciding how to represent and elaborate quotations then the resulting language will be unsound.

In the introduction there was already an example of unsoundness due to type information not being preserved in the representation.

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

The trim function composed of the two quoted functions does not retain the fact that the show and read combinators were monomorphised to work only with Ints.

$$\mathsf{trim} :: \mathsf{Code}\ (\mathsf{String} \to \mathsf{String})$$
$$\mathsf{trim} = [\![\ \$(\mathsf{qshow}) \circ \$(\mathsf{qread})\ ]\!]$$

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

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

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

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

This simple example is the most direct and clearest indication that the quotation representation needs to also store the proceeds of elaboration in a language where type inference alone is insufficient.

Explicit Type Variables

The obvious way to fix this issue with type variables may be to attempt to be *explicit* about the type applications in a program. For the trim' function, this would amount to explicitly specifying that the type variable a should be applied to the show function. In order to use trim' at all, you will have to explicitly apply it to a type argument as otherwise the type variable a will be ambiguous.

$$\begin{aligned}&\mathsf{trim'} :: \forall \mathsf{a}\ .\ \mathsf{Code}\ (\mathsf{String} \to \mathsf{String})\\&\mathsf{trim'} = [\![\ \mathsf{show@a} \circ \mathsf{read}\ ]\!]\end{aligned}$$

In order to create the Int normaliser, trim' is applied to the type @Int.

$$\begin{aligned}&\mathsf{qtrimInt} :: \mathsf{Code}\ (\mathsf{String} \to \mathsf{String})\\&\mathsf{qtrimInt} = \mathsf{trim'@Int}\end{aligned}$$

Now, as may be predictable, qtrimInt will fail to compile when spliced into a program.

$$\begin{aligned}&\mathsf{trimInt} :: \mathsf{String} \to \mathsf{String}\\&\mathsf{trimInt} = \$(\mathsf{qtrimInt})\end{aligned}$$

This simple solution leads to an out-of-scope type variable in the generated program:

```
Trim.hs:6:14: error:
    * The exact Name 'a' is not in scope
        Probable cause: you used a unique Template Haskell name (NameU),
        perhaps via newName, but did not bind it
        If that's it, then -ddump-splices might be useful
    * In the result of the splice:
        $qtrimInt
      To see what the splice expanded to, use -ddump-splices
      In the Template Haskell splice $$(qtrimInt)
      In the expression: $$(qtrimInt)
```

```
  |
6 | trimInt = $$(qtrimInt)
  |              ^^^^^^^^

Trim.hs:6:14: error:
    * GHC internal error: 'a' is not in scope during type checking,
      but it passed the renamer
      tcl_env of environment: [r5zz :-> Identifier[trimInt::t1,
                               TopLevelLet {} False]]
    * In the type 'a'
      In the first argument of '(.)', namely 'show @a'
      In the expression: (show @a . read)
  |
6 | trimInt = $$(qtrimInt)
  |              ^^^^^^^^
```

It seems the consequences of type variables and binding sites were not deeply considered when extending the compiler with explicit type applications (Eisenberg et al., 2016). The first two errors are about the out of scope type variable `a`, which we attempted to instantiate in the definition of `qtrimInt`. This choice did not make its way into the quotation.

The second type of error is a failure in abstraction where the generated code contains a type application and therefore GHC is requiring that we turn on the `TypeApplications` extension.

```
Trim.hs:6:14: error:
    * Illegal visible type application '@a'
        Perhaps you intended to use TypeApplications
    * In the result of the splice:
        $qtrimInt
      To see what the splice expanded to, use -ddump-splices
      In the Template Haskell splice $$(qtrimInt)
      In the expression: $$(qtrimInt)
  |
6 | trimInt = $$(qtrimInt)
  |              ^^^^^^^^
```

In a similar way to how the definition of a function is opaque at the calling site, the

generated program should be opaque at the splice site and whatever implementation choice the generator used should be invisible to the user.

Explicit type applications give us a further clue that the combination of types and quotations is not very well understood. Similar problems can be reproduced just using lexically scoped type variables (Peyton Jones and Shields, 2002) whose implementation predates the implementation of Template Haskell. The treatment of type variables is something we will discuss a lot more later in the thesis (Section 3.6.2).

3.1.3 Type Classes

The second most common form of implicit information in Haskell are type classes (Wadler and Blott, 1989). They are elaborated during type checking by the constraint solver (Vytiniotis et al., 2011).

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

```
class Show a where
  show :: a → String
showInt :: Int → String
showInt = show
```

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

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

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

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

It is in the generator where the decision about which instance needs to be used and that decision must be persisted to the generated code. What about when show is specialised to a monomorphic type? Now there is only one possible type for qshowInt.

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

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

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

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

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

Overlapping Instances and Quotation

By defining overlapping instances it is possible to make GHC select the wrong instance as the example in Figure 3.1 demonstrates.

In module A a type class Show is defined along with an instance for Show Int. In modules B1 and B2 which both depend on A (but not on each other) two overlapping instances are defined. The implementions return a string which indicates which instance was used. In B1 a general instance is defined for [a] whilst in B2 a specific instance for [Int]. Then in B1 the code value qshowList is created which quotes show at the specific type [Int] → String. show uses the instance for [a] defined in B2 and the instance for Int defined in A.

The semantics of overlapping instances (Peyton Jones et al., 1997) are that when there are two competing instances then the more specific of the two instances is selected. In using show at type [Int] in C the constraint solver should choose the instance defined in B2 as [Int] is more specific than [a]. However, since the type of qshowList does not mention anything about the class Show, its meaning should be fully determined when

```
module A where
class Show a where
  show :: a → String
instance Show Int where
  show _ = "int"
```

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

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

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

Figure 3.1: A library which demonstrates the interaction of overlapping instances and quotation.

it is quoted in B1. Specifically, qshowList should use the instance defined in B1 rather than B2. Therefore, the generated code should be show@[Int] ($dShow [a]@Int) when qshowList is spliced into C.

By evaluating main it can be observed the instance selected when running qshowList is in fact the instance from B2 which should be impossible because the splice should have used the instance in B1. This is a symptom of the fact that the elaborated type information is not serialised. In Section 3.2 we will consider more closely how to properly treat constraints in a multi-stage language but for now it suffices to observe that by not storing the result of elaboration in the quotation has led to unsoundness.

3.1.4 Implicit Parameters

Implicit parameters (Lewis et al., 2000) are an extension to Haskell which allow arguments to be passed implicitly to functions.

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

$$\mathsf{sort} :: (?cmp :: \mathsf{a} \to \mathsf{a} \to \mathsf{Ordering}) \Rightarrow [\mathsf{a}] \to [\mathsf{a}]$$

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

$$\mathsf{sort} = \mathsf{sortBy}\ ?cmp$$

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

$$\begin{aligned}&\mathsf{sortInt} :: [\mathsf{Int}] \to [\mathsf{Int}]\\&\mathsf{sortInt} = \textbf{let}\ ?cmp = \mathsf{compare}\ \textbf{in}\ \mathsf{sort}\end{aligned}$$

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

In sortInt, the type of sort must be [Int] → [Int] and thus the type of the quote is Code ([Int] → [Int]).

$$\begin{aligned}&\mathsf{qsortInt} :: \mathsf{Code}\ ([\mathsf{Int}] \to [\mathsf{Int}])\\&\mathsf{qsortInt} = \textbf{let}\ ?cmp = \mathsf{compare}\ \textbf{in}\ [\![\ \mathsf{sort}\ ]\!]\end{aligned}$$

However things go badly wrong when qsortInt is spliced.

$$\mathsf{sortInt'} = \$(\mathsf{qsortInt})$$

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

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

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

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

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

HasCallStack constraints

Implicit parameters on their own may seem like an esoteric concern but the now increasingly popular HasCallStack constraint is implemented in terms of implicit parameters. In particular the error function now includes an HasCallStack constraint.

```
error :: HasCallStack ⇒ String → a
```

The compiler automatically solves HasCallStack constraints by passing the current call stack to error so that it can be printed along with the exception. In particular, the location where the call to error originates from will be appended to the callstack so the user can quickly identify which call to error caused the runtime exception.

Therefore, the question about how implicit parameters should interact with multiple levels arises when you use a function which contains a call to error or another function with one of these constraints. It has not been made precise how HasCallStack and quotations should interact. There are two plausible options:

1. Use implicit cross-stage persistence and insert a the CallStack from where the generator was called. For example, if you write ⟦ error ⟧ then when error is called in the generated program it would report the source location of the quotation.
2. Only solve HasCallStack constraints at level 1. Reject any usage of functions which require callstacks in generated programs.

The advantage of the first suggestion is that more programs are typeable and accepted by the compiler, the disadvantage being that the error location may be different to what you expect. In general it seems like a failure in abstraction to want to write a program generator which calls functions which may fail at runtime and produce errors pointing into the program generator.

This discussion would take us further afield into what the intended and correct use of HasCallStack is supposed to be but at least is a realistic example where implicit parameters need to be taken into account when writing quotations.

What currently happens is that the HasCallStack constraint is solved again at the splice site!

```
module HCS where
hcs = ⟦ error "error" ⟧
module Main where
```

```
import HCS
main = $(hcs)
```

When the executable is run, the error position is the location of the splice in the final program.

```
HCS_A: error
CallStack (from HasCallStack):
  error, called at HCS_A.hs:6:11 in main:Main
```

The error location is reported at the position of the top-level splice in the main module rather than at the site of the quotation in the HSC module. As a result, tracking down which exact call to error caused the runtime crash may be quite tricky.

3.1.5 Runtime Representation Polymorphism Checks

Other type system checks are not implemented in the type checker and therefore are subverted in other ways to the prior examples.

The type system checks to enforce the restrictions imposed by representation polymorphism (Eisenberg and Peyton Jones, 2017) are only checked during the elaboration stage of the compiler. The runtime representation polymorphism extension forbids any variable to have a runtime representational polymorphic type.

Typed quotations are only typechecked before being converted into their untyped representation. The elaborator is not called on the contents of a quotation until it is spliced into a program. Therefore, expressions which would usually fail the runtime representation polymorphism checks are permitted to appear inside typed quotations. If the runtime representation variables are not instantiated to monotypes when the generator is executed, the resulting program will contain an expression which is rejected by the subsequent checks.

As a specific example, consider the function bad, taken from the GHC User Manual:

```
bad :: ∀r1 :: RuntimeRep (r2 :: RuntimeRep)
  (a :: TYPE r1) (b :: TYPE r2).
  (a → b) → a → b
bad f x = f x
```

bad fails the runtime representation polymorphism checks because the kind of type variable a is of unknown representation. Quoting bad is however not rejected by the compiler:

```
qbad :: ∀r1 :: RuntimeRep (r2 :: RuntimeRep)
  (a :: TYPE r1) (b :: TYPE r2).
  Code ((a → b) → a → b)
qbad = ⟦ λf x → f x ⟧
```

Therefore, qbad can be spliced at a polymorphic type, and unsound will subsequently fail the runtime representation polymorphism check. Our program generator qbad type checked but subsequently failed at splice time with a type error.

```
unsound :: ∀r1 :: RuntimeRep (r2 :: RuntimeRep)
  (a :: TYPE r1) (b :: TYPE r2).
  ((a → b) → a → b)
unsound = $(qbad)
```

Of course, if the kind variable r1 is instantiated concretely by the time the program is generated then the runtime representation polymorphism checks no longer fail. This technique can be used as a workaround to define functions which don't obey the restrictions but will be used at a monomorphic type at the use site.

```
workaround :: Int
workaround = $(qbad) id 5
```

Rejecting a runtime representation polymorphic code generator is not the final word and would be a conservative position to take. It is not surprising that there is a more interesting relationship between the two concepts to discover due to the similarity with C++ template metaprogramming and C#/.Net compilation of polymorphism by monomorphisation. It could be possible to extend the language with a quantifier which ensures that certain type variables were monomorphically instantiated by the time the generator was executed. This would allow the definition of qbad to be written with a "monomorphic quantification" over the r1 variable and prevent using qbad in the polymorphic unsound function.

3.1.6 Plugins

Programs which rely on source plugins will also behave unexpectedly with Typed Template Haskell. A source plugin (Pickering et al., 2019b) provides several different

extension points of the library writer to inject custom passes into the compilation pipeline. For example, one extension point is to perform an action after parsing, another after typechecking. In particular, the extension points which run before the end of the name resolution phase can be used with Typed Template Haskell but any extension point which runs after the renaming phase will suffer the same problems as the other unsoundness issues presented in this section.

There are also similar problems with constraint solver plugins (Gundry, 2015) which add custom constraint solving rules when they are enabled in a module. For example, the GHC-TYPELITS-EXTRA plugin adds rules for solving constraints involving arithmetic constraints. If the plugin is required to solve some constraints generated by a quoted program then unless this plugin is also enabled in the module where the generated program is spliced then the constraint solver will fail to solve these constraints. The solution, as always, is to also persist the evidence of the constraint solver in the body of the quotation.

3.1.7 Overloaded Syntax

Overloaded syntax also doesn't interact correctly with typed quotations. When `RebindableSyntax` is turned on, an **if** expression is interpreted in terms of the current ifThenElse :: Bool → a → a → b function in scope. So, when you turn on rebindable syntax and quote an expression you would expect that the choice about ifThenElse function is preserved to the splice site as the type of the expression depends on the ifThenElse function which is in-scope at the quotation site. In the following example, the **if** function is overloaded to just concatenate together the two branches of the conditional. Therefore the type of **if** True **then** `'a'` **else** `'b'` is String.

```
module OLS2 where

import Prelude (Bool (..), String, Char)

ifThenElse :: Bool → Char → Char → String
ifThenElse _a b c = [b, c]

ols :: Code String
ols = ⟦ if True then 'a' else 'b' ⟧

module OLS where

main :: IO ()
main = putStrLn $(ols)
```

The value ols has type Code String but when used in a top-level splice the usage of

rebindable syntax has been forgotten and the **if** syntax is interpreted as normal, which leads to a type error.

```
OLS.hs:7:11: error:
    * Couldn't match type 'Char' with '[Char]'
      Expected type: String
        Actual type: Char
    * In the expression: 'a'
      In the expression: if True then 'a' else 'b'
      In the result of the splice:
        $ols
      To see what the splice expanded to, use -ddump-splices
  |
7 | main = putStrLn $$(ols)
  |
```

The choice about which version of ifThenElse to use is not even a type-directed decision! Therefore it seems more like an oversight in the implementation rather than a cause of unsoundness rooted in the representation type.

3.1.8 Untyped Template Haskell is Sound

The statement of soundness for Untyped Template Haskell is different to Typed Template Haskell. In the untyped setting the statement is instead relaxed to extend the definition of typechecking to also include typechecking the generated programs after being inserted by a splice (Sheard and Peyton Jones, 2002). Therefore the program can't "go wrong" at the final runtime but it can certainly "go wrong" during the compile time execution stage if the generated program fails to typecheck. Practically speaking this also provides quite a different experience to programming in Haskell to normal, it is quite often that your program typechecks but will fail when run on a specific user input at compile time. Typed Template Haskell moves the failure one stage earlier so the code generator will fail to typecheck and indicate to the library author there may be a bug in their program.

3.1.9 Intermediate Summary

Now it is clear that Typed Template Haskell does not interact correctly with a very large number of other features implemented in GHC the rest of this thesis will mostly be about correcting these soundness issues and developing a solid theoretical foundation for the combination of staged programming and a language with an elaboration procedure. There are two distinct phases to the development. The first is to realise that in order to fix the problem that the representation of a term needs to contain more information than it currently does. Secondly, the result of elaboration during typechecking also needs to be preserved so that no decisions about elaboration are needed at the splice site. As soon as you add more information into a quotation then a variety of other issues become evident, the location of any bound evidence suddenly becomes important which has theoretical and practical consequences. For example, types, type class evidence, implicit parameters and so on introduce new variables into the quotation and so the binding site for these variables needs to be level-correct to ensure that the generated program has no unbound variables in it.

3.2 Staging with Type Classes and Polymorphism

The examples in Section 2.1.2 were simple demonstrations of the importance of considering levels in a well-staged program. This section discusses more complicated cases which involve class constraints, top-level splices, instance definitions and type variables. It is here where our proposed version of Typed Template Haskell deviates from GHC's current implementation. Therefore, for each example we will report how the program *should* behave and contrast it with how the current implementation in GHC behaves.

3.2.1 Constraints

Constraints introduced by type classes have the potential to cause level errors: type classes are implemented by passing dictionaries as evidence, and the implicit use of these dictionaries must adhere to the level restrictions discussed in Section 2.1.2. This requires a careful treatment of the interaction between constraints and staging, which GHC currently does not handle correctly.

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

$$\mathsf{c}_1 :: \mathsf{Show\ a} \Rightarrow \mathsf{Code\ (a \rightarrow String)}$$
$$\mathsf{c}_1 = [\![\ \mathsf{show}\ ]\!]$$

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

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

$$\mathsf{d}_1 :: \mathsf{ShowDict\ a} \rightarrow \mathsf{Code\ (a \rightarrow String)}$$
$$\mathsf{d}_1\ \mathsf{dShow} = [\![\ \mathsf{show\ dShow}\ ]\!]$$

Now this reveals a subtle problem: naively elaborating without considering the *levels of constraints* has introduced a cross-stage reference where the dictionary variable dShow is introduced at level 0 but used at level 1. As we have learned in Section 2.1.2, one remedy of this situation is to try to make dShow cross-stage persistent by lifting. However, in general, lifting of dictionaries is not straightforward to implement. Recall that lifting in GHC is restricted to instances of the Lift class which excludes functions – but type class dictionaries are nearly always a record of functions, so automatic lifting given the current implementation is not possible.

Another argument could be that the global type class coherence condition must ensure that the instances available at different stages must be the same, but the example in Section 3.1.3 demonstrates that when overlapping instances are defined the coherence condition no longer holds. Therefore, it can't be relied upon to select the same instance at different stages.

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

$$\mathsf{c}_1' :: \mathsf{CodeC\ (Show\ a)} \Rightarrow \mathsf{Code\ (a \rightarrow String)}$$
$$\mathsf{c}_1' = [\![\ \mathsf{show}\ ]\!]$$

The CodeC (Show a) constraint is introduced at level 0 but indicates that the Show a constraint will be available to be used at level 1. Therefore the Show a constraint can be

used to satisfy the show method used inside the quotation. The corresponding elaborated version is similar to the following:

$$\mathsf{d}'_1 :: \mathsf{Code}\ (\mathsf{ShowDict\ a}) \to \mathsf{Code}\ (\mathsf{a} \to \mathsf{String})$$
$$\mathsf{d}'_1\ \mathsf{cdShow} = [\![\ \mathsf{show}\ \$(\mathsf{cdShow})\ ]\!]$$

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

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

$$\mathsf{c}_2 :: \mathsf{Code}\ (\mathsf{Int} \to \mathsf{String})$$
$$\mathsf{c}_2 = [\![\ \mathsf{show}\ ]\!]$$

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

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

3.2.2 Top-level Splices

A splice that appears in a definition at the top-level scope of a module[1] introduces new scoping challenges because it can potentially require class constraints to be used at levels prior to the ones where they are introduced.

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

Example TS1. In the following example there are two modules. Module A defines the Lift class and contains the definition of a global instance Lift Int. The lift function from

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

this instance is used in module B in the definition of ts_1, which is a top-level definition. The reference to lift occurs inside a top-level splice, thus at level -1, and so the Lift Int instance is needed at compile time.

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

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

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

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

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

```
data A = A
ts2 :: Lift A ⇒ A
ts2 = $(lift A)
```

We assume lift is imported from another module as in Example TS1 and therefore cross-stage persistent. However, the problem is that the dictionary that will be used to implement the Lift A constraint will not be known until runtime, when it will be passed to ts_2. Therefore the definition of Lift A is not available to be used at compile-time in the top-level splice, and GHC correctly rejects this program. It is evident why this program should be rejected considering the dictionary translation, in a similar vein to C1. The elaboration would amount to a future-stage reference inside the splice.

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

```
ts3 :: Ord a ⇒ a → a → Ordering
ts3 = $([| compare |])
```

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

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

3.2.3 The Power Function Revisited

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

```
power5 :: Int → Int            -- Option M
power5 :: Num a ⇒ a → a        -- Option P
power5 n = $(power 5 [| n |])
```

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

```
power :: Num a ⇒ Int → Code a → Code a
```

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

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

```
power :: CodeC (Num a) ⇒ Int → Code a → Code a
```

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

```
power5' :: NumDict a → a → a
power5' dNum n = $(power [|| dNum ||] 5 [|| n ||])
```

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

3.2.4 Instance Definitions

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

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

```
data Stream = Cons {hd :: Int, tl :: Stream}
instance Eq Stream where
  s1 == s2 = hd s1 == hd s2 && tl s1 == tl s2
```

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

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

```
instance Eq Stream where
  (==) = $([|| λs1 s2 → hd s1 == hd s2 && tl s1 == tl s2 ||])
```

This program should be accepted and equivalent to I1, because in general, $\$([\![\mathsf{e}]\!]) = \mathsf{e}$. However, it is presently rejected by GHC, with the claim that the recursive use of (==) for comparing the tails is not stage-correct. The program should be accepted as the instance is introduced and used at the same-level.

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

```
eqStream :: CodeC (Eq Stream) ⇒ Code (Stream → Stream → Bool)
eqStream = ⟦ λs1 s2 → hd s1 == hd s2 && tl s1 == tl s2 ⟧
```

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

```
instance Eq Stream where
  (==) = $(eqStream)
```

It is important that the definition of eqStream uses the new constraint form CodeC (Eq Stream) so that the definition of eqStream is well-staged. As Eq Stream is the instance which is currently being defined, the Eq Stream constraint is introduced at level 0 when typing the instance definition. Therefore, in a top-level splice the local constraint can only be used by functions which require a CodeC (Eq Stream) constraint. This example is similar to power in Section 3.2.3 but the local constraint is introduced implicitly by the instance definition.

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

3.2.5 Type Variables

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

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

```
list :: ∀a.[Code a] → Code [a]
list []       = ⟦ [] @a ⟧
list (x : xs) = ⟦ (:) @a $(x) $(list @a xs) ⟧
```

The type variable a is bound at level 0 and used both at at level 1 (in its application to the type constructors), and at level 0 (in its recursive application to the list function). If

we took the lead from values then this program would be rejected by the typechecker. However, because of the in-built phase distinction, evaluation cannot get stuck on an unknown type variable as all the types will be fixed in the program before execution starts. We just need to make sure that the substitution operation can substitute a type inside a quotation.

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

3.2.6 Implicit Parameters

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

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

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

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

Then qadd will elaborate to a function similar to:

$$\mathsf{qadd'} :: \mathsf{LiftDict\ a} \to \mathsf{Code\ (Num\ a)} \to \mathsf{Code\ (a \to a)}$$
$$\mathsf{qadd'\ ld\ nd} = \textbf{let}\ x = 1\ \textbf{in}\ [\![\ \mathsf{add}\ \$(\mathsf{nd})\ \$(\mathsf{lift\ ld}\ x)\ ]\!]$$

3.2.7 Intermediate Summary

This section has uncovered the fact that the treatment of constraints is rather arbitrary in the current implementation of GHC. The examples we discussed have been motivated by

three different ad-hoc restrictions that the current implementation exhibits. Firstly, prior stage constraint references (Section 3.2.1) are permitted without any checks. Secondly, top-level splices (Section 3.2.2) are typechecked in an environment isolated from the local constraint environment which is an overly conservative restriction. Thirdly, in an instance definition (Section 3.2.4), any reference to the instance currently being defined inside a top-level splice is rejected even if it is level-correct. Additionally we showed that although constraints have a role to play in understanding staging, type variables need no special treatment beyond the correct use of type application (Section 3.2.5) and implicit parameters can use the same lifting idea as normal values (Section 3.2.6).

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

The problems observed so far are the result of interactions between class constraints and staging. Our goal now will be to develop a formal calculus to model and resolve these problems. The approach will be to introduce a source language that includes type constraints in addition to quotes and splices (Section 3.3), and a core language that is a variant of the explicit polymorphic lambda calculus with multi-stage constructs (Section 3.4). We then show how the source language can be translated into a core language where constraints have been elaborated away into a representation form for a quotation that is typed and where all contextual type information is saved (Section 3.5).

3.3 Source Language

The source language we introduce has been designed to incorporate the essential features of a language with metaprogramming and qualified types that is able to correctly handle the situations discussed in Section 3.2. We try to stay faithful to GHC's current implementation of Typed Template Haskell where possible, with the addition of the quoted constraint form $\mathsf{Code}\mathsf{C}$ for the reasons discussed in Section 3.2. The key features of this language are:

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

Figure 3.2: Source Syntax

1. Values *and constraints* are always indexed by the level at which they are introduced to ensure that they are well-staged.

2. The constraint form CodeC indicates that a constraint is available at the next stage.

3. Types (including type variables) are not level indexed and can be used at any level.

4. There is no **run** operation in the language. All evaluation of code is performed at compile-time by mean of top-level splices which imply the existence of negative levels.

5. Path-based cross-stage persistence is supported.

These features have been carefully chosen to allow a specification of Typed Template Haskell that addresses all the shortcomings identified in Section 3.2 while staying close to the general flavour of meta-programming that is enabled in its current implementation in GHC.

3.3.1 Syntax

The syntax of the source language (Figure 3.2) models programs with type classes and metaprogramming features. This syntax closely follows the models of languages with

type classes of Bottu et al. (2017) and Chakravarty et al. (2005), but with the addition of multi-stage features and without equality or quantified constraints.

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

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

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

The expression language e is a standard λ-calculus with the addition of the two multi-stage constructs, quotation $[\![\ \mathsf{e} \ ]\!]$ and splicing $\$(\mathsf{e})$ which can be used to separate a program into stages. The language omits local definitions but we anticipate no complications introducing them to the language.

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

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

The environment Γ is used for locally introduced information, including value variables $x : (\tau, \mathsf{n})$, type variables a, and local type class axioms (C, n). The environment keeps track of the (integer) level n that value and constraint variables are introduced at; the typing rules ensure that the variables are only used at the current level.

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

3.3.2 Typing Rules

The typing rules proceed in a predictable way for anyone familiar with qualified type systems. The difference is that the source expression typing judgement (Figure 3.3) and constraint entailment judgement (Figure 3.4) are now indexed by a level. Furthermore, since these rules are the basis for elaboration into the core language, they have been combined with the elaboration rules, as highlighted by a grey box whose contents can be safely ignored until the discussion of elaboration (Section 3.5).

The typing rules also refer to type and constraint formation rules (Figures 3.9 and 3.10), which check just that type variables are well-scoped (our language does not include an advanced kind system).

Levels

A large part of the rules are indexed by their (integer) level. It is standard to index the expression judgement at a specific level (for example Taha and Sheard (1997) and many others) but less standard to index the constraint entailment judgement. The purpose of the level index is to ensure that expression variables and constraints can only be used at the level they are introduced. The result is a program which is separated into stages in such a way that when imbued with operational semantics, the fragments at level n will be evaluated before the fragments at level $\mathsf{n} + 1$.

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

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

- A variable can be introduced at any specific level n by a λ-abstraction (E_ABS) and used at precisely that level (E_VAR).
- Variables introduced by top-level definitions can be used at any level (E_VAR_TOPLEVEL).
- Type variables can be introduced (E_TABS) and used at any level (E_TAPP).
- Constraints can be introduced at any level (E_C_ABS), but in practice, due to the system being predicative, this level will always be 0 (see Section 3.3.2). Constraints are also introduced into the program logic by instance declarations.

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

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

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

$$\frac{\mathsf{P};\Gamma \vdash^{\mathsf{n}} \mathsf{e}_1 : \tau_1 \to \tau_2 \rightsquigarrow \mathsf{t}_1 \mid \mathsf{TSP}_1 \qquad \mathsf{P};\Gamma \vdash^{\mathsf{n}} \mathsf{e}_2 : \tau_1 \rightsquigarrow \mathsf{t}_2 \mid \mathsf{TSP}_2}{\mathsf{P};\Gamma \vdash^{\mathsf{n}} \mathsf{e}_1\ \mathsf{e}_2 : \tau_2 \rightsquigarrow \mathsf{t}_1\ \mathsf{t}_2 \mid \mathsf{TSP}_1 \cup \mathsf{TSP}_2}\ \text{E\_APP}$$

$$\frac{\mathsf{a} \notin \Gamma \qquad \mathsf{P};\Gamma, \mathsf{a} \vdash^{\mathsf{n}} \mathsf{e} : \sigma \rightsquigarrow \mathsf{t} \mid \mathsf{TSP}}{\mathsf{P};\Gamma \vdash^{\mathsf{n}} \mathsf{e} : \forall \mathsf{a}.\sigma \rightsquigarrow \Lambda \mathsf{a}.\mathsf{t} \mid \mathsf{TSP}}\ \text{E\_TABS}$$

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

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

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

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

$$\frac{\mathsf{P};\Gamma \vdash^{\mathsf{n}} \mathsf{e} : \mathsf{C} \Rightarrow \rho \rightsquigarrow \mathsf{t}_1 \mid \mathsf{TSP}_1 \qquad \mathsf{P};\Gamma \vDash^{\mathsf{n}} \mathsf{C} \rightsquigarrow \mathsf{t}_2 \mid \mathsf{TSP}_2}{\mathsf{P};\Gamma \vdash^{\mathsf{n}} \mathsf{e} : \rho \rightsquigarrow \mathsf{t}_1\ \mathsf{t}_2 \mid \mathsf{TSP}_1 \cup \mathsf{TSP}_2}\ \text{E\_C\_APP}$$

Figure 3.3: Source Expression Typing with Elaboration Semantics

> Constraints are appropriately eliminated by application (E_C_APP). The entailment relation (Figure 3.4) ensures that a constraint is available at the current stage.

Note that we do not allow implicit lifting as discussed in Section 2.1.2 and implemented in GHC. Explicit lifting via a Lift class can easily be expressed in this language, and implicit lifting, if desired, can be orthogonally added as an additional elaboration step.

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

Level of Constraints

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

The new constraint form $\mathsf{CodeC\ C}$ can be used to allow local constraints to be used at positive levels in E_C_APP. The rules C_INCR and C_DECR in constraint entailment (Figure 3.4) can convert from $\mathsf{CodeC\ C}$ and C and vice versa. During elaboration, these rules will insert splices and quotes around the dictionary arguments as needed.

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

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

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

From this we can use E_VAR_TOPLEVEL to conclude that

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

Because $[\![\ \mathsf{show}\ ]\!]$ must have type $\mathsf{Code}\ \tau$ for some monotype τ, we cannot apply E_QUOTE yet, but must first apply E_TAPP and E_C_APP. We can conclude

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

if the entailment

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

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

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

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

Figure 3.4: Source Constraints with Elaboration

$$\mathsf{P};\Gamma \vDash^{1} \mathsf{Show\ a}$$

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

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

This in turn follows from C_LOCAL if we assume that Γ contains $\mathsf{CodeC}\ (\mathsf{Show\ a})$:

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

At this point, we know that

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

and by applying E_C_ABS and E_TABS, we obtain

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

as desired.

Program Typing

A program (Figure 3.8) is a sequence of value, class and instances declarations followed by an expression. The declaration forms extend the program theory which is used

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

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

Figure 3.5: Source Definition Typing with Elaboration

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

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

Figure 3.6: Source Class Typing with Elaboration

to typecheck subsequent definitions. Value definitions extend the list of top-level definitions available at all stages. The judgement makes it clear that the top-level of the program is level 0 and that each expression is checked in an empty local environment.

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

Notice that before the type class method is checked, the local environment Γ is extended with an axiom for the instance we are currently checking. This is important to allow recursive definitions (Example I1) to be defined in a natural fashion. The constraint is introduced at level 0, the top-level of the program. It is important to introduce the constraint to the local environment rather than program theory because this constraint should not be available at negative levels. If the constraint was introduced to the program theory, then it could be used at negative levels, which would amount to attempting to use the instance in order to affect the definition of said instance. This nuance in the typing rules explains why Example I2 is accepted and the necessity of the CodeC constraint in Example I3.

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

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

Figure 3.7: Source Instance Typing with Elaboration

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

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

$$\frac{\mathsf{P}_2 \vdash_{\mathsf{pgm}} \mathsf{pgm} : \sigma \rightsquigarrow \mathsf{pgm} \qquad \mathsf{pgm} \mid \mathsf{P}_1 \vdash_{\mathsf{def}} \mathsf{def} : \mathsf{P}_2 \rightsquigarrow \mathsf{pgm}_1}{\mathsf{P}_1 \vdash_{\mathsf{pgm}} \mathsf{def}; \mathsf{pgm} : \sigma \rightsquigarrow \mathsf{pgm}_1}\ \textsc{P\_Def}$$

$$\frac{\mathsf{P}_2 \vdash_{\mathsf{pgm}} \mathsf{pgm} : \sigma \rightsquigarrow \mathsf{pgm} \qquad \mathsf{pgm} \mid \mathsf{P}_1 \vdash_{\mathsf{cls}} \mathsf{cls} : \mathsf{P}_2 \rightsquigarrow \mathsf{pgm}_2}{\mathsf{P}_1 \vdash_{\mathsf{pgm}} \mathsf{cls}; \mathsf{pgm} : \sigma \rightsquigarrow \mathsf{pgm}_1}\ \textsc{P\_Cls}$$

$$\frac{\mathsf{P}_2 \vdash_{\mathsf{pgm}} \mathsf{pgm} : \sigma \rightsquigarrow \mathsf{pgm} \qquad \mathsf{pgm} \mid \mathsf{P}_1 \vdash_{\mathsf{inst}} \mathsf{inst} : \mathsf{P}_2 \rightsquigarrow \mathsf{pgm}_1}{\mathsf{P}_1 \vdash_{\mathsf{pgm}} \mathsf{inst}; \mathsf{pgm} : \sigma \rightsquigarrow \mathsf{pgm}_1}\ \textsc{P\_Inst}$$

Figure 3.8: Source Program Typing with Elaboration

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

$$\frac{\mathsf{a} \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} \mathsf{a} \rightsquigarrow \mathsf{a}}\ \text{T\_VAR} \qquad \frac{}{\Gamma \vdash_{\mathsf{ty}} \mathsf{H} \rightsquigarrow \mathsf{H}}\ \text{T\_CONST}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \rightsquigarrow \tau_1' \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2 \rightsquigarrow \tau_2'}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \to \tau_2 \rightsquigarrow \tau_1' \to \tau_2'}\ \text{T\_ARROW}$$

$$\frac{\Gamma \vdash_{\mathsf{ct}} \mathsf{C} \rightsquigarrow \tau_1 \qquad \Gamma \vdash_{\mathsf{ty}} \rho \rightsquigarrow \tau_2}{\Gamma \vdash_{\mathsf{ty}} \mathsf{C} \Rightarrow \rho \rightsquigarrow \tau_1 \to \tau_2}\ \text{T\_CARROW} \qquad \frac{\Gamma, \mathsf{a} \vdash_{\mathsf{ty}} \sigma \rightsquigarrow \tau}{\Gamma \vdash_{\mathsf{ty}} \forall \mathsf{a}.\sigma \rightsquigarrow \forall \mathsf{a}.\tau}\ \text{T\_FOR\_ALL}$$

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

Figure 3.9: Source Type Formation with Elaboration

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

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

$$\frac{\Gamma \vdash_{\mathsf{ct}} \mathsf{C} \rightsquigarrow \tau}{\Gamma \vdash_{\mathsf{ct}} \mathsf{CodeC}\ \mathsf{C} \rightsquigarrow \mathsf{Code}\ \tau}\ \text{C\_CODEC}$$

Figure 3.10: Source Constraint Formation with Elaboration

3.4 The Core Language

In this section we describe an explicitly typed core language which is suitable as a compilation target for the declarative source language we described in the Section 3.3.

3.4.1 Syntax

The syntax for the core language is presented in Figure 3.11. It is a variant of the explicit polymorphic lambda calculus with multi-stage constructs, top-level definitions and top-level splice definitions.

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

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

A top-level splice definition is used to support elaborating from negative levels in the source language. These top level declarations are of the form $\mathbf{spdef}\ \Delta \vdash^{\mathsf{n}} \mathsf{sp} : \tau = \mathsf{e}$ which indicates that the expression e will have type $\mathsf{Code}\ \tau$ at level n in environment Δ. Top-level splice definitions also explicitly record the level of the original splice so that the declaration can be typechecked at the correct level. The level index is not necessary for the quotation splice environments because the level of the whole environment is fixed by the level at which the quotation appears.

3.4.2 Typing Rules

The expression typing rules for the core language are for the most part the same as those in the source language. The rules that differ are shown in Figure 3.12.

The splice environment typing rules are given in Figure 3.13. A splice environment is well-typed if each of its definitions are well-typed. We check that the body of each

definition has type $\mathsf{Code}\ \tau$ in an environment extended by Δ. In the E_QUOTE rule, the body of the quotation is checked in an environment Γ extended with the contents of the splice environment. A splice variable sp is associated with an environment Δ, a type τ and a level n in Γ. When a splice variable is used, the E_SPLICE_VAR rule ensures that the claimed local environment aligns with the actual environment, the type of the variable is correct and the level matches the surrounding context.

Top-level splice definitions are typed in a similar manner in Figure 3.14. The body of the definition is checked to have type $\mathsf{Code}\ \tau$ at level n in environment Γ, the program theory is then extended with a splice variable $\mathsf{sp} : (\Gamma, \tau)$ which can be used in the remainder of the program.

3.4.3 Dynamic Semantics

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

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

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

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

Using splice environments and top-level splice definitions is reminiscent of the approach taken in logically inspired languages by Bierman and de Paiva (2000), Nanevski (2002) and Davies and Pfenning (2001).

3.4.4 Module Restriction

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

This reflects the situation in a compiler such as GHC which supports separate compilation. When compiling a program which uses multiple modules, one module may contain a top-level splice which is then used inside another top-level splice in a different module. Although syntactically both top-level splices occur at level -1, they effectively occur at different stages. In the same way as different modules occur at different stages due to separate compilation, in our calculus, each top-level definition can be considered to be evaluated at a new stage.

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

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

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

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

Figure 3.11: Core Language Syntax

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

$$\frac{\Delta \vdash \mathsf{sp} : (\tau, \mathsf{n}) \in \Gamma \qquad \Delta \subseteq \Gamma}{\mathsf{P}; \Gamma \vdash^{\mathsf{n}} \mathsf{sp} : \tau}\ \text{E\_SPLICE\_VAR}$$

$$\frac{\mathsf{sp} : (\Delta, \tau) \in \mathsf{P} \qquad \Delta \subseteq \Gamma}{\mathsf{P}; \Gamma \vdash^{\mathsf{n}} \mathsf{sp} : \tau}\ \text{E\_TOP\_SPLICE\_VAR} \qquad \frac{\mathsf{P}; \Gamma, \mathsf{a} \vdash^{\mathsf{n}} \mathsf{e} : \tau}{\mathsf{P}; \Gamma \vdash^{\mathsf{n}} \Lambda \mathsf{a} . \mathsf{e} : \forall \mathsf{a} . \tau}\ \text{E\_TABS}$$

$$\frac{\mathsf{P}; \Gamma \vdash^{\mathsf{n}} \mathsf{e} : \forall \mathsf{a} . \tau_2}{\mathsf{P}; \Gamma \vdash^{\mathsf{n}} \mathsf{e}\ \langle \tau_1 \rangle : \tau_2 [\tau_1 / \mathsf{a}]}\ \text{E\_TAPP}$$

$$\frac{\mathsf{P}; \Gamma, \overline{\mathsf{sp_i} : (\Delta_\mathsf{i}, \tau_\mathsf{i}, \mathsf{n}+1)} \vdash^{\mathsf{n}+1} \mathsf{e} : \tau \qquad \mathsf{P}; \Gamma \vdash^{\mathsf{n}}_{\mathsf{SP}} \mathsf{SP} \qquad \mathsf{SP} = \overline{\Delta_\mathsf{i} \vdash \mathsf{sp_i} : \tau_\mathsf{i} = \mathsf{e_i}}}{\mathsf{P}; \Gamma \vdash^{\mathsf{n}} [\![\mathsf{e}]\!]_{\mathsf{SP}} : \mathsf{Code}\ \tau}\ \text{E\_QUOTE}$$

Figure 3.12: Core Expression Typing

$$\boxed{\mathsf{P}; \Gamma \vdash^{\mathsf{n}}_{\mathsf{SP}} \mathsf{SP}}$$

$$\frac{}{\mathsf{P}; \Gamma \vdash^{\mathsf{n}}_{\mathsf{SP}} \bullet}\ \text{SP\_EMPTY} \qquad \frac{\mathsf{P}; \Gamma \vdash^{\mathsf{n}}_{\mathsf{SP}} \mathsf{SP} \qquad \mathsf{P}; \Gamma, \Delta \vdash^{\mathsf{n}} \mathsf{e} : \mathsf{Code}\ \tau}{\mathsf{P}; \Gamma \vdash^{\mathsf{n}}_{\mathsf{SP}} \mathsf{SP}, \Delta \vdash \mathsf{sp} : \tau = \mathsf{e}}\ \text{SP\_CONS}$$

Figure 3.13: Splice Environment Typing

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

$$\frac{\mathsf{P}; \bullet \vdash^{0} \mathsf{e} : \tau}{\mathsf{P} \vdash_{\mathsf{def}} \mathbf{def}\ \mathsf{k} : \tau = \mathsf{e} : \mathsf{P}, \mathsf{k} : \tau}\ \text{C\_DEF} \qquad \frac{\mathsf{P}; \Gamma \vdash^{\mathsf{n}} \mathsf{e} : \mathsf{Code}\ \tau}{\mathsf{P} \vdash_{\mathsf{spdef}} \mathbf{spdef}\ \Gamma \vdash^{\mathsf{n}} \mathsf{sp} : \tau = \mathsf{e} : \mathsf{P}, \mathsf{sp} : (\Gamma, \tau)}\ \text{C\_SPDEF}$$

Figure 3.14: Core Definition Typing

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

$$\frac{\mathsf{P}; \bullet \vdash^{0} \mathsf{e} : \tau}{\mathsf{P} \vdash_{\mathsf{pgm}} \mathsf{e} : \tau}\ \text{CP\_MAIN} \qquad \frac{\mathsf{P}_1 \vdash_{\mathsf{def}} \mathsf{def} : \mathsf{P}_2 \qquad \mathsf{P}_2 \vdash_{\mathsf{pgm}} \mathsf{pgm}}{\mathsf{P}_1 \vdash_{\mathsf{pgm}} \mathsf{def}; \mathsf{pgm}}\ \text{CP\_DEF}$$

$$\frac{\mathsf{P}_1 \vdash_{\mathsf{spdef}} \mathsf{spdef} : \mathsf{P}_2 \qquad \mathsf{P}_2 \vdash_{\mathsf{pgm}} \mathsf{pgm}}{\mathsf{P}_1 \vdash_{\mathsf{pgm}} \mathsf{spdef}; \mathsf{pgm}}\ \text{CP\_SPDEF}$$

Figure 3.15: Core Program Typing

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

$$\frac{\mathsf{e}_1 \rightsquigarrow \mathsf{e}_1'}{\mathsf{e}_1\ \mathsf{e}_2 \rightsquigarrow \mathsf{e}_1'\ \mathsf{e}_2}\ \text{D\_E\_APP\_L} \qquad \frac{\mathsf{e}_2 \rightsquigarrow \mathsf{e}_2'}{\mathsf{e}_1\ \mathsf{e}_2 \rightsquigarrow \mathsf{e}_1\ \mathsf{e}_2'}\ \text{D\_E\_APP\_R}$$

$$\frac{}{(\lambda x : \tau . \mathsf{e})\ \mathsf{v} \rightsquigarrow \mathsf{e}[\mathsf{v}/x]}\ \text{D\_E\_BETA} \qquad \frac{\mathsf{e} \rightsquigarrow \mathsf{e}'}{\mathsf{e}\ \langle\tau\rangle \rightsquigarrow \mathsf{e}'\ \langle\tau\rangle}\ \text{D\_E\_TAPP}$$

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

Figure 3.16: Dynamic Semantics

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

$$\frac{\mathsf{SP} \rightsquigarrow \mathsf{SP}'}{\mathsf{SP}, \Delta \vdash \mathsf{sp} : \tau = \mathsf{e} \rightsquigarrow \mathsf{SP}', \Delta \vdash \mathsf{sp} : \tau = \mathsf{e}}\ \text{D\_SP\_BODY}$$

$$\frac{\mathsf{e} \rightsquigarrow \mathsf{e}'}{\mathsf{SP}, \Delta \vdash \mathsf{sp} : \tau = \mathsf{e} \rightsquigarrow \mathsf{SP}, \Delta \vdash \mathsf{sp} : \tau = \mathsf{e}'}\ \text{D\_SP\_CONS}$$

Figure 3.17: Dynamic Semantics for Splice Environment

$$\boxed{\mathsf{def} \rightsquigarrow \mathsf{def}}$$

$$\frac{\mathsf{e} \rightsquigarrow \mathsf{e'}}{\textbf{def}\ \mathsf{k} : \tau = \mathsf{e} \rightsquigarrow \textbf{def}\ \mathsf{k} : \tau = \mathsf{e'}}\ \text{D\_DEF}$$

$$\boxed{\mathsf{spdef} \rightsquigarrow \mathsf{spdef}}$$

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

Figure 3.18: Dynamic Semantics for Definitions

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

$$\frac{\mathsf{def} \rightsquigarrow \mathsf{def'}}{\mathsf{def}; \mathsf{pgm} \rightsquigarrow \mathsf{def'}; \mathsf{pgm}}\ \text{D\_P\_DEF} \qquad \frac{\mathsf{spdef} \rightsquigarrow \mathsf{spdef'}}{\mathsf{spdef}; \mathsf{pgm} \rightsquigarrow \mathsf{spdef'}; \mathsf{pgm}}\ \text{D\_P\_SPDEF}$$

$$\frac{\mathsf{e} \rightsquigarrow \mathsf{e'}}{\mathsf{e} : \tau \rightsquigarrow \mathsf{e'} : \tau}\ \text{D\_P\_MAIN} \qquad \frac{}{\textbf{def}\ \mathsf{k} : \tau = \mathsf{v}; \mathsf{pgm} \rightsquigarrow \mathsf{pgm}[\mathsf{v}/\mathsf{k}]}\ \text{D\_P\_DEF\_BETA}$$

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

Figure 3.19: Dynamic Semantics for Program

3.5 Elaboration

In this section we describe the process of elaboration from the source language to the core language. Elaboration starts from a well-typed term in the source language and hence is defined by induction over the typing derivation tree (Figure 3.3).

3.5.1 Elaboration Procedure

There are three key aspects of the elaboration procedure:

1. Splices at positive levels are removed in favour of a splice environment. The elaboration process returns a splice environment which is attached to the quotation form.

2. Splices at non-positive levels are elaborated to top-level splice definitions, which are bound prior to the expression within which they were originally contained.

3. Type class constraints are converted to explicit dictionary passing. We describe how to understand the new constraint form $\mathsf{CodeC\ C}$ in terms of quotation.

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

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

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

3.5.2 Splice Elaboration

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

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

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

If a splice occurs at non-positive level, it is a top-level splice and will become a top-level splice definition in rules DEF or INST, in such a way that the splice definitions are made prior to the value definition which yielded them. The splice definitions are created by the $\mathsf{collapse}$ judgement (Figure 3.20), which takes a splice environment returned by the expression elaboration judgement and creates top-level splice declarations for each negative splice which appeared inside a term. To guarantee a stage-correct execution, the splices are inserted in order of their level.

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

3.5.3 Constraint Elaboration

The second point of interest is the constraint elaboration rules given in Figure 3.4. Since in our language type classes have just a single method, we use the function corresponding to the method itself as evidence for a class instance in rule INST.

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

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

3.5.4 Examples

As an example of elaboration, let us consider the expression $\$(\mathsf{c}'_1)$ where $\mathsf{c}'_1 = [\![\ \mathsf{show}\ ]\!]$ from Example C1. There are two points of interest here: there is a top-level splice which will be floated to a top-level splice definition, and the $\mathsf{CodeC}\ (\mathsf{Show\ a})$ constraint of c'_1 must be elaborated into quoted evidence using rule C_DECR.

We assume that the program environment contains c'_1:

$$\mathsf{P} = \bullet, \mathsf{c}'_1 : \forall \mathsf{a}.\mathsf{CodeC}\ (\mathsf{Show\ a}) \Rightarrow \mathsf{Code}\ (\mathsf{a} \to \mathsf{String})$$

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

$$\begin{aligned}&\mathsf{P}; \bullet \vdash^0 \$(\mathsf{c}'_1) : \forall \mathsf{a}.\mathsf{Show\ a} \Rightarrow \mathsf{a} \to \mathsf{String} \\ &\quad \rightsquigarrow \Lambda \mathsf{a}.\lambda \mathsf{ev} : \mathsf{a} \to \mathsf{String}.\mathsf{sp} \mid \{\Delta \vdash \mathsf{sp} : \mathsf{a} \to \mathsf{String} = \mathsf{c}'_1\ \langle \mathsf{a} \rangle\ [\![\ \mathsf{ev}\ ]\!]_\bullet\} \cup_{-1} \bullet\end{aligned}$$

where

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

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

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

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

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

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

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

$$\frac{\mathsf{ds} = \mathsf{collapse}(\mathsf{n}-1, \lfloor \mathsf{TSP} \rfloor^{\mathsf{n}}, \overline{\mathsf{spdef_i}}; \mathsf{pgm}) \qquad \overline{\mathsf{spdef_i} = \mathbf{spdef}\ \Delta_i \vdash^{\mathsf{n}} \mathsf{sp_i} : \tau_\mathsf{i} = \mathsf{e_i}} \quad \overline{\mathsf{TSP_n} = \Delta_\mathsf{i} \vdash \mathsf{sp_i} : \tau_\mathsf{i} = \mathsf{e_i}}}{\mathsf{collapse}(\mathsf{n}, \mathsf{TSP}, \mathsf{pgm}) = \mathsf{ds}}\ \text{C\_STRIP}$$

Figure 3.20: Definition of $collapse$

$$\mathbf{spdef}\ \Delta \vdash^{-1} \mathsf{sp} : \mathsf{a} \to \mathsf{String} = \mathsf{c}'_1\ \langle \mathsf{a} \rangle\ [\![\ \mathsf{ev}\ ]\!]_\bullet ; \mathsf{p}$$

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

As a second example we consider the elaboration of $[\![\ \lambda x : \mathsf{Int}.\$([\![\ x\ ]\!])\ ]\!]$ which demonstrates how the elaboration deals with open terms.

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

$$\bullet; \bullet \vdash^0 ([\![\ \lambda x : \mathsf{Int}.\$([\![\ x\ ]\!])\ ]\!]) : \mathsf{Code}\ (\mathsf{Int} \to \mathsf{Int})$$
$$\rightsquigarrow [\![\ \lambda x : \mathsf{Int}.\mathsf{sp}\ ]\!]_{\bullet, (\bullet, x : (\mathsf{Int}, 1)) \vdash \mathsf{sp} : \mathsf{Int} = [\![\ x\ ]\!]_\bullet} \mid \bullet$$

This example includes the quoted open term $[\![\ x\ ]\!]$, therefore the splice environment attached to the quotion includes an environment which records what the type of x was when the splice was lifted out of the quotation.

The body of the quotation is checked at level 1, therefore the variable introduced by $\lambda x : \mathsf{Int}.\cdot$ is added to the environment as $(\bullet, x : (\mathsf{Int}, 1)$ Then the splice is checked by rule E_SPLICE, which adds the body of the splice to the splice environment, and replaces it with a splice variable.

Then when the outer quote is checked at level 0, TSP_0 extracts the splice from the environment and adds it to the local environment for the outer quotation. As this was the only nested splice, the truncation results in the empty splice environment ($\lfloor \mathsf{TSP} \rfloor^0 = \bullet$). This means that at the end of elaboration, the top splice environment is empty ($\bullet$) because there are no top-level splices which we need to add into the program.

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

$$\frac{}{\bullet \rightsquigarrow \bullet}\ \text{TE\_EMPTY} \qquad \frac{\Gamma \rightsquigarrow \Delta \qquad \Gamma \vdash_{\mathsf{ty}} \tau \rightsquigarrow \tau'}{\Gamma, x : (\tau, \mathsf{n}) \rightsquigarrow \Delta, x : (\tau', \mathsf{n})}\ \text{TE\_VAR}$$

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

Figure 3.21: Elaboration of Type Environments

3.6 Pragmatic Considerations

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

3.6.1 Type Inference

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

3.6.2 Type Variables

In the formalism we stated with confidence that because of the phase-distinction that type variables, known statically before runtime, did not need to obey the same phase-distinction as term-variables which may be unknown until runtime. In a theoretical setting, as our calculus demonstrates, this is true. Theory poses no issues with substituting a type into a quotation and usually operates in the whole-program setting. Further to this, the reduction semantics explicitly represent and substitute types, into terms and into quotations.

In a practical setting the reality is not this straightforward. Languages such as Haskell have typically erased types by runtime. To highlight the problem, consider the quoted

identity function once again:

```
qid :: ∀a.Code (a → a)
qid = [|| id ||]
```

Neither of the two methods of executing Haskell programs, the bytecode interpreter and compilation to object code, retain type information at runtime. After qid is compiled, all the type arguments are erased. Therefore we are left in a situation where qid is supposed to generate well-typed core at runtime for any type but with no knowledge of which type it is actually applied to. Something else needs to be done in order to enable a well-typed intermediate program to be generated once the specific type that qid is used at is known.

This problem is quite similar to the problem which is solved by TypeRep (Peyton Jones et al., 2016). TypeRep is another runtime type representation which is normally used when a runtime type witness is needed. The typical use case is creating a heterogeneous map where each key can store something of a different type. When the key is retrieved from the map at a specific type, a runtime check ensures that the value has type type expected by the programmer. For this reason the original idea was to introduce a constraint like Typeable which provided a runtime representation of a type which could be used to construct a quotation.

Constraint Based Approach

The approach taken in the current implementation is to turn an irrelevant type argument into a relevant argument by using a special constraint. A constraint maps a type to a dictionary and therefore can be used to map a type to the representation of the type, and in this case the LiftT constraint maps a type to its Template Haskell representation. The evidence carried by LiftT will be used in a splice in a type position so that when the LiftT evidence is known and provided, the well-typed core program can be constructed by inserting the representation into the appropriate place.

```
class LiftT (t :: k) where
  liftTyCl :: TTExp
```

Principle Idea Behind LiftT The purpose of LiftT is to make sure that at the end of compilation that a quotation contains no cross-stage references to type variables. During type checking, LiftT constraints are emitted in places which might eventually contain a reference to a type variable. Specifically LiftT is emitted in three places:

1. On the type of each variable used inside a quotation
2. On the free variables of the return type of a quotation
3. On any type variable used in an explicit type application

The result of this is quite a large number of LiftT constraints, in normal programs most of these constraints will be easily solved as they will result in closed types. Unsolved constraints will correspond to unknown or free type variables and will be propagated to the context of the function so that the caller of the function, at the known type, should provide the relevant LiftT evidence. For example, the inferred type of qid will instead now include a LiftT constraint, as the information about the type variable a is needed in order to compute the result of calling qid.

$$\begin{aligned}&\mathsf{qid} :: \forall \mathsf{a}.\mathsf{LiftT}\ \mathsf{a} \Rightarrow \mathsf{Code}\ (\mathsf{a} \to \mathsf{a})\\&\mathsf{qid} = [\![\ \mathsf{id}\ ]\!]\end{aligned}$$

All the constraints emitted in these three places are collected and stored on the quotation. During elaboration, the solved constraints are used to substitute occurrences of free type variables for a splice which will contain the representation of the type. For example, if a new metavariable is created inside a quotation, then there needs to be a LiftT constraint attached to that metavariable because the value the type variable is attached to will ultimate appear inside the quotation.

Then qid is elaborated to include a type splice to insert the LiftT evidence in the location where the type variable a was applied to id.

$$\begin{aligned}&\mathsf{qid} :: \forall \mathsf{a}.\mathsf{LiftT}\ \mathsf{a} \to \mathsf{Code}\ (\mathsf{a} \to \mathsf{a})\\&\mathsf{qid}\ \mathsf{a}\ \mathsf{ltEv} = [\![\ \mathsf{id@\$(ltEv)}\ ]\!]\end{aligned}$$

So that when the type that qid is used at is known then the suitable LiftT evidence can be used to construct an appropiately typed core expression.

Solving LiftT Constraints LiftT constraints are solvable at any level for any types other than type variables. For an open type such as (a, b) then the $\mathsf{LiftT}\ (\mathsf{a}, \mathsf{b})$ constraint is solved by recursively emitting goals for solving LiftT a and LiftT b.

For type variables, a LiftT a constraint can be solved at level k if the type variable a is rigid and introduced at level $\mathsf{k} + 1$. This solving rule is critical in order to generate polymorphic programs. For example, considering how to use the qid function defined above in a top-level splice in order to generate a polymorphic identity function.

```
newId :: ∀b.b → b
newId = $(qid)
```

The qid function requires a LiftT b constraint at level -1. The rigid variable b is introduced at level 0 so the LiftT b constraint is solved, and the evidence is this quoted type variable. This way of solving LiftT constraints is similar to how in normal multi-stage programming it is permitted to quote open terms in some similar circumstances.

For now, a type argument is made relevant using this constraint based approach but there are certain issues with the implementation in this approach which will be described in the next chapter (Section 4.1.4).

It is also worth noting that Dotty (Stucki et al., 2018) takes a similar approach to LiftT constraints in order to persist types into quotations. They face a similar issue to Haskell where by default the types are erased by runtime. Their runtime representation is called `Type[T]` and a cross-stage type reference of a type `T` requires emits an implicit argument requiring `Type[T]` to be provided.

Relevance Quantifier

A more principled solution is to introduce a new quantifier form which can introduce relevant type variables. A relevant type variable will carry its Template Haskell representation at runtime and therefore as the program is being generated the type information can be used in order to construct well-typed core. Any rigid type variable appearing inside a quotation would be required to be a relevantly quantified variable. Then from the representation the Template Haskell representation for the passed type would be computed and inserted into the right place. Suppose that a new quantifier forallr is introduced in order to state that a specific type variable should be relevant, the definition of qid would be modified to be:

```
qid :: forallr a.Code (a → a)
qid = ⟦ id ⟧
```

This approach removes the need for complications to do with metavariables that the LiftT approach ran into but adds complexity as the runtime has to be extended to explicitly pass type arguments. Many languages already pass type arguments at runtime so in principle there is no great difficulty. There will probably be some further engineering challenges but aside from introducing the new syntax for the different quantifier and then modifying the code generation to create and pass a type representation there is not much to implement. Once the type arguments are passed

at runtime, it will be straightforward to replace free type variables appearing inside a quotation by a projection to extract the representation of the type.

The idea of a relevance quantifier (and indeed relevance polymorphism) has become popular again due to the proposals around Dependent Haskell (Eisenberg, 2016; Weirich et al., 2017). The precise implementation details of this quantifier are not spelled out in any published work to my knowledge. Nor is its applicability to staged programming discussed in any detail.

In much older versions of GHC, there was the flag which kept runtime type information for backends which required type information (originally for the .NET ILX) but that was removed in 2008. Now we are doing code generation for a backend (the core language) which requires type information, a similar solution is needed again.

Erasure

Finally, we could also avoid all the problems with `LiftT` if the programs we generated didn't contain type annotations. The reason for targeting core in the implementation was because then the normal optimisation passes can be used to further improve the generated code. Core is an intrinsically typed language and so every binder must be annotated with its type. An alternative to this is to erase any unknown type variable from the generated program and replace it with a placeholder value. The resulting core program would still behave identically as type arguments are not relevant for computation but there are two disadvantages to removing the type information.

1. The generated program can't be type checked using the core linter.
2. Certain optimisations which rely on type information may be inhibited by erasing types. In general it is known that placing calls to `unsafeCoerce` can also inhibit the optimiser and this erasure would have a similar effect (Kusee, 2017).

The erasure approach is similar to the method used to extract a Haskell program from an Agda program. In essence the translation from an Agda program to a Haskell program is quite direct but in order to patch over the expressive difference between the two type systems certain expressions are wrapped in `unsafeCoerce`. The resulting Haskell program is littered with calls to `unsafeCoerce` (Kusee, 2017) which inhibit the optimiser.

In a language where the type information dwarfs the runtime content of terms, it would be desirable to also explore the option to store erased or partially erased terms (Brady et al., 2003; Jay and Peyton Jones, 2008).

3.6.3 Interaction with Existing Features

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

GADTs

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

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

Quantified Constraints

The quantified constraints extension (Bottu et al., 2017) relaxes the form of the constraint schemes allowed in method contexts to also allow the quantified and implication forms which in our calculus are restricted to top-level axioms. Under this restriction there are some questions about how the CodeC constraint form should interact especially with implication constraints. In particular, whether constraint entailment should deduce that $\mathsf{CodeC}\ (\mathsf{C}_1 \Rightarrow \mathsf{C}_2)$ entails $\mathsf{CodeC}\ \mathsf{C}_1 \Rightarrow \mathsf{CodeC}\ \mathsf{C}_2$ or the inverse and what consequences this has for type inference involving these more complicated constraint forms.

3.6.4 Interaction with Future Features

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

Impredicativity

For a number of the examples that we have discussed in this paper, an alternative would be to use impredicative instantiation to more precisely express the binding position of a constraint. For instance, the function $\mathsf{c}_1 :: \mathsf{Show\ a} \Rightarrow \mathsf{Code\ (a \rightarrow String)}$ from Example C1 might instead have been expressed in the following manner:

$$\mathsf{c}'_1 :: \mathsf{Code\ (\forall a.Show\ a \Rightarrow a \rightarrow String)}$$
$$\mathsf{c}'_1 = [\![\ \mathsf{show}\ ]\!]$$

The type of c'_1 now binds and uses the $\mathsf{Show\ a}$ constraint at level 1 without the use of CodeC.

However, despite many attempts (Peyton Jones et al., 2007), GHC has never properly supported impredicative instantiation due to complications with type inference. Recent work (Serrano et al., 2018, 2020) has proposed the inclusion of restricted impredicative instantiations, and these would accept c'_1. In any case, impredicativity is not a silver bullet, and there is still a need for the CodeC constraint form. Without the CodeC constraint form there is no way to manipulate "open" constraints (constraints which elaborate to quotations containing free variables).

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

Dependent Haskell

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

The specification for Dependent Haskell (Weirich et al., 2017) introduces the so-called relevance quantifier in order to distinguish between relevant and irrelevant variables.

The irrelevant quantifier is intended to model a form of parametric polymorphism like the $\forall$ in System F, the relevant quantifier is written as Π after the dependent quantifier from dependent type theory.

Perhaps it is sound to modify their system in order to enforce the stage discipline for relevant type variables not irrelevant ones. It may be that the concept of relevance should be framed in terms of stages, where the irrelevant stage proceeds all relevant and computation stages – in which case it might be desirable to separate the irrelevant stage itself into multiple stages which can be evaluated in turn. The exact nature of this interaction is left as a question for future work.

3.6.5 Alternative to CodeC

The introduction of the CodeC constraint was necessary in order to allow us to use overloaded functions inside quotations. The special constraint makes sure that the usage of an dictionary can be elaborated into something which is level correct. What this ultimately amounts to is top-level identifiers being quoted and passed around before being directly inserted into the generated programs in the correct place.

This approach is quite conservative and the disadvantage is that the original dictionaries will still appear in the generated code and so we could still be at the whim of the specialiser in order to remove the overhead of the dictionary translation. It would be preferable if our treatment of constraints would also remove the need for a specialisation pass, and the idea behind how to solve that is sketched in this section.

The principal idea is to perform binding-time improvement on the datatype used to represent a dictionary by pushing the Code constructor one level further in. This exploits the static knowledge that a dictionary is a k-product of class methods. Therefore the static knowledge of the structure of the dictionary can also be exploited at compile-time so that the dictionary structure is guaranteed to not appear.

Let's consider a simple example of an idealised Show class and an explicit ShowDict data type which will be used to represent the dictionary for Show.

```
class Show a where
  show :: a → String
data ShowDict a = ShowDict {dshow :: a → String}
```

In the formalism for CodeC, CodeC (Show a) is represented by Code (ShowDict a). Under this new proposal, instead a new representation data type is created which is

similar to ShowDict but each field is wrapped with the Code constructor. This is a binding-time improvement transformation from Code (ShowDict a) to CShowDict a.

```
data CShowDict a = CShowDict {cshow :: Code (a → String)}
```

Now the idea is that CodeC (Show a) is instead represented by CShowDict a, in order to make this work the dictionary desugaring will have to be modified but once that change is made then specialisation will no longer be necessary as the generated code will no longer contain any mention of ShowDict or CShowDict.

Modifying Elaboration

The informal idea behind elaboration is to replace uses of class methods such as show inside a quotation with the selectors for the new dictionary form such as cshow. Using a CodeC (Show a) constraint now will elaborate to a normal function which accepts CShowDict a record as an argument. Then inside the quotation, a usage of show will be replaced with $(cshow d) and the selection of the class method will happen during compile-time. Time for an example:

foo is defined as the quoted show function. It will suffice to demonstrate the principle behind the translation.

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

The CodeC (Show a) constraint is elaborated to the CShowDict a type, and the selector inside the quotation replaced with a splice which selects the correct method from CShowDict a.

```
foo :: CShowDict a → Code (a → String)
foo sd = ⟦ $(cshow sd) ⟧
```

Now for a specific datatype Baz, what would the generated program look like if foo was used at type Baz.

```
data Baz = Baz

instance Show Baz where
  show Baz = "baz"
```

The Show Baz instance results in two new internal definitions. A normal dictionary of type ShowDict and another of type CShowDict which will be used when elaborating class methods used inside quotations.

```
dShowBazShow :: Baz → String
dShowBazShow Baz = "baz"

dShowBaz :: ShowDict Baz
dShowBaz = ShowDict dShowBazShow

dShowBazC :: CShowDict Baz
dShowBazC = CShowDict ⟦ dShowBazShow ⟧
```

Then using foo at type Baz will elaborate to use the dShowBazC function and the generated code will *not* contain a record selector.

```
showBaz :: Baz → String
showBaz = $(foo dShowBazC)
  -- Generated code does not contain an accessor
showBaz = dShowBazShow
```

In this example, the advantage of removing the indirection of one function call may not appear so important but as discussed in Section 5.3, specialisation is a very important optimisation which is designed to remove these kinds of indirections.

In Composition What happens when you use foo in order to produce a larger code generator? The constraint form propagates and is solved like normal. doubleShow uses foo twice, the CodeC (Show a) dictionary is just passed to foo like normal because foo already appears inside a splice.

```
doublePrint :: CodeC (Show a) ⇒ Code (a → String)
doublePrint = ⟦ λa → $(foo) a ++ $(foo) a ⟧
```

When doublePrint is elaborated, the dictionary d is passed to foo inside the splice. This is in contrast to before where the dictionary would be spliced into the quotation.

```
doublePrintElab d = ⟦ λa → $(foo d) a ++ $(foo d) a ⟧
```

The modifications are local to the point where the method is projected from the dictionary.

Modifications to Formalism In order to support the new elaboration idea for CodeC there would need to be a few modifications to the formalism.

1. Extend the formalism to add data type declarations and type classes with multiple methods.

2. Generate a new form of dictionary according to the specification. Each instance generates both a normal dictionary and a partially static dictionary which is used when satisfying CodeC constraints.

3. A type class method used inside a quotation is desugared to require this new dictionary form rather than a quoted dictionary as in the current formalism.

The technique which we have described in this section could be performed automatically by the compiler. Others have already exploited this optimisation by hand, for an example of similar treatment but in a manual dictionary passing see Willis et al. (2020).

3.7 Other API Changes

In this section we will reflect on some more changes to the existing API which are either motivated or related of the idea of using a different representation form for quotations.

Along with the fundamental changes of the new representation type and the modifications to how constraints interact with typed quotations, there are also a number of other points in the API which need to be modified to provide the best ergonomic experience.

3.7.1 Effectful Code Generation

In the formalism so far, we have used the abstract Code type in order to represent expressions. The reality of this situation is more complicated. The Code constructor actually has been extended in order to carry an additional type parameter which indicates the monadic context which will be used by the code generator. The extension to the formalism is quite straightforward to specify but we will only informally specify the extension and API here. This work was originally part of (Pickering, 2019).

The principle idea is that Code a is extended to Code m a. The type parameter m is the monadic context which the splices will run in. This is useful for writing effectful code generators which use effects such as an environment, state or exceptions.

Quoting an expression of type a yields a value of type Quote m ⇒ Code m a. Quote is a type class which is defined to support just the operations needed to create the

representation. In the current implementation the Quote class just contains one method which is used to generate fresh names. Therefore the class can be implemented using a simple State monad, an IORef in IO or some other more powerful monad such as Q.

```
class Monad m ⇒ Quote m where
  newName :: String → m Name
```

The type of ⟦ () ⟧ is now Quote m ⇒ Code m ().

The monadic type parameter m comes into use when the quotation contains nested splices. The constraints of all nested splices are propagated into the context and then m can be instantiated to any type variable which satisfies the requirements placed on it by the specific quotation.

Suppose f has a more restrained type than usual with the addition of the C constraint.

```
f :: (Quote m, C m) ⇒ Code m Int
f = ⟦ 5 ⟧
```

When f is used in a nested splice constraint on f, namely C, is propagated to a constraint on the whole representation.

```
g :: (Quote m, C m) ⇒ Code m Int
g = ⟦ $(f) + $(f) ⟧
```

In a top-level splice, the resulting expression must still have type Code Q a, which is similar to how the top-level main function must have type IO (). For untyped quotations, the overloaded quote can be run usefully in other ways than at the top-level, for example, in order to construct and inspect the TEMPLATE-HASKELL syntax tree at runtime. For typed quotations this ability is less useful because the new representation is opaque. Overloaded quotations are still useful though for threading a context through a code generator.

Code Generation with Effects

Overloaded quotations make programming with effects and quotations natural. Effectful code generation is indispensable, for example, consider generating any program with division. If you statically know that you will generate a program which divides by zero you would prefer to throw an exception at compile time rather than generate a division which will certainly fail at runtime. In general, effects are used during code generation

in multi-stage languages in just the same manner as when writing normal programs because writing code generators is just the same as writing programs which manipulate fragments of code.

The Q monad has long supported a limited range of effects, but this takes us away from programming in the usual MTL style. Using monad transformers to implement effectful computations plays more nicely into the rest of the Haskell ecosystem and being able to use the same programming style when writing staged and unstaged programs is very important in the staging methodology.

It is also an important fact to consider when refactoring an interpreter into a staged style. If your interpreter is implemented in the standard MTL style (Jones, 1995b) with a stack of monad transformers, during the refactoring process you need to distinguish between which effects happen at compile-time and runtime. Practically this amounts to splitting up the monad transformer stack with part of the stack used during code generation and the other part appearing in the generated code. In the original program, the environment carried by the reader monad may comprise of static and dynamic information, therefore it would be split up into two reader transformers with an separate environment for the static and dynamic. In the old style of programming, after splitting up the monad stack, the static reader component had to be implemented in terms of methods from the Q monad. In the new style, the same MTL style can be used during program generation, and only interpreting any latent effects into the Q monad just before executing the top-level splice.

For example, using the new API consider a code generator with two split reader monads.

```
data StaticInfo = StaticInfo ...
data DynamicInfo = DynamicInfo ...
myInterpreter :: Code (ReaderT StaticInfo m) (ReaderT DynamicInfo IO Int)
```

myInterpreter uses a reader monad during code generation in order to generate a program which uses the reader monad at runtime. For intuition the StaticInfo may contain information such as which flags are enabled for the interpreter, information which doesn't change through the whole run of the interpreter. The dynamic information carried by the inner reader monad may be information such as a variable environment.

Finally, as the top-level splice requires a value of type Code Q a, the outer reader effect needs to be interpreted before the splice can run. The combinator hoistCode can transform the inner monadic action by running the ReaderT transformer.

```
hoistCode :: (∀a.m a → n a) → Code m a → Code n a
```

```
runStaticReader :: Quote m ⇒ StaticInfo → Code (ReaderT StaticInfo m) a
                                        → Code m a
runStaticReader si act = hoistCode (flip runReaderT si) act

interpret :: ReaderT DynamicInfo IO Int
interpret = $(runStaticReader (StaticInfo...) myInterpreter)
```

This abstract but simple example should highlight how during refactoring it is natural to want to split up existing monad transformer stacks in order to use some effects at compile-time and some at runtime. Of course, the more computation and checks which can be moved to compile-time the better.

A New Monad Untyped splices require an expression of type Code Q a. The Q monad provides a number of operations which are impossible or undesirable to support for Typed Splices. Therefore a new monad is introduced, TC, which provides methods specific to typed quotations. Making this more fine-grained also allows for further separate extension in future without considering how the feature can work for both untyped and typed quotations.

```
class TC m where
  tqTypecheck :: TTExp → Exp → Code m t

  tqReport :: Bool → String → m ()
  tqRunIO :: IO a → m a

  tqAddDependentFile :: FilePath → m ()
  tqAddTempFile :: String → m FilePath
  tqAddForeignFilePath :: ForeignSrcLang → String → m ()
  tqAddCorePlugin :: String → m ()
```

The tqTypecheck method allows for untyped expressions to be injected into the type representation. The tqReport operation is still useful to provide good error messages to users if code generation results in a warning at compile time. The tqRunIO operation is an invaluable escape hatch to perform IO actions during compile-time, for example, to read from a file. tqAddDependentFile adds a dependency on a certain file so that if it is modified then the module will be recompiled. tqAddTempFile creates a new temporary file to be cleaned up at the end of compilation. tqAddForeignFilePath adds a new foreign file which will be linked against the object for the current module. tqAddCorePlugin dynamically enables a core plugin pass.

Of note, all the reification functions are missing from this list. They are not needed or useful when using only typed quotations. The qRecover function and the qGetQ and

qPutQ functions are also removed. If you want to do code generation using exceptions or state then it is recommended to instead use the normal monad transformer approach.

3.7.2 What to do about Lift?

In the overloaded quotations extension described in Section 3.7.1 we noted that the type of lift was divorced from the Q monad so that untyped quotations could be used to generate syntax without using the Q monad.

Now the typed quotation representation has changed, we also have to consider how liftTyped needs to be modified to fit into the new interface. At first blush it appears that a type such as liftTyped :: Monad m ⇒ a → Code m a may be appropiate because the generation of the type representation doesn't require any specific functions whereas the untyped representation requires the newName function to generate fresh names. The issue with this very general type is that many Lift instances are currently implemented in terms of the untyped lift method by the composition of unsafeTExpCoerce and lift. unsafeTExpCoerce has been replaced by tqTypeCheck in the proposed modifications to the interface but the implementation of tqTypeCheck is quite a stringent requirement.

However, the problem is not as serious as it appears. It is hard to imagine a use of typed quotations which does not eventually end up with the code being used inside a top-level splice. The representation is completely opaque so inspecting it will have far less value than the possibility of inspecting the untyped representation. Therefore we instead change liftTyped back to having a more stringent constraint on the TC monad.

liftTyped :: TC m ⇒ a → Code m a

The TC constraint enables instances to be implemented as a combination of lift and typecheck but also means that typed code generation can use standard monad transformers such as StateT if the base monad implements TC.

A New Extension for Typed Template Haskell Providing a separate extension to `TemplateHaskell` to enabled typed quotations and splicing will mean that programs can be completely typechecked before having to run any top-level splices. From a practical viewpoint this is a great improvement as programs using only the typed portion of Template Haskell will compile faster and work better with other IDE features.

Implementing an Escape Hatch It is also important to consider how Untyped and Typed Template Haskell interact with each other. At the moment Untyped Template

Haskell can be used as an escape hatch in order to implement custom code generation which is difficult to achieve in the Typed style. In particular, when implementing let insertion it is common to resort to using the untyped combinators to construct the let bindings. In the standard expression style the only way to introduce a let is by explicitly writing it, it is difficult to construct a mutually recursive block of let bound variables for instance but trivial in the untyped setting.

As a first attempt at providing support, a new primitive can be added to the new TC monad which enables an untyped expression to be injected into the typed representation. This is known as tqTypecheck and when given an expression and a type, verifies that the expression has the claimed type, raising an error at compile-time if the type is not as claimed by the user.

```
tqTypecheck :: TC m ⇒ TTExp → Exp → Code m t
```

By using cross-stage persistence for types a more convenient interface can be provided where the Type argument is created automatically by using the same machinery as LiftT.

```
typecheck :: ∀r (a :: TYPE r).LiftT a ⇒ Exp → Code m a
typecheck e = do
  let t = (liftTyCl @_ @a)
  tqTypecheck t e
```

As this is the only way for a user to create a TTExp argument, then using typecheck is the only way to use the tqTypecheck interface. tqTypecheck works well for lifting the untyped representation into the typed representation but these escape hatches also require the ability to embed typed fragments into untyped fragments. Considering the case of generating let expressions the right-hand-sides of the bound variables are usually passed as typed expressions which are combined together using the untyped Let constructor before being injected back into the typed representation.

There are two options. Either the typed representation can also contain the untyped representation or a new constructor can be added to Exp to allow a TExp a to be embedded into the untyped representation. In the first case, the unType function is now implemented by projection in the second case, the unType function implemented by using the new constructor.

The first proposal is easier to implement overall as it requires one local change to the typed representation but in doing so negates some of the benefits to the new representation. The combinators used to build the untyped representation can lead to very big expressions which take a long time to typecheck. The new representation was

supposed to avoid this issue but now if we included both the problem would be there again. Therefore it is probably impractical to take this implementation path.

A better idea is to allow the typed representation to be directly embedded into the untyped representation by means of a new constructor. The implementation for this is more complicated as now the internal compiler AST also has to be extended so that the typed representation can be stored until the moment it can be reinserted into the tree. It is also unclear how well it would work to use an untyped splice on a typed quotation embedded in the AST. This approach would also not support the inspection, manipulation or printing of the typed fragment so the feature should *only* be used in order to facilitate further typed code generation. Overall it seems like a more attractive idea because there is no additional cost to normal uses of Typed Template Haskell.

Both approaches allow the unType operation to be implemented for backwards compatibility.

3.8 Chapter Summary

In this chapter we first described the fundamental soundness issues facing Typed Template Haskell and sketched a proposal in order to remedy them. The proposal was the simple suggestion to retain the result of elaboration in the representation of quotations. This had quite far reaching consequences as soon as you start storing more information inside quotations you have to concern yourself with the binding position of terms in the quotation. In particular, we set out formally to understand the interaction of constraints and quotations which resulted in the introduction of the CodeC constraint form. Along the way how other implicit information such as type variable and implicit parameters were also considered less formally. Armed with the knowledge about how best to fix the soundness issues in Typed Template Haskell the next chapter will be about implementing these ideas and will culminate in a case study which uses these new features in order to stage a generic programming library.

Chapter 4

Implementation

In this chapter we will discuss the significant challenges faced when implementing the specification for Typed Template Haskell as described in the previous chapter. As it may be expected, integrating any simple specification into GHC is a significant undertaking and raises unforeseen issues which we will discuss in this chapter and will be useful for other implementers. After discussing the choices made during the representation, in Section 4.2 a collection of microbenchmarks which test the performance of the implementation will be presented and discussed.

4.1 Choosing a Representation

The first observation which led us down this foundational path was the observation that using an untyped internal representation for quotations meant that there were several vectors of unsoundness. We decided that in order to fix this, the internal representation of quotations needed to retain more implicit information only discovered from the context where the quotation was used. In particular, along with the actual meat of the expression, we also need to know about the types involved and other implicit arguments filled in by the typechecker.

The representation of terms needs to contain typechecked information and therefore needs to be one of the representations that GHC uses after typechecking (Marlow and Peyton Jones, 2012). Figure 4.1 shows the GHC compilation pipeline. Each node indicates a different internal representation and the arrows indicate transformations between the different representations. The yellow boxes in the diagram indicate representations which contain elaborated information. The current implementation of Typed Template Haskell is demostrated by the loop on the right of the diagram which

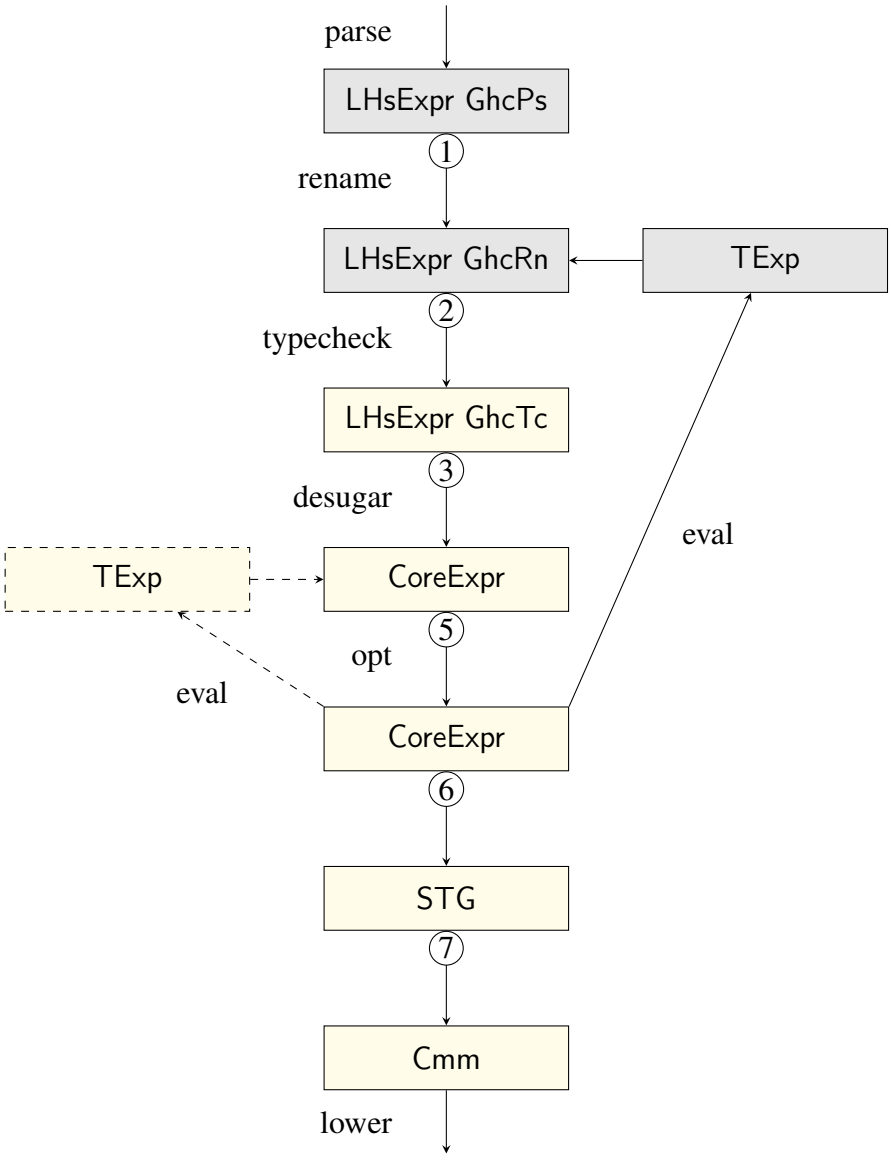

Figure 4.1: GHC Compilation Pipeline

shows how a core expression is evaluated in order to yield a TExp expression, which is then converted into a renamed expression.

There are two obvious candidates for the representation of terms, LHsExpr GhcTc terms which are fully-elaborated source syntax, or CoreExpr terms which are the representation terms for the CORE language. In our implementation we choose to use CoreExprs. The loop on the left indicates the proposed implementation, which evaluates a core expression in order to yield another core expression which is inserted at the start of the core optimisation pipeline.

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

- CORE expressions are explicitly typed so after they are serialised the type information is fixed.
- CORE expressions are already serialised and loaded by the compiler in order to support inlining definitions across modules. The same machinery can be reused for representing typed quotations.
- There is no redundant work performed typechecking expressions which have already been typechecked.
- Program compilation is faster. Modules which use a lot of quotations generate large programs as the existing representation type needs to closely mirror the source language as it can be inspected. These big terms can take a long time to compile.
- Generated programs are optimised like normal user written programs so new optimisation opportunities can be taken advantage of. This is important as distant fragments of code come together to unveil simple optimisation opportunities such as β-reduction or case elimination. The automatic cleanup is important for any explicit staging framework. Lower level representations start to introduce undesirable machine dependence and reduce other optimisation opportunities.

In the new representation there are new kinds of values which are introduced during elaboration. Therefore the new representation forces you to take care of cross-stage

persistence as if you do not, scope extrusion errors are very easy to introduce during elaboration. The introduction of CodeC constraints was motivated to fix these cross-stage errors for constraints, LiftT for type variables and Lift for normal values.

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

Therefore the fundamental idea is that a quotation ⟦ e ⟧ is represented by the core expression which represents e. The same serialisation and deserialisation logic is reused from the logic which is used to serialise core expressions. We will have to modify it to support quoting open terms but in general the representation of e will an opaque compressed bytestring. The representation will not be built using combinators like the current implementation or MetaOCaml (Kiselyov, 2014).

Splices and an Opaque Representation A splice inside a quotation is a point where substitution needs to be performed after an expression has been evaluated. When building expressions with combinators, the substitution step was performed when the expression was translated to the combinator language. Splices are interleaved with the combinators so that when the overall expression is evaluated the effect is that the result of the splice is substituted into the expression. On the other hand, the new representation is not built from combinators so there is no convenient place to put the bodies of splices so that when the expression is evaluated the splices will be immediately substituted into the right place. Therefore the substitution is also represented explicitly and performed explicitly when the body of the quotation is deserialised.

Substituting splices takes two steps. Firstly the body of a quotation is traversed and any splice is replaced by a *splice point* which marks a substitution point where the result of running a splice needs to be inserted. The corresponding splice is added to an environment which maps the splice point identifier to the splice. When the representation is loaded back into the compiler, for each splice point, the environment is consulted and the splice point substituted for the result of evaluating the splice. There are three different varieties of splices, one for values, one for types and one for constraints.

4.1.1 Implementation Strategy

The first question that has to be answered is the precise representations of expressions. Storing a single CORE expression is not enough as it doesn't account for nested splices. There are three different environments for storing splices. The tenv is for type splices, which are needed for substituting types into quotations, the eenv is for normal expression splices. ev is for quoted evidence, which can't contain nested splices and expr_renamed is a substitution for dynamically renaming the expression. The quoted expression itself is stored as a ByteString, we will write the ByteString representing an expression e as $\boxed{\mathsf{e}}$.

```
data TExpU = TExpU {tenv :: [(Int, TTExp)]
                   , eenv :: [(Int, TExpU)]
                   , ev :: [(Int, THRep)]
                   , expr_renamed :: [(Int, Int)]
                   , expr :: ByteString}
```

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

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

Quoting an expression of type e results in an expression of type Code e. For example, ⟦ True ⟧ :: Code Bool would be represented by:

```
qtrue = Code (TExpU [] [] [] [] True)
```

The value True contains no splices nor binders therefore all environments are empty. The expression True is turned into a CORE expression by desugaring and then serialised to a ByteString.

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

delayed until the top-level splice is executed. The environments are used to delay this evaluation until the top-level splice is used in another module.

$$\begin{aligned}&\mathsf{qfalse} :: \mathsf{Code\ Bool}\\&\mathsf{qfalse} = [\![\ \mathsf{not}\ \$(\mathsf{qtrue})\ ]\!]\end{aligned}$$

qfalse uses an explicit value splice which means the environment is populated with one pair for the single splice. For example, 213131 may be the unique integer representing the splice point so $\$(\mathsf{qtrue})$ would be replaced with 213131 and then serialised.

$$\mathsf{qfalse} = \mathsf{Code}\ (\mathsf{TExpU}\ [\,]\ [(213131, \mathsf{qtrue})]\ [\,]\ [\,]\ \boxed{\mathsf{not}\ 213131})$$

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

The final result of loading the information represented in a value of type Code a is a CORE expression which when evaluated produces a result of type a.

The second environment is for type splices. Type splices arise either from using explicit type application or in any other situation where the core term is annotated with a variable implicitly. There are a surprising amount of type splices present in a program as the core term usually contains some type variables. The simplest program which will contain a type splice is the quoted identity function.

$$\begin{aligned}&\mathsf{qid} :: \forall \mathsf{a}.\mathsf{LiftT\ a} \Rightarrow \mathsf{Code}\ (\mathsf{a} \to \mathsf{a})\\&\mathsf{qid} = [\![\ \mathsf{id}\ ]\!]\end{aligned}$$

qid will be elaborated to contain a type splice which will use the LiftT evidence to copy the type qid is applied to inside the quotation.

$$\mathsf{qid} = \Lambda.\, a \to \lambda \mathsf{dLiftT} \to [\![\ \mathsf{id@\$(dLiftT)}\ ]\!]$$

Therefore the representation will contain one entry in the type splice environment.

$$\mathsf{qid} = \Lambda.\, a \to \lambda \mathsf{dLiftT} \to \mathsf{Code}\ (\mathsf{TExpU}\ [\,]\ [\,]\ (1234, \mathsf{dLiftT})\ [\,]\ \boxed{\mathsf{id@}1234})$$

The third environment is used to store evidence splices. The form of splices created during constraint solving is simpler than the structure of splices which can be found in user programs. They will not contain nested splices, type splices or need dynamic renaming which is why the THRep is stored directly rather than the augmented TExpU.

Type Representation Types are represented using TTExp which is defined as follows:

$$\mathbf{data}\ \mathsf{TTExp} = \mathsf{TTExp}\ \{\mathsf{env\_t} :: [(\mathsf{Int}, \mathsf{TTExp})], \mathsf{tstr} :: \mathsf{THRep}\}$$

Type representations only contain one environment for nested type splices and the representation is a type serialised as a bytestring rather than an expression.

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

In order to load a CORE expression the bytestring is deserialised to an IfaceExpr which is deserialised to produce the necessary CoreExpr. Whilst deserialising the IfaceExpr any splice points are replaced by the result of performing the splice. The result of performing a splice is an expression of type TExpU which could itself contain splices, therefore, in order to load it, the same method is called recursively interpreting the TExpU as a CoreExpr.

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

$$\mathsf{lam} = [\![\lambda \mathsf{x} \rightarrow \$([\![\mathsf{x}]\!])]\!]$$

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

Scope Extrusion Scope extrusion is when a piece of generated code contains reference to a variable that is not in scope. In systems such as MetaOCaml, the scope extrusion check is performed at splice time (Kiselyov, 2014). Any unbound variable is detected and an error is raised.

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

Dynamic Variable Renaming Dynamic variable renaming makes sure that each time an expression is executed each bound variable in the quotation is given a fresh name. For example, each time qid is used, the generated program will use a fresh name for the variable x as it is bound inside the quotation.

```
qid = [[ λx → x ]]
```

It wasn't obvious to me at first that it was necessary to dynamically rename binders in the new implementation but in retrospect, it is clearly a requirement. In particular, if you write a recursive code generator which binds a variable inside a quotation and then passes an open code fragment which mentions this variable it's important to create a unique name for the variable x so that it is not confused with the nested xs which will be bound in the recursive call.

```
scurry_NP ::
  ∀r xs.(All LiftT xs, AllTails (LiftTCurry r) xs) ⇒
  (NP C xs → Code r) → Code (Curry r xs)
scurry_NP =
  case sList :: SList xs of
    SNil → λf → f Nil
    SCons → λf → [[ λx → $(scurry_NP (λxs → f (C [[ x ]] :* xs))) ]]
```

The choice of representation meant that the strategy used in Untyped Template Haskell had to be modified. In Untyped Template Haskell all binding structures are elaborated into a call to the newName function which binds a fresh name (at runtime) and the variable is passed to the combinators which construct the binding

For example:

```
untypedId = [| λx → x |]
 ⇝
untypedId = newName "x" >>= λx → lamE x (varE x)
```

Every time that untypedId is executed it will construct an expression with fresh identifiers.

In the core representation, you can't immediately substitute the variables into the representation of an expression. In particular, notice in the untyped case that the occurrence of x in the combinator language is a normal Haskell variable and so will be substituted and evaluated as a normal Haskell variable. There's not immediately

anywhere to put a reference to an actual variable in the TExpU representation as the expression is stored as a bytestring rather than built from the combinators. In essence this is exactly the same issue as with splices, type splices and constraint splices. So the solution is also similar, to create a new environment which will be used to perform substitution for the bound variables in an expression when the expression is reloaded into the program.

The implementation first traverses a quotation to find all the identifiers bound by the quotation. In the case of the identity function there is just one binder, x. Then all the binders are dynamically renamed at once on the outside of the quotation. This is implemented in the bindVars function.

```
bindVars :: ∀r :: RuntimeRep (a :: TYPE r) m.Quote m
   ⇒ [Int] → ([(Int, Int)] → m (TExp a)) → m (TExp a)
bindVars vs k = do
  renamed_vars ← mapM (λn → (n, ) ‹$› freshInt) vs
  k renamed_vars
```

At compile time, a program is generated where the key which was used to serialise a binder is passed to the bindVars function. Then when the function is executed at runtime, a fresh name will be created and stored along with the serialised representation.

When the code fragment is interpreted, the environment created by renaming the bound variables is used as a substitution to rename the bound variables in the quotation and their occurrences. This way each call to the function ends up creating distinct names for the binders, and so variables are not bound incorrectly when using functions such as scurry.

bindVars is implemented in continuation-passing style so that the rebound environment can be propagated to quotations nested in the same static scope. In particular consider the combination of a variation of quoting the identity function where the body is a spliced quoted open term.

```
qid = ⟦ λx → $(⟦ x ⟧) ⟧
```

The outer quotation binds the variable x and so the traversal will find this bound variable and add an entry to the dynamic renaming environment so it will have a fresh name each time qid is called. The inner quotation is in the same scope as the bound x and therefore when the variable x is loaded it must be bound to the newly bound name as well. Looking directly at the core in this situation has the potential to be enlightening:

```
-- RHS size: {terms: 143, types: 234, coercions: 0, joins: 0/0}
qid
  = bindVars
      $fQuoteQ
      [6989586621679012506]
      (\ ds_d23z ->
         unsafeTExpCoerce
           $fQuoteQ
           (Nothing, ...
            (6989586621679012507,
               unTypeQT
                 $fQuoteQ
                 (bindVars
                    $fQuoteQ
                    []
                    (\ ds1_d23A ->
                       unsafeTExpCoerce
                         $fQuoteQ
                         (Nothing, ...,
                          [], [], ds1_d23A ++ ds_d23z ++ [],
                          mkTHRep
                            (I# 24#)
                            <BS>))))
             [],
            [], ds_d23z ++ [],
            mkTHRep
              (I# 46#)
              <BS>
```

The first call to bindVars takes the single unique identifier of the bound variable x as an argument, 6989586621679012506, and then binds the substitution to ds_d23z, In the main quotation, the dynamic environment is ds_d23z ++ [], or just this renaming. In the nested splice, bindVars takes the empty list as an argument as there are no variables bound in the inner quotation. However, the renaming environment is computed as ds1_d23A ++ ds_d23z ++ [], which contains the renamed variable 6989586621679012506, which happens to occur in the quotation. As the expression containing quotations is elaborated, the environment is persisted to any quotation in the same static scope so that all variables are renamed appropriately.

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

Since GHC 8.0.1, the recommended way to implement Lift has been with the `DeriveLift` extension. The current implementation still works by explicitly generating the untyped representation terms which makes it harder to modify to support the CORE expression representation we have described. In the GHC 8.10 release, `DeriveLift` will be implemented in terms of quotation brackets (Theriault, 2019) which means that any derived instance will work seamlessly with this modified backend.

4.1.2 Modifications to Typechecking

Now that the principles of the representation have been established the hard work of modifying the typechecker can begin. There are two main modifications needed to the typechecking pass, firstly the CodeC constraint forms needs to be introduced in accordance with the specification in Section 3.2.1. Secondly, type variables also need to become level aware so LiftT constraints can be used to make type variables used inside quotations relevant. Before getting to those changes we will reflect further on how modifying splices to run later in the pipeline already improves the typechecking experience.

Improved Type Inference The effect of moving splices to run later in the compilation pipeline, after typechecking has finished, means that type inference for programs using typed quotes and splices is more robust. In some situations, as highlighted by the examples in Figure 4.2, the ability to delay type checking a splice can lead to more programs being accepted as the type is inferred from the wider context.

In particular considering the second example, the failure message from the existing implementation is:

```
F.hs:9:17: error:
    * Ambiguous type variable 'a0' arising from a use of 'codeFromString'
      prevents the constraint '(CodeFromString a0)' from being solved.
      Probable fix: use a type annotation to specify what 'a0' should be.
```

```
module CodeFromString where
import Language.Haskell.TH
import Language.Haskell.TH.Syntax
class CodeFromString a where
  codeFromString :: String → TExpQ a
instance a∼Char ⇒ CodeFromString [a] where
  codeFromString = liftTyped
module Main (main) where
import CodeFromString
main :: IO ()
main = do
  putStrLn $(codeFromString "example string")
    -- doesn't work
  print $ $(codeFromString "example string") == "example string"
    -- doesn't work
  print $ $(codeFromString "example string") == ("example string" :: String)
    -- works.
  print $ ($(codeFromString "example string") :: String)
          == ("example string" :: String)
```

Figure 4.2: Failure of type inference when using top-level splices in GHC 8.10

```
      These potential instance exist:
        instance [safe] (a ~ Char) => CodeFromString [a]
          -- Defined at CodeFromString.hs:10:10
    * In the expression: codeFromString "example string"
      In the Template Haskell splice $$(codeFromString "example string")
      In the first argument of '(==)', namely
        '$$(codeFromString "example string")'
  |
9 |     print $  $$(codeFromString "example string") == "example string"
  |                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

This is indicative of the fact that previously the constraints arising from each top-level splice were eagerly solved without considering the surrounding context. Now that top-level splices are typechecked in much the same way as normal syntax, and all constraints are solved at the same time at the top-level of the definition, the metavariable a is no longer ambiguous as it is fixed by the second argument to $(==)$. In general now typechecking of top-level splices is a bit less ad-hoc, type inference is improved.

4.1.3 Implementing CodeC

For a sound treatment of constraints, the CodeC constraints form was introduced in the formalism section. In this section, the implementation specifics and the changes that were needed to other parts of the constraint solving process are discussed.

There are three steps in the implementation of CodeC.

- How to solve wanted CodeC constraints.
- How to use given CodeC constraints.
- Enforcing level invariants in the constraint solver.

GHC's constraint solver is an implementation of OutsideIn(X) (Vytiniotis et al., 2011). There are two primary types of constraints, wanted and given. Wanted constraints are constraints which arise from the use of constrained functions. Wanted constraints must be solved by the constraint solver in order to complete the program. Given constraints are information provided by the user that the constraint solver can use in order to solve wanted constraints. Given constraints are usually provided by a method signature or bound by a GADT constructor. The final piece of the OutsideIn(X) story is implication

constraints. An implication constraint can be used to locally introduce given constraints. They also provide a location where solved wanted constraints can be stored as evidence. This is useful for us because the exact binding position of evidence is now important. Certain evidence will need to be bound inside specific quotes and splices. Therefore a new implication constraint is created to surround each quote and splice so solved constraints are bound at the correct level.

The implementation strategy for CodeC is roughly as follows:

- Internally modify the constraint representation so that each constraint is introduced at a specific level. The level of a constraint is the level it will eventually be bound at.
- Introduce implication constraints at the quote and splice points to give appropriate places to bind evidence.
- Augment the constraint solver to solve the new in-built CodeC form by applying the rules from the formalism.

Wanted Constraints Wanted constraints are solved in a similar manner to existing built in constraints such as Typeable. For a wanted constraint CodeC c at level k then the constraint is solved by a given constraint c at level k + 1. The evidence for a wanted CodeC constraint is a quoted variable which represents the evidence for c. As the constraint c is from level k + 1 then the evidence variable is guaranteed to be bound at level k + 1.

```
qshow :: CodeC (Show a) ⇒ Code (a → String)
sshow :: Show a ⇒ a → String
sshow = $(qshow)
```

In this example, the usage of qshow inside the top-level splice requires the wanted constraint CodeC (Show a) at level 0. The context of sshow provides the given constraint Show a at level 1. Therefore the in-built constraint solving rules solve the CodeC (Show a) constraint using the Show a constraint by forming a quotation which wraps the evidence variable.

Given Constraints Most given constraints in GHC are used as-is, they are not rewritten by the constraint solver. For example, if a user specifies that a Show a constraint is available then the constraint solver will just use that information to solve

a Show a wanted. For a CodeC c given at level k, the constraint solver will use this information to also solve a c wanted at level $\mathsf{k} + 1$ and this is implemented by rewriting the CodeC c given into a c given. This transformation is performed by using a splice, as guided by the elaboration rules in the formalism.

Therefore both wanted and given constraints will be simplified in order to remove the CodeC constraint form so that eventually the constraint solver is left with a set of levelled constraints without any mention of CodeC. The rewriting is confluent as each rewriting step acts to remove a CodeC layer.

Generalisation of Residual Constraints At the end of the constraint solving process there may be unsolved constraints. Usually during generalisation the type inference engine will infer a type with a context containing these constraints. Now all the constraints are also level aware, the generalisation process needs to account for levels. The top-level of a program is at level 0, therefore any constraints which are at level 0 can be generalised as normal. A constraint at level k where $k > 0$ can be generalised to k applications of CodeC. A constraint at level k where $k < 0$ is unsolvable by any rules and should be reported as unsolvable to the user.

For example, the inferred type of qshow is $(\mathsf{CodeC}\ (\mathsf{Show\ a}), \mathsf{LiftT\ a}) \Rightarrow \mathsf{Code}\ (\mathsf{a} \rightarrow \mathsf{String})$ because the usage of show inside the quotation causes a wanted constraint for Show a at level 1. The point of generalisation is at level 0 so type inference infers the context $\mathsf{CodeC}\ (\mathsf{Show\ a})$.

$$\mathsf{qshow} = [\![\ \mathsf{show}\ ]\!]$$

Location of Evidence Wanted constraints in GHC are traditionally solved by replacing the location of the evidence by a variable which is later let-bound in an outer scope when the constraint solver works out what the variable should be bound to. For CodeC constraint this scheme causes a number of issues because let binding a variable inside vs outside a quotation has a different meaning. Therefore we need to be very careful to make sure that if evidence is let bound then it is bound at the correct level, otherwise the generated program will have an out-of-scope dictionary.

For some programs, there is no sensible place to let-bind a dictionary. For instance:

$$\mathsf{numGen} :: (\mathsf{LiftT\ a}, \mathsf{CodeC}\ (\mathsf{Num\ a})) \Rightarrow \mathsf{Code\ a}$$
$$\mathsf{numGen} = [\![\ 5\ ]\!]$$
$$\mathsf{numInt} :: \mathsf{Code\ Int}$$
$$\mathsf{numInt} = \mathsf{numGen}$$

numInt is a specialised version of numGen where the CodeC (Num a) constraint is solved by the top-level instance for Num Int. The idealised desugaring for numInt would involve directly quoting the dNumInt : Num Int dictionary.

$$\mathsf{numInt} = \mathsf{numGen}\ \mathsf{dLiftInt}\ [\![\ \mathsf{dNumInt}\ ]\!]$$

Given the way the constraint solver works though, compositionally, the way the CodeC (Num Int) constraint is solved is by first emitting the wanted constraint Num Int at level 1 which is then solved by the top-level instance witnessed by dNumInt. Under normal operation the dNumInt constraint would be bound at the top-level of numInt and result in the following stage-incorrect program.

$$\mathsf{numInt} = \textbf{let}\ \mathsf{dInt} = \mathsf{dNumInt}; \mathsf{dLift} = \mathsf{dLiftInt}\ \textbf{in}\ \mathsf{numGen}\ \mathsf{dLift}\ [\![\ \mathsf{dInt}\ ]\!]$$

However, given our invariant that the evidence for a constraint must be bound at its level, where is the appropriate place to bind the Num Int evidence? There is no context which is at level 1 in the definition of numInt so we will have to write a quote in order to place the evidence.

The solution to this problem is to directly write the evidence which satisfies the CodeC constraint directly into place where it is needed. This ensures that all the levels align correctly and the constraint solver does not create incorrectly levelled let bindings. The mechanism to achieve this is the introduction of *expression holes*. An expression hole is represented by a new constructor to the core data type which contains an IORef which is written to when the value is known that should fill this hole. Once all the holes are known to be filled, the syntax tree is traversed and each hole replaced by its contents.

In this particular case, the expression holes are generated and filled during type checking. The only place where expression holes are created is in the solving of CodeC constraints. When the constraint solver works out what evidence is needed to solve the constraint then it directly writes the evidence into the hole rather than the usual mechanism of creating let bindings. Therefore by the time the metavariables in the expression are instantiated, the expression holes can be replaced by the actual expressions.

This treatment is similar to the usage of *coercion holes* which are used internally in the constraint solver to directly write kind equalities into the correct place during constraint solving. The motivation is similar here because there is no context where the correctly levelled evidence can be let-bound by the constraint solver so we have to write it directly into place.

There are still situations where CodeC constraints can be solved using let-bound variables but perhaps it is simpler to always write evidence solving CodeC directly into the created quotation. In particular, when used in a top-level splice, a given constraint is provided at the right level to solve the CodeC constraint.

```
polyNum :: Num a ⇒ a
polyNum = $(numGen)
```

In polyNum the CodeC (Num a) constraint is required at level -1, and therefore solved by the given Num a constraint at level 0. In this case the Num a constraint is already bound at level 0 and there's somewhere to create the let-binding used by the constraint solver to solve the wanted Num a constraint.

```
polyNum :: NumDict a → a
polyNum dNum = let dNum' = dNum in $(numGen ⟦ dNum' ⟧)
```

4.1.4 Implementing LiftT

In this section we describe some particular details about how LiftT constraints are emitted and some shortcomings in the implementation. To recall from Section 3.6.2, the purpose of LiftT is to make type variables relevant. This is necessary so that during run-time the type information can be used to construct a typed core expression. The design follows a similar design to the Typeable class.

The class LiftT is defined as:

```
class LiftT (t :: k) where
  liftTyCl :: TTExp
```

Note that by default GHC rejects this class as the type of liftTyCl doesn't contain the type variable t, therethere which instance to use can never be determined by a naked use of liftTyCl. Since this class is intended to be used internally, we can engineer that the correct type aruments are supplied to liftTyCl and hence the instances be solved.

The evidence for the class is a runtime representation of the type t which is this case is a serialised core type. The class is polykinded to support runtime evidence for any kind of types.

Emitting and Solving LiftT constraints During typechecking, the type of all identifiers used inside a quotation is required to be an instance of LiftT. This is an over

approximation as we only will eventually need LiftT constraints for free type variables which are used inside a quotation. At the point of quotation, any LiftT constraint for a closed type will be immediately solved but not used inside the quotation and the evidence can be discarded by the compiler. This over approximation is sometimes necessary because of the usage of metavariables during type inference. It is sometimes not known until the whole expression is typechecked what type a certain identifier will have but the decision about whether LiftT is needed or not also affects the type of the expression so the decision can't be delayed until after typechecking has established the type of all variables. It is also quite unpredictable from the source syntax how the elaborator will translate a source expression into core and where it will insert type annotations which may result in free type variables.

For example, in the quoted identity function, the type of id is $\forall a.a \to a$. During typechecking the type is instantiated by metavariables so that the LiftT constraint is emitted on the metavariable a, later when the variable is instantiated to either a skolem variable or a concrete type, either the constraint is solved by a local instance from the context (as in the type for qid) or automatically by built-in rules.

Substituting Skolems? After all the LiftT constraints have been emitted, the evidence is used to make a substitution which is applied to the quotation in the zonker. The type variables for which LiftT constraints were emitted are substituted for a splice which will later contain a representation of that type which can be used to construct the core expression. It was thought desirable at one stage to only substitute for skolem variables. Only substituing for skolem variables would mean that LiftT constraints would only need to be applied to user written type variables. Unfortunately this approach doesn't work generally in the case of type families. Therefore metavariables are also directly substituted for their LiftT evidence.

In particular, LiftT constraints are often applied to metavariables which are instantiated with some more complicated structure later on in type checking. If a metavariable is instantiated to a closed type then there would be no need to invoke LiftT at all, because the type itself could just be inserted into the quotation. On the other hand, if the metavariable is instantiated to a skolem variable then we definitely need to insert a type splice and use the LiftT evidence. Otherwise the quotation would end up with a free type variable. Finally, sometimes a metavariable will be instantiated to a compound type where more of the structure of the type is known but there are still free variables. In this case it seems desirable to use of much of the statically known structure of the type as possible but still substitute a splice point for the skolem variable.

For example, rather than replacing a metavariable with evidence for LiftT (a, b) you could use the (,) type directly in the program and insert splices for the LiftT a and LiftT b evidence. At this stage a and b would be skolem variables and we could maintain the principle that only skolem variables are substituted for LiftT evidence. This approach looks somewhat futile in the presence of type families.

For type families a similar decomposition does not fit in well with how the constraint solver works. The constraint solver will not attempt to solve a constraint such as LiftT (Index n xs) where Index is a type family. Therefore n and xs will not require LiftT constraints and so if they occur in the body then they will not be substituted for the correct evidence and end up appearing free in the resulting quotation.

These free variables are avoided by directly substituting metavariables for LiftT evidence rather than just substituting skolem variables (n and xs in this example) Unfortunately this often leaves unsightly LiftT (Index n xs) constraints which violates the principle that a user should only write LiftT constraints for free type variables. Soundness is preserved in this case by using the LiftT (Index n xs) evidence in the quotation rather than generating a program which contains an application of Index to two type splices for n and xs.

It seems more natural to only substitute for skolems and this intuition is strengthened as certain parts of elaboration expect certain values to have specific types. For instance, there is an assertion which checks that when constructing an application that the expression in the function position has a function type. This assertion fails for the simple example below:

```
failure :: String → Code ((String → Int) → Int)
failure s = ⟦ λk → k $(liftTyped s) ⟧
```

When k is introduced, a new metavariable for the type of k is also introduced. Later on this will be instantiated to String → Int but because the metavariable is introduced inside a quotation, the type of k will be replaced by a splice which will later be filled with the evidence for LiftT (String → Int). In this case the introduction of a splice is unnecessary as String → Int is a closed type. If the variable was instead instantiated to String → a for some skolem variable a then it would be necessary to substitute the a variable for a splice point.

This issue could be solved by engineering but as a general principle if you find yourself pushing too hard against built-in assumptions in GHC then your implementation is probably too complicated or wrong. The issues with type families point to the fact that relevance is not a semantic property of types but rather a syntactic property of type

variables and suggests that an approach using relevant type variables would lead to a smoother implementation.

At the moment the story behind LiftT is quite unsatisfying from a practical and implementation perspective. We were inspired by the success of Typeable but seeing as LiftT is emitted in many more places and is required for soundness, the constraint based approach looks doomed to be an unergonomic failure. Therefore in the future we are looking at other ways in which type variables can be dealt with.

For Typeable, the soundness of the program does not depend on ensure that all type variables have a runtime representation. The custom typeable solver just fails to solve constraints involving type families. We predict that any serious usage of Typeable would also run into similar ergonomic issues.

Explicit Type Variables Stage errors for explicitly mentioned type variables are solved in the same manner as value-level stage errors. Type variables can appear explicitly in explicit type applications (Eisenberg et al., 2016) and expression type signatures.

```
crossTy :: ∀a.Code (a → Int)          crossTy' :: ∀a.Code Int
crossTy = ⟦ const 0 :: a → Int ⟧      crossTy' = ⟦ get@a ⟧
```

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

```
crossTy :: ∀a.LiftT a ⇒ Code (a → Int)
crossTy = ⟦ const 0 :: $(liftT@a) → Int ⟧
```

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

4.1.5 Relaxing the Staging Restriction

The staging restriction can also be lifted easily thanks to the new representation form. The restriction was introduced in Section 2.1.5 and states that only identifiers defined in another module may be used inside a top-level splice.

The staging restriction is more of a practical disadvantage in the typed setting because it is easier to define ad-hoc higher-order code generators using other higher-order

combinators. Therefore it becomes a hindrance to have to move these ad-hoc definitions into a new module just when we want to give a name to a new construct.

Why does the staging restriction exist to begin with? In order to execute a top-level splice the function definition needs to be compiled to bytecode and executed. Any identifier which is used inside a top-level splice needs to be likewise compiled before the splice can be run. In the case of a top-level identifier bound in another module, the module is imported and therefore guaranteed to be compiled to object code already. The interpreter can load and use the compiled version of the function. For identifiers defined in the same module as the splice appears, then the definition will not already be compiled to object code. Therefore some other strategy would be needed in order to make sure the relevant definitions are compiled before the splice is executed.

The decision to move the execution of splices until after the module has been elaborated makes performing the necessary dependency analysis straightforward.

1. Elaborate the module, but do not execute top-level splices yet, collect top-level splices as elaboration proceeds.
2. Perform dependency analysis of the elaborated program and top-level splices.
3. In dependency order process each splice in turn. Compile all the dependencies of the splice and extend the environment of the interpreter.
4. Run the top-level splice in the extended environment.

The implementation of running splices also makes use of expression holes. When the syntax tree is traversed to find any splices which need to be run, each splice is replaced by an expression hole where the result of the splice will be inserted. Using an expression hole means that the top-level splices can be placed as normal nodes in the dependency graph and executed and inserted without having to manipulate the syntax tree again. The pattern of traversing, analysing, running and filling means that each step of the process is well-defined and easy to implement separately.

```
data SpliceNodeType = SpliceNode (CoreExpr, CExprHole Id)
                    | DefnNode (Id, CoreExpr)
```

The SpliceNodeType represents a node in the dependency analysis. A node is either a top-level definition represented by its name and definition, or a top-level splice represented by the contents of the splice and a hole to be filled in. The dependency analysis orders splices and definitions relative to each other. The graph is then thinned

by keeping the top-level splices as roots, there is no need to compile top-level definitions which do not appear in splices, and then traversed from the leaves compiling each dependency in turn. Finally, a top-level splice is run with all the dependencies in the environment and the result of running the splice is placed into the expression hole.

Because we can delay running splices until the core level, where it is easy to perform this kind of dependency analysis, the implementation only amounts to about 200 lines of extra code but fixes a GHC issue which has been open for 12 years, at least for the case of typed splices.

Unfortunately the same strategy can't be used for untyped splices which expose more about the internal state of the typechecker and hence have to run much earlier in the compiler. Dependency analysis at the source level is performed already but it is more difficult to translate this dependency analysis into instructions about which top-level definitions need to be compiled. For example, during the renaming stage the dependency analysis can't account for where any implicit evidence will be bound whether locally or to what identifier. So when the top-level splice is compiled already in the renamer, it isn't obvious what else would need to be compiled in order to provide the right definitions.

4.1.6 A Combinator Based Elaboration?

MetaOCaml and Typed Template Haskell have traditionally followed combinator based elaborations. The representation term is built compositionally by using combinators designed in order to construct the correct syntax tree. In this implementation, the combinators would be used to build the representation of IfaceExpr. One primary advantage to the combinator approach is that it makes performing substitution easier, there would be no need for all these different environments. The reason we wanted to avoid the combinator approach is that the generated program ends up being quite a bit larger than the quoted expression. Once you also take into account all the additional information introduced by elaboration, it seemed that the generated programs could end up as quite large terms. This increase in program size causes modules which contain a large number of quotations to become quite slow to compile. The effect is more noticeable with Typed Template Haskell rather than the untyped variant because quotations are used extensively when writing programs in the typed style but much more rarely in the untyped style.

I also wanted to avoid the large engineering effort needed to design all the combinators and implement the elaboration from core expressions to the combinator format. For the source syntax tree this module is over 2000 lines long and there are hundreds of

combinators. Whenever the source syntax tree is modified, it's necessary to update the elaboration and Template Haskell AST, there is a lot of redundant work. The version using the existing serialisation functions will always be up to date and is designed to create small, efficient representations of terms.

Experimenting in future with the combinator-based approach is still possible, the details about the precise representation form are separate from any concerns to do with implicit evidence and so on.

Status of Implementation The implementation of the ideas in this section is completed as a fork of GHC-8.10. The fork has been used for the following case study but there are still a number of known bugs related to the interaction of CodeC constraints and existing parts of the constraint solver. It is hoped that in future months the implementation can be completed to the level necessary for the changes to be merged into the main compiler.

4.2 Microbenchmarks

In this section we include a few microbenchmarks which test two specific important parts of the implementation for performance. No great effort has been taken to optimise the current implementation but it is still interesting to see how modifying the representation can affect the speed of compilation for certain programs. These benchmarks give a good foundation for improving the performance of the implementation in future.

In particular we are interested in two aspects of performance.

- The time taken to compile a module which contains a lot of quotations or a single large quotation.
- The time taken to generate a program.

In both situations, we expect the new representation to improve performance. In the first case, compiling modules containing a number of quotations can currently be quite slow because the combinator elaboration results in a large expression which the optimiser can take a while to optimise. In the second case, depending on the program, the untyped representation form can cause type inference to dominate compile time. By using an already typed representation the cost of type inference will be be paid at the point the

program is quoted rather than once when the quotation is typechecked and again when the expression is inserted.

4.2.1 A Big Quotation

The first benchmark is a single quotation which contains a list of 100 identical terms:

```
bigq :: Code [Maybe Int → [Int]]
bigq = ⟦ [λx → case x of {(Just { }) → [ ]; Nothing → [1, 2, 3]}, ...] ⟧
```

This benchmark is intended to test how much faster the new way of compiling quotations is as a big combinator expression will not be created during elaboration. The programs are compiled firstly with `-O0` and then also with `-O2` to see how much of an impact the optimiser has on compilation time.

	Time (s) `-O0`	Time (s) `-O2`
Old	1.2	1.7
New	0.3	0.9

The new implementation is about 2 and 4 times faster for this example. Initial measurements indicate that the code generation time in the `-O0` dominates compile time for the old combinator approach.

4.2.2 Many Quotations

The second benchmark contains 100 identical small quotations in the form of `f1`:

```
f1 :: Code (Maybe Int → [Int])
f1 = ⟦ λx → case x of {(Just { }) → [ ]; Nothing → [1, 2, 3]} ⟧
```

The module is compiled with both `-O0` and `-O2` again.

	Time (s) `-O0`	Time (s) `-O2`
Old	1.0	1.7
New	1.0	3.4

Using `-O2` with many quotations results in worse performance currently. This might be because of the large amount of redundant `LiftT` evidence generated by the current implementation. Further investigation is required.

4.2.3 A Big Generated Program

The next test is to generate a big but simple program. In order to do this we implement a simple staged interpreter (Figure 4.3) and use it to compile a program which will result in a large Haskell expression. This test is supposed to stress the main mechanisms for loading expressions back into the compiler after the meta programs have been executed.

The test program creates an expression which computes whether a number given by the user is less than a static parameter k by generating a program which performs an equality check on all the numbers less than k. The code generated by this test will contain nested case statements so it's also a test to see how well the optimiser can perform the clean-up operations described in Section 5.1, in particular the transformation which collapses nested cases into a single case expression.

```
runTest :: Int → Expr
runTest k = Lam "x" (go k)
  where
    e n k = If (Eq (Var "x") (Num n)) (Cond True) k
    go 0 = e 0 (Cond False)
```

The benchmark computes a program which checks if number is less than or equal to 1000.

```
test x = $(compile (runTest 1000 `App` (VQ [| VNum x |]) []))
```

For this benchmark the importance of the clean-up transformations becomes evident. Compilation with `-O0` is *slower* than with optimisation because the generated program is so big that code generation takes longer than the time it would have taken to optimise it.

	Time (s) `-O0`	Time (s) `-O2`
Old	2.5	1.9
New	3.6	1.7

Inspecting the generated program, there is a certain threshold (around 35) where the optimiser starts failing to optimise the program into a single case expression as desired. There are also differences in how well the code generated by the old and new approach optimise which would be interesting to understand.

```
module Interp where

import Language.Haskell.TH
import Data.Maybe
import Data.Function

data Expr = If Expr Expr Expr
          | Num Int
          | Eq Expr Expr
          | Cond Bool
          | Var String
          | Lam String Expr
          | App Expr Expr
          | Fix Expr
          | VQ (TExpQ Val)

data Val = VNum Int | VBool Bool | VClos (Val -> Val) | Error String

type Env = [(String, TExpQ Val)]

compile :: Expr -> Env -> TExpQ Val
compile (Num x) _ = [|| VNum x ||]
compile (VQ e)   _ = e
compile (Cond x) _ = [|| VBool x ||]
compile (Eq e1 e2) env  =
  [|| case ($$(compile e1 env), $$(compile e2 env)) of
         (VNum x, VNum y) -> VBool (x == y)
         _ -> Error "Expecting Int" ||]
compile (If e1 e2 e3) env =
  [|| case $$(compile e1 env) of
        VBool b -> if b then $$(compile e2 env)
                        else $$(compile e3 env)
        _ -> Error "Expected Bool" ||]
compile (Var s) env = fromJust (lookup s env)
compile (Lam v e) env =
  [|| VClos (\x -> $$(compile e ((v, [|| x ||]) : env))) ||]
compile (App e1 e2) env =
  [|| case $$(compile e1 env) of
        VClos c -> c $$(compile e2 env)
        _ -> Error "Expected Closure" ||]
compile (Fix e1) env =
  [|| case $$(compile e1 env) of
        VClos c -> fix c
        _ -> Error "Expecting closure" ||]
```

Figure 4.3: A Staged Interpreter

4.2.4 A Complicated Generated Program

Of course the primary reason for this endeavour has been soundness but we also claimed that the new implementation would be faster if the generated program took a long time to compile due to type inference. Therefore we also include the classic test-piece of type inference to check the time taken to perform a splice of a program which takes a long time to infer the type for. The working hypothesis is that in the new implementation the type will already have been inferred so it should be much faster.

The test program is the composition of many dup functions which results is a very large inferred type. Type inference of this program normally takes about 4 seconds.

```
def :: ∀a.(LiftT a, _) ⇒ _
def = ⟦ let dup :: b → (b, b)
            dup = λx → (x, x)
    in dup ∘ dup ∘ dup ∘ dup ∘ dup ∘ dup ∘ dup ∘ dup ∘ dup
       ∘ dup ∘ dup ∘ dup ∘ dup ∘ dup ∘ dup ∘ dup ∘ dup ⟧
```

	Time (Quote) (s)	Time (Splice) (s)	Time (Total) (s)
Old	3.6	5.6	9.3
New	17.1	11.3	31.9

The benchmark is separated into two parts, the time taken to compile just the quote and the time taken to execute def using a top-level splice.

The new implementation is actually a lot slower for this program despite the fact that it does less work when splicing the expression. Further analysis reveals that the generated core program has about 3,000,000 types in it after typechecking has made types explicit. Therefore I conjecture that the cost of serialising and deserialising this amount of elaboration is why the new implementation is much slower for this example. The bytestring used to represent the fully elaborated term is approximately 3×10^8 bytes long and the interface file is about 2.4 megabytes big. Code generation is the part of compilation which dominates the compile time.

It seems like it would be worthwhile to investigate ways to reduce the amount of explicit type information present in a core program as suggested by Jay and Peyton Jones (2008). Savings can also be made by deduplicating types using a hash consing based approach, a similar idea is already used to save space in `.hie` files.

4.2.5 Another Complicated Generated Program

Programs using type families can also cause simple looking programs to generate very big core terms due to the number of coercions. This benchmark is taken from a GHC ticket which described the problem of compilation taking a long time when using type families. I conjectured that it would be a good test for my implementation as the work of computing the type family would only be performed once at the point of quotation rather than twice, also at the splice site.

This program uses the Replicate type family and in order to type check the example the equality Replicate 1000 ()∼Replicate 1000 () has to be computed.

```
data Nat1 = Zero | Succ Nat1
type family Replicate (n :: Nat1) (x :: a) :: [a]
type instance Replicate Zero x = []
type instance Replicate (Succ n) x = x : (Replicate n x)
```

Unfortunately the generated program contains 100,000,000 coercions as evidence for the type family computation. This causes serialisation and deserialisation to take several minutes.

```
{terms: 10, types: 370,055, coercions: 100,360,043, joins: 0/3}
```

Unlike types, it should be possible to erase coercions if you don't want to run the core linting pass. There is ongoing work to implement this as a mechanism to speed up the compiler when typechecking programs using type families independently of this work.

4.2.6 Conclusion

The result of the microbenchmarks indicates that there is still work to be done improving the performance of the implementation. These examples provide some food for thought about different challenges that will need to be considered in the future. In particular the biggest issues seems to be that the elaborator can introduce a large amount of evidence even for simple programs so the compilation strategy needs to be careful to be efficient when serialising these large programs.

Chapter 5

Inlining and Specialisation

In this chapter we we move onto some concepts from automatic partial evaluation. The material in this chapter is not novel, but the operation of GHC's automatic optimiser is complicated and not well documented. Peyton Jones and Marlow (2002), Santos (1995) and Jones (1995a) provide the foundations but in the intervening years the threshold and heuristics used have been refined. This chapter provides a more up-to-date explanation of the inner workings of the inliner and specialiser.

Discussing inlining and specialisation motivates the central motivation for the thesis: If you want to write a high-level program with guarantees about how it is optimised, then you should use a multi-stage language rather than relying on automatic optimisers. A secondary motivation is that automatic optimisers are still useful in multi-stage languages to perform simple code clean-up operations in the generated program. We highlight some reliable and convenient optimisations which help improve generated code. Finally, a case study is provided which demonstrates how inlining and specialisation can be used with great care to write high-level libraries which are amenable to automatic optimisation. This story though, is far from perfect and requires modification to default thresholds to work for most cases. The case study is then modified in Section 5.5 to the staged style using the contributions from the previous chapters.

5.1 Core Language

In this section we will describe the core language which all Haskell programs are compiled to by GHC. Firstly, by describing how some of the optimisations which operate on the core language it will become evident that automatic optimisation of

programs is not a good technique to rely on for designing high performance libraries. Secondly, the core language was used in Chapter 4 as the representation for quotations. Automatic optimisation is still useful in multi-stage languages as generated programs can contain simple optimisation opportunities such as β-reduction.

5.1.1 The Core Language

The Core language is based on a variant of System F called FC (Sulzmann et al., 2007) which extends the polymorphic lambda-calculus with type equality coercions. The coercions witness non-syntactic type equality. Therefore the language contains value abstraction and application, type abstraction and application, let expressions, data constructors and case elimination, and casts. Both value and type abstractions are represented using the Lam and App constructors, internally the same type is used to represent both types of binders.

The definition of the core language of GHC is found in Figure 5.1. All source programs are compiled into this core language, a core program is a sequence of core bindings. All optimisation passes are in terms of this core language rather than the source programs. The much simpler structure of this language makes optimisation passes easier to specify and implement.

5.1.2 Basic Optimisations on Core

There are several optimisations performed on a core program which are always beneficial as they correspond to a normal reduction step but performed during compilation rather than at runtime and reduce program size.

These kinds of transformations are important to distinguish from the heuristic based optimisations such as inlining and specialisation because we can rely on them happening. When writing staged programs, an automatic evaluator is still necessary to clean up a generated program. The generation process can remove recursive loops but will usually end up with a program which contains some of these basic optimisation opportunities so in order to get to the final program a swift pass over the generated program is necessary to remove these redundancies.

For a more complete description of the different optimisations which are performed on the core language see Santos (1995). We highlight a few of these reductions here in order to give a flavour of some of the clean-up operations which the compiler can be expected to perform.

```
data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type Type
  | Coercion Coercion
type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)
data AltCon
  = DataAlt DataCon
  | LitAlt Literal
  | DEFAULT
data Bind b  =  NonRec b (Expr b)
             |  Rec [(b, (Expr b))]
```

Figure 5.1: Core Language Syntax

β-reduction

The simplest optimisation is to perform one-step of β-reduction and reduce a lambda by substituting the argument into its body.

$$(\lambda x \to e)\ y \rightsquigarrow e\ [x\ /\ y]$$

Redexes can often appear in generated code if you have a code generator which returns a function type. For example, by composing the generator for the identify function with a quoted unit value, the resulting program will be the identity function applied to the unit, which can be reduced by β-reduction.

$$\begin{aligned}
&qid = [\![\ \lambda x \to x\ ]\!] \\
&qunit = [\![\ ()\ ]\!] \\
&qres = \$([\![\ (\$(qid)\ \$(qunit))\ ]\!]) \\
&\quad \rightsquigarrow (\lambda x \to x)\ () \\
&\quad \rightsquigarrow ()
\end{aligned}$$

Case of Known Constructor

Case expressions can be reduced if the scrutinee evaluates to a known constructor. For example, when casing on a boolean, if the scrutinee is known to be True then the case expression can perform one step of reduction and select the first branch.

```
case True of
  True → False
  False → True
 ⇝
False
```

When writing a compositional code generator it is common to parametrise the generator on the value of the scrutinee.

```
genNot :: Code Bool → Code Bool
genNot x = ⟦ case $(x) of
               True → False
               False → True ⟧
```

Then if genNot is applied to ⟦ True ⟧ then the generated program will result in a case of known constructor optimisation opportunity.

Case-of-case

If a case appears in the scrutinee position then a program is often simplified by reassociating the expression so that the case is pushed into both branches. In the composition of not ∘ not, the transformation unveils an opportunity of case of known constructor which allows the program to be simplified to the identify function.

```
not (not x) ⇝
case (case x of {True → False; False → True}) of
  True → False
  False → True
 ⇝
case x of
  True → case False of {True → False; False → True}
  False → case True of {True → False; False → True}
 ⇝
```

```
case x of
  True → True
  False → False
```

Any program which generates cases will often end up using cases in the scrutinee position so this automatic rewriting can produce a much better final result.

Nested Case

Finally, collapsing multiple case expressions which scrutinise the same value can result in a simpler core program. In the following example, the nested case expression could have been written as a single case expression with three branches.

```
case x of
  'a' → False
  _ → case x of
    'b' → False
    _ → True
 ⇝
case x of
  'a' → False
  'b' → False
  _ → True
```

This situation ends up being quite common when generating programs as instead of directly generating a n-ary case statement to match on all alternatives, it is often more compositional to generate a sequence of nested cases which are then collapsed by the compiler. For instance, the qmatchOne function is a generator that generates a program which checks to see if a character matches one of a statically provided list of characters.

```
qmatch :: Char → Code (Char → Bool)
qmatch c = ⟦ (==c) ⟧
qmatchOne :: [Char] → Code (Char → Bool)
qmatchOne cs = ⟦ λc → $(foldr (λa r → ⟦ $(qmatch a) c ∨ $(r) ⟧) ⟦ False ⟧ cs) ⟧
```

The result of calling qmatchOne ['a', 'b', 'c'] is:

```
λc → (=='c') c ∨ (=='b') c ∨ (=='a') c ∨ False
```

which will be translated by the compiler into nested case expressions before eventually ending up as a single case by the nested case optimisation.

Without a special compiler primitive then generating a case expression which matches on a certain number of constructors is not usually possible in multi-stage programming languages. The crux of the issue is that multi-stage languages have constructs for generating expressions and the branches of a case expression are not expressions.

Summary

This family of simplifications are usually sufficient to clean up generated programs so that the result is similar to the result of writing a hand-written version. They have the advantage of being simple to specify and hence can be relied upon. It is usually possible to write a staged program as to not generate administrative redexes which need to be β-reduced by the compiler but having the additional clean-up steps gives some more flexibility. The role of staging is often to provide the means to unroll recursive functions in a directed manner so that all these simple optimisation opportunities become evident and the optimiser can finish off the job.

5.2 Inlining

The most critical optimisation for an automatic optimiser is inlining. Inlining is the enabling optimisation which replaces a function name by the definition of that function. After a function definition has been inlined, new optimisation opportunities are now evident to the optimiser such as the previously discussed β-reduction and know-case elimination optimisations.

The difficulty with inlining is that on its own it is not a beneficial code transformation. Inlining a function which does not unlock any further optimisation possibilities is wasted work which increases code size. Therefore any automatic partial evaluator must decide at what thresholds functions must be inlined and what factors about the call-site must be taken into account when making an inlining decision. Unfortunately designing a library which relies on these thresholds and decisions is inadvisable. Small changes to the source program can have big changes on which thresholds are breached and understanding how your changes will affect the thresholds is very difficult.

In this section we will describe the parameters the GHC's inliner (Peyton Jones and Marlow, 2002) operates under.

`-funfolding-creation-threshold` Functions under this size will be given unfoldings and can therefore be inlined.

`-funfolding-dict-discount` The argument discount if the argument is a dictionary.

`-funfolding-fun-discount` The argument discount if the argument is a function.

`-funfolding-use-threshold` The threshold for deciding whether to inline a function at the call-site. The size is after the argument discounts have been applied.

Figure 5.2: Flags controlling inlining

5.2.1 Inlining Thresholds

When each function is defined, the body of the function is analysed and metrics calculated about under what conditions the compiler considers it worthwhile to inline the function.

What is size? The size of an unfolding is also a heuristic measure with the understanding that inlining a lot of "big" definitions will lead to a very "big" program which will take a long time to optimise.

When is a Function Not Inlined?

It is easier to start with the situations where a function is not inlined.

- Self-recursive functions are never inlined.
- Functions which are "too big" are not inlined. The threshold is controlled by the `-funfolding-use-threshold` flag.
- Functions which are known to never terminate are not inlined.

The first two restrictions are conservative aimed at reducing the chance of the inliner looping or creating very large intermediate programs. Partial evaluators of the past have inlined recursive functions under precise conditions. In GHC the inliner never inlines a recursive definition which is conservative and predictable. Bottoming functions are not inlined as they are lifted to the top-level and inlining them would undo this work.

Loop-Breakers In order to prevent recursive functions from being inlined, some care is needed to deal with mutually recursive groups. For each mutually recursive group

of functions dependency analysis is performed and then a single loop-breaker is chosen which breaks any cyclic dependencies. The loop-breaker is never inlined, but other functions in the group can be. The loop-breaker is decided based on a heuristic which attempts to select the member of the group which would benefit least from being inlined. The choice of loop-breaker affects how a program is optimised (den Heijer, 2011) but the programmer has little control about which member of a recursive group is chosen.

This explains why self-recursive functions are never inlined as they are always chosen as the loop-breaker for their recursive group of size one.

When is a Function Inlined?

For each function definition the body of the function is analysed and the unfolding guidance is calculated. There are two varieties of unfolding guidance, UnfWhen and UnfIfGoodArgs. For normal identifiers, UnfIfGoodArgs stipulates under what conditions the compiler thinks inlining would be beneficial.

UnfIfGoodArgs calculates this using three factors.

1. The size of the definition.
2. A discount for each argument which has already been evaluated to WHNF.
3. A discount if the function appears in a scrutinee position.

The size of the function is taken into account to avoid creating program bloat. The argument discounts are used to encourage the inlining of functions which are guaranteed to unleash further optimisation opportunities. If an argument appears in a scrutinee position in the body of a function then inlining that function will cause the case to be reduced. Finally, if the function itself returns a constructor then inlining the function into a scrutinee position will enable the `case` to be eliminated.

These facts are all calculated by looking at the definition of a function. The size and discounts are unitless created solely for the purpose of informing the inlining heuristic.

Call Site Heuristics At the call site, the unfolding guidance is used to decide whether to inline the function call.

1. The expression is “work-free”, that is, we are not going to duplicate a lot of work by inlining it.

2. The unfolding is small enough, as calculated using the guidance.
3. There is some perceived benefit to inlining into this context. For example, inlining into a scrutinee position.

The important take-home message from this description is that there are two halves to making an inlining decision. The definition site and call site, at the definition site certain criteria are set up which determine the decision about whether to inline at the call-site when more context is available.

5.2.2 Controlling Inlining

The compiler is naturally conservative when deciding whether to inline a function as if an incorrect decision is made, the resulting compilation can take a very long time and use a lot of memory. Therefore it is sometimes necessary for experts to direct the optimiser in more precise ways using compiler options of function-specific pragmas. These mechanisms can be used to control whether and how non-recursive functions are inlined. Recursive functions will still never be inlined.

Pragmas The most well-known and dangerous way of controlling inlining is by using the `INLINE` pragma. By annotating a function with `INLINE` you inform the compiler to always inline the function regardless of its size or calling context.

Not only does the `INLINE` pragma force the function to be inlined at every call site, it also changes what is inlined. If a function is inlined naturally then the optimised unfolding will be used to replace the function name. With the pragma, instead the unoptimised unfolding is used which amounts to an unoptimised version of the user-written right-hand side of the function being inserted directly.

On the other hand, the `INLINEABLE` pragma is far more benign. Instead of influencing the inlining decision from the call site, the pragma only overrides the `-funfolding-creation-threshold` option and makes sure the unfolding for a definition is included in the interface file. Like the `INLINE` pragma, it is the unoptimised definition which is included. Despite its name the pragma is mainly used for ensuring the definition of recursive functions is made available in interface files so that they can be specialised across modules.

In addition to these positive instructions there is also the negative instruction, `NOINLINE`, which indicates that a function should never be inlined. The usual reason for using `NOINLINE` is in conjunction with `RULES` pragmas (Peyton Jones et al., 2001).

`INLINE[k]` Do not inline until phase k and then be very keen to inline

`NOINLINE[k]` Do not inline until phase k and then inline as normal

`INLINE[~k]` Be keen to inline until phase k but then do not inline it

`NOINLINE[~k]` Inline normally until phase k but then do not inline it

Figure 5.3: Meanings of phase annotations

All three of these pragmas can be augmented with phase annotations. Phases count down from the initial starting phase. By default this is two, so therefore there are phases two, one and zero. Each phase corresponds to one run of the main simplifier in the optimisation pipeline. Adding a phase annotation to an `INLINE`, `INLINEABLE` or `NOINLINE` pragma affects which of these phases the pragma is active in.

These phase annotations provide a way to loosely control the order definitions get inlined into each other in the attempt to remedy the situation where function bodies become too big to inline due to other definitions being inlined into the body. For example, if your function f contains a lot of other functions which when inlined would make f itself too large to inline under the default heuristics then f could be marked with `INLINE[2]` so that it is inlined before its body.

It is also worthwhile when dealing with rules to mark definitions which you want to interact with rules with `INLINE[0]` so that they are not inlined until the last phase and the rule matching facility has the greatest opportunity to rewrite expressions before they are removed by inlining. Figure 5.3 describes how the phase annotations interact with the `INLINE` and `NOINLINE` pragmas.

The greatest issue with the `INLINE` pragma story is that a decision is made at the definition site about how each function should behave at the call site. As a first approximation it is desirable to inline a function if it is applied to a statically or partially statically known argument (which is the purpose of the argument discounts), whether this will happen is not known at the definition site. Therefore a certain amount of hubris is required by a library author to ever add an `INLINE` to a definition as they predict, that without reservation, the decision will be beneficial to users.

At the call-site the magic function inline can be used to dictate that a specific call of a function should be inlined. For inline to work, the unfolding of the identifier the function is applied to has to be available. Using inline is sensible but quite fragile. At a call-site you know whether inlining a certain function is a good idea or not (consider the from∘to case from generic programming) but once the outer-most layer is inlined then you are

once again at the whim of the inliner.

Despite these dire consequences a common approach to attempting to optimise an application is to add `INLINE` pragmas to every function in a library. Sometimes this results in some improvement but usually just leads to your program becoming bloated and taking a long time to compile. In an open-world where anyone may use your function, marking it as `INLINE` is probably not the optimum decision in all cases. When this is combined with the complication of phases and rules, establishing when and what gets inlined in which phase gets more complicated.

5.2.3 Disadvantages to Relying on Inlining

There are two primary reasons why relying on the automatic inliner is a poor idea as a library author. Firstly due to the many different thresholds the inliner takes into account when trying to decide whether to inline a function or not, you can't tell easily from looking at a program how it will be optimised. At the local level, it's hard to work out whether a definition will trip over a size threshold or not which can have a catastrophic impact if a single function fails to inline when you expect it to. On the other hand, you decide to take matters into your own hands and use an `INLINE` pragma. At this point, the `INLINE` pragma may inline a function into another function which then causes it to become too big and trip a size threshold, also causing a failure in optimisation.

Considering a relevant example, the from and to functions from the Generic class are inverse to each other and not recursive. Therefore the composition from ∘ to should be optimised to the identity function if from and to are inlined. The difficulty with this suggestion is that both from and to are larger than the size threshold for most datatypes of a reasonable size. As the class methods are defined in a library, the only solution is to add a `INLINE` pragma to the class method definitions but not these big functions will be inlined into other functions potentially making them "big" as well.

Secondly, once the failure to inline a function has been identified there is no reliable way to modify your library in order to ensure the relevant function gets inlined in future. This is ignoring the difficulty in analysing that there has been a failure to begin with. The unfortunate way which experts work this out is by directly reading the core program and investigating parts which look unusual.

Once you have identified the problem, and fixed it in some manner, then the behaviour of the inliner is still liable to change between GHC releases. If you rely on the optimiser in a complicated way then in a future GHC release your program will break at some point.

It is possible to design libraries which use the optimiser in order to remove quite a significant amount of overhead (Magalhães, 2012; Kiss et al., 2018). However, this isn't without a significant effort looking at the generated program and making small changes to the source program which make the inliner perform in a certain way. It's hard even for an expert to understand how small tweaks can result in a very different outcome.

Others have resorted to extending the optimisation pipeline in domain specific ways. In particular, HERMIT (Farmer, 2015; Farmer et al., 2012) is a compiler plugin which allows a user to specify rewrites to be performed on the core programs. This has the advantage of providing a higher-degree of certainty to the optimisation process but still takes the user outside of the language itself to reason about performance. HERMIT has also been used to improve the performance of generic programming libraries (Adams et al., 2014).

What instead we desire is the ability to precisely control what will be inlined and in what order, under a well-defined semantics. The control about the structure of the generated program and what should be appear in it should be precisely controlled by the programmer rather than at the mercy of the optimiser. If some information is statically known, and can be eliminated, its elimination should be guaranteed. This is what multi-stage programming promises us.

5.3 Specialisation

Type class constraints (Wadler and Blott, 1989) in Haskell are implemented by a dictionary passing translation. A function accepting a type class constraint is passed a record which provides evidence for the constraint. Therefore despite the fact that by runtime, all instances have been statically resolved, the dictionaries are still passed to functions which can incur significant overhead.

Since type classes are ubiquitous, getting rid of this overhead is critical for any high-performance Haskell programs. Therefore one of the most important optimisations in the compiler is *specialisation* which rewrites functions to remove the overhead of passing a statically known dictionary (Jones, 1995a).

The process of specialisation proceeds in two phases. Firstly the body of the program is traversed in order to find functions applied to a statically known dictionary argument.

```
$dDictShowInt :: Show Int
```

```
f :: Show a => ...

g = ... f @Int $dDictShowInt ...
```

For each occurrence, a new top-level function is created whose arity is one less than the original function. The definition of the top-level function is the original function applied to just the static dictionary argument.

```
f_spec = f @Int $dDictShowInt
```

Finally a rewrite rule is emitted which rewrites the original function application to the new specialised version.

```
{-# RULE f @Int $dDictShowInt = f_spec }
```

Notice that we rely on `f` getting inlined in the definition of `f_spec`, but GHC will certainly do this due to how recursive overloaded functions are elaborated into a non-recursive function which takes the dictionary arguments and returns a recursive function the user wrote. For example the body of `f` will have structure similar to the following:

```
f :: Show a -> ...
f = /\ a . \dShow -> let f' = ... f' ...  in f'
```

The end result is that the statically known dictionary will no longer appear in a call to `f`, and its methods inlined into the body. Saving this indirection is a sure-fire way to improve performance and can unveil more optimisation opportunities based on the interaction of the class method and surrounding context.

Relationship with Inlining Specialisation is very closely related to inlining. Another way to implement specialisation would be directly inline the overloaded function at each call site where there was a statically known dictionary.

The advantage of specialisation is that it usually improves the program because the specialised function is applied to a statically known argument. In general inlining is a heuristic which may not succeed in removing any static information. The argument discount is a generalised version of the specialisation check.

Secondly, code bloat is avoided by processing a whole module at once. Specialisations are shared throughout a whole program so each function is only specialised for each

type once. This contrasts with inlining where the decision is made locally to a call-site and work is not shared between decisions.

Finally, recursive functions are specialised but not inlined. Therefore the only way to eliminate an argument from a recursive function is if the argument is a type class dictionary. Specialisation is mainly of use for recursive functions as for non-recursive functions inlining can play a similar role. However, for recursive functions, if they are not specialised then the overhead of passing the static argument will be paid on each iteration.

5.3.1 Issues with Specialisation

Despite the seemingly simple guarantee, statically knowing a dictionary argument has not been sufficient for specialisation to reliably remove the overhead of dictionary passing. In this section we will discuss several examples which have been reported by users of GHC where specialisation has failed to happen and how the optimiser has been modified in order to account for this unanticipated possibility.

In Section 3.6.5 we will propose an alternative to the dictionary passing translation which obsoletes the need for specialisation by using multi-stage features.

Cross-Module Specialisation

As most projects split definitions into separate modules, the ability to specialise a function in a module other than the one it is defined is crucial. By default, only functions marked `INLINABLE` are specialised across modules. The `INLINABLE` pragma makes the unfolding of a definition available in the interface file for a module. Therefore a new specialised definition can be created when the function is used at a known type in another module.

Availability of Unfoldings In order for specialisation of imported functions to be possible, the optimiser must have access to the definition of the function. This is because a new top-level definition is created using the body of the specialised function. Therefore we must consider under what situations an overloaded function will have an unfolding available if it should be specialised.

There are two common situations where GHC decides not to include the unfolding of an exported identifier in the interface file:

`-fexpose-all-unfoldings` Always expose an unfolding for a definition.

`-fspecialise-aggressively` Specialise functions which have an unfolding, even if they are not marked `INLINABLE`.

Figure 5.4: Flags related to specialisation

1. The size of the function is bigger than `-funfold-creation-threshold`.
2. The function is recursive, therefore, a loop-breaker, and will therefore never be inlined.

The implications of this is that without intervention, loop-breakers and big functions will also not be specialised. There are two flags which are useful to increase the availability of unfoldings (Figure 5.4). `-fexpose-all-unfoldings` makes sure the definition of every identifier is given an unfolding and `-fspecialise-aggressively` means any overloaded function with a static dictionary will be specialised. Due to the critical nature of specialisation to performance, some projects enable both options by default. This is at the expense of bigger interface files and longer compilation times.

Ordering of the Specialisation Pass The optimiser runs its passes in a specific order. There are situations where a specialisation opportunity can only become evident after other passes have been run but also after the specialisation pass has been run. In this case the function will not be specialised despite appearing in the program in the correct form.

At a high-level the issue arises when the first stage of specialisation itself produces a function with an overloaded type. This can happen if the type class method itself returns a universally quantified function with a type class constraint or a newtype wrapping such a function. The GENERIC-LENS library has an example of such functions.

```
class HasTypes a s where
  types :: ∀f.Applicative f ⇒ (a → f b) → s → f t
```

If HasTypes is specialised by the first specialisation pass then it may still leave the residual application of types to a known type f if types is used monomorphically. At this point, a later specialisation pass would be worthwhile in order to eliminate this application to a statically known dictionary.

Describing the precise situations where this possibility arises is not very easy due to the complexities of how the inliner works but a larger benchmark using HasTypes and a

plugin to enable a late specialisation pass demonstrated that in some situations the pass is worthwhile. The moral for the reader is that specialisation does not always fire when expected, even if the program looks to be in the correct form.

There is now a flag is GHC `-flate-specialise` which can be used to enable a specialisation pass towards the end of the core optimisation pipeline. Enabling that flag allows some programs using HasType to be optimised.

Argument Order In older versions of GHC, the specialiser would only specialise functions of a very specific form. The type of the function had to be a set of universally quantified variables, followed by dictionary arguments and then the normal argument types

$$\mathsf{normal} :: \forall \mathsf{a_i}.\sigma \Rightarrow \tau$$

GHC also permits more complicated types where quantification is interleaved with constraint arguments:

$$\mathsf{go} :: \mathsf{Int\#} \rightarrow \forall \mathsf{m} :: * \rightarrow *.\mathsf{Monad\ m} \Rightarrow ...$$

In go, the type variables and constraint arguments do not appear in the prefix of function type so the specialiser would fail to specialise go. It is a little tricky to end up in this situation as if go is written directly in a user program then the function type will be rearranged so that the type variable and constraint arguments are passed before the Int#.

Functions of this type can appear in core programs though after other optimisation phases. For instance, consider if the higher-order argument is instead first wrapped in a newtype, then the function will not be rewritten into the prenex normal form.

$$\textbf{newtype}\ \mathsf{Func} = \mathsf{Func}\ (\forall \mathsf{m} :: * \rightarrow *.\mathsf{Monad\ m} \Rightarrow ...)$$
$$\mathsf{go'} :: \mathsf{Int\#} \rightarrow \mathsf{Func}$$

In the core program the **newtype** wrapper will be replace by a coercion and hence have a similar form to go once elaborated.

Sandy Maguire and Alexis King have recently worked[1] to improve the specialiser in cases like this by relaxing the form a function needs to be in to be specialised. The restriction about functions having to be in prenex normal form has been lifted so that functions like go can be specialised.

[1] see `https://gitlab.haskell.org/ghc/ghc/-/issues/16473`

Implicit Parameters Another artificial restriction noticed by Alexis King was that if a function used the `ImplicitParameters` extension then it would not be specialised under any circumstances. This limitation applies to all the arguments of any function with at least one implicit parameter: even the statically-known non-implicit arguments of such a function are not specialised.

```
foo :: (?qux :: Bool, Show a) ⇒ a → String
```

If foo is used at a specific type a you would expect it to be specialised for a specific Show a dictionary. However the implicit parameter qux will mean that foo is specialised under no circumstance.

Partially Constant Dictionaries A related problem to specialisation is due to the interaction between multi-parameter type classes and super classes. In particular, in older versions of GHC, a superclass constraint would always be provided by projection from the evidence of the subclass.

```
module G where
class Num a ⇒ C a b where m :: a → b
f :: C Int b ⇒ b → Int → Int
f _ x = x + 1
```

The C Int b constraint is only partially known in the definition of f. The function is still polymorphic in the b type variable. Nevertheless, the knowledge about a means that the superclass Num a is now known to be Num Int.

The short-cut solver was implemented by Dan Haraj[2] in order to solve this problem. At the use site of +, which requires a Num dictionary, the constraint could be satisfied in two ways. It could either be satisfied by the super class dictionary of C Int b or directly by the top-level instance for Num Int. It is preferable to use the top-level instance where possible as it can reveal more specialisation opportunities. It is sound to use the top-level instance due to global type class coherence, in the presence of overlapping instances the shortcut is disabled.

This situation is in some ways similar between the difference of foo and foo1.

```
foo :: Int → Int → Int
foo = (+)
```

[2] see `https://gitlab.haskell.org/ghc/ghc/-/issues/12791`

```
foo1 :: Num Int ⇒ Int → Int → Int
foo1 = (+)
```

foo1 will be passed the Num Int dictionary at runtime, whilst foo will not. The issue in particular with multi-parameter type classes is that it was impossible to indicate to GHC that you wished the top-level instance to be used (as is the case in foo because the constraint is elided.). You are obliged to write the constraint for C if you use any of the method for C, even if enough type parameters are known to decide upon superclass constraints.

In the wild, this is a realistic problem. The problem originally arose from inspecting how the REFLEX library was optimised. In the library there is a DomBuilder m t class which has a simpler super class constraint:

```
class Reflex t ⇒ DomBuilder m t where ...
```

A more familiar example may be the MonoidWriter class, where knowing either parameter satisfies one of the super class constraints.

```
class (Monoid w, Monad m) ⇒ MonadWriter w m | m → w where
```

So despite the widespread usage of multi-parameter type classes, for a number of years they led to a performance trap as they would stop specialisation from happening.

Summary These previous examples have mostly been fixed in recent versions of GHC and so the specialiser is more predictable than it was. However, we have included them here to demonstrate that it has long been an unreliable way of optimising your program and should not be depended on. There is a lack of formal guarantee that the specialiser will certainly eliminate this very specific form of static argument. Our goal will be to investigate reliable methods of eliminating the kind of overhead created by the dictionary passing translation for situations where we want to be sure that the overhead is eliminated.

5.3.2 The Specialisation Trick

In Section 5.2 we stated that GHC will never inline a recursive function. There is one folklore technique which I learnt from Andres Löh where instead you can use specialisation to force GHC to unroll a finite amount of recursion.

Consider the problem of applying a function f k times to an argument. If k is known statically then the loop can be unrolled to k applications of f.

$$\mathsf{nTimes}\ 0\ (+1)\ 0 \equiv 0$$
$$\mathsf{nTimes}\ 4\ (+1)\ 0 \equiv (+1)\ ((+1)\ ((+1)\ ((+1)\ 0)))$$

Writing the function naively is straightforward but will not be specialised for a statically known k because it will not be inlined, because it is self-recursive.

```
nTimes :: Int → (a → a) → a → a
nTimes 0 f x = x
nTimes k f x = f (nTimes (k − 1) f x)
```

How can specialisation be used in order to define a version of nTimes which is unrolled? For specialisation to fire the argument to nTimes must be a type class dictionary which means defining nTimes using a typeclass (Unroll) which is parametrised by a type representing natural numbers. An instance is defined for the base case and inductive case which corresponds to the two cases of nTimes given above.

```
data Proxy (t :: k) = Proxy

data N = Z | S N

class Unroll (n :: N) where
  nTimes :: Proxy n → (a → a) → a → a

instance Unroll Z where
  nTimes _ f x = x

instance Unroll n ⇒ Unroll (S n) where
  nTimes p f x =
    let Proxy :: Proxy (S n) = p
    in f (nTimes (Proxy :: Proxy n) f x)
```

Now the argument to nTimes is essentially a type of kind N passed using the Proxy constructor.

```
oneTime :: (a → a) → a → a
oneTime = nTimes (Proxy :: Proxy (S Z))

fiveTimes :: (a → a) → a → a
fiveTimes = nTimes (Proxy :: Proxy (S (S (S (S (S Z))))))
```

Now specialisation can work on nTimes because the argument is a dictionary. In oneTime, first a specialisation of oneTime to the S Z dictionary is created before

a specialisation to the Z dictionary. The eventual result is an unrolled pipeline of functions. GHC is happy to do this repeated specialisation because the constraint language ensures that the type class evidence is finite and we can not enter situations where specialisation would happen forever.

nTimes is a long way from the original definition, the definition used a special type class, the `DataKinds` extension in order to create a value level natural number type and relied on specialisation firing in order to eliminate the first argument. For this simple example, it was already quite annoying to write the number 5, by repeated application of the S constructor but this is the tip of the iceberg. The type level language is very restricted in comparison to the value level language.

It is clearly deeply unsatisfying to have to program using the arid type level language in order to convince the optimiser to unroll a recursive function. Whenever possible the term-level language should be used for programming. Another solution proposed to the StackOverflow[3] question suggests instead to use *Untyped* Template Haskell to solve the problem.

```
nTimesTH :: Int → Q Exp
nTimesTH n = do
  f ← newName "f"
  x ← newName "x"
  when (n ⩽ 0) (reportWarning "nTimesTH: argument non-positive")
  let go k | k ⩽ 0 = VarE x
      go k = AppE (VarE f) (go (k − 1))
  return $ LamE [VarP f, VarP x] (go n)
```

The solution also readily yields a solution using Typed Template Haskell whose intent is much clearer than nTimeTH.

```
nTimesTTH :: Int → Code ((a → a) → a → a)
nTimesTTH 0 = ⟦ λf x → x ⟧
nTimesTTH n = ⟦ λf x → f ($(nTimesTTH (n − 1)) f x) ⟧
```

The existence of this folklore technique has inspired some authors to attempt to use it in order to write highly-optimising libraries. For example, it works quite well to implement functions over naturals and vectors. In private communication, Andres Löh indicated that he had attempted to reimplement GENERICS-SOP using similar techniques

[3] `https://stackoverflow.com/questions/42179783/is-there-any-way-to-inline-a-recursive-function`

but eventually ran into situations where the optimiser would not optimise his program any further.

I have also attempted to use this technique in order to transparently optimise generic traversals. The finer reasons about why this worked and why it was difficult are in the next section.

5.4 Deriving Efficient Lenses

The material of this section originally appeared in:

> Kiss, C., Pickering, M., and Wu, N. (2018). Generic deriving of generic traversals. *Proc. ACM Program. Lang.*, 2(ICFP)

In this section we will discuss the techniques used to force the optimiser to optimise the GENERIC-LENS library. Attempting to convince the optimiser to optimise the library took about a month of full-time work which consisted of looking at the core program after optimising and then modifying the library so that the optimisations worked for our test cases. Overall it was a very unsatisfying experience with no guarantee that in a future version of the compiler that the optimiser would behave in the same way.

At this point, one is left with two choices, either attempt to enforce some discipline on the optimiser or attempt to find an optimisation mechanism with a well-defined semantics. It was deemed hopeless to rely on the optimiser for any significant optimisation problem given the severe difficulties and corner cases this section has highlighted. Therefore, we began the search for a programming paradigm with guarantees about evaluation and the research into multi-stage programming begun in earnest.

5.4.1 Interface

GENERIC-LENS is a package which generically derives generic traversals. It works in the `GHC.Generics` generic programming style and uses the Generic class built into GHC in order to generate a generic traversal for any type which is an instance of Generic.

The library provides a number of traversal schemes, for example the generic function field generates a lens which accesses a specific field; positition, a specific position; types, generates a traversal which focuses on a specific type. The functions are implemented

using type classes so that when instantiated to a specific type, specialisation can remove the generic overhead. The type of types is as follows:

```
types :: ∀a s.HasTypes s a ⇒ Traversal s s a a
```

So once s and a are known, the HasType class can be specialised and generate a specialised traversal function which hopefully doesn't mention anything of the generic overhead.

5.4.2 Performance

When working generically we must always ask whether the abstraction comes at the cost of performance. In this case, it is pleasing that our use of generics is optimised away by the compiler. There are four crucial reasons why we can be confident that GHC will produce efficient code.

Evidence generation. By using a type-directed approach, the call hierarchy is known at compile time and this information can be used to unroll the definitions. This unrolling is achieved during *evidence generation*.

Specialisation. Functions using our methods will have constrained types but this overhead is eliminated via *specialisation*.

Inlining. We define our operations such that the composition operator is not recursive and can hence be readily *inlined*.

Internal representation. Finally, we choose an internal representation of our optics such that they expose the optimisation opportunities to the compiler.

In this section we describe the optimisations which we rely on to produce efficient code. We explain each of these techniques in turn. Our running example in this section is the incList function which maps over a list of trees and increments the Ints inside the tree.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
incList :: [Tree Int] → [Tree Int]
incList [] = []
incList (x : xs) = over (types@Int) (+1) x : incList xs
```

5.4.3 Evidence Generation

During compilation, type class constraints are desugared into arguments to the function (Wadler and Blott, 1989). The argument is known as a dictionary and contains a field for each method of a type class. Type class methods are then desugared as lookup functions into this dictionary.

The definition of incList uses types so the constraint solver must generate an evidence of HasTypes (Tree Int) Int, it does so by creating an appropriate dictionary.

The instance for HasTypes s a has constraints Generic s, and GHasTypes (Rep s) a. We focus on HasTypes and GHasTypes, treating the dictionary for Generic (Tree Int) implicitly. Thus the produced dictionaries are HasTypesDict and GHasTypesDict, corresponding to the appropriate classes.

```
data HasTypesDict s a = HasTypesDict {types :: Traversal s s a a}
data GHasTypesDict s a = GHasTypesDict {gtypes :: Traversal s s a a}
```

The necessary evidence generated for GHasTypes (Rep (Tree Int)) Int comes by providing the dictionaries for this type. The simplified representation for Tree Int without metadata nodes is:

```
Rep (Tree Int) ≡ K Int :+: (K (Tree Int) :×: K (Tree Int))
```

By working through this structure methodically, we arrive at the following dictionary definitions:

```
hasTypesDict_TreeInt :: HasTypesDict (Tree Int) Int
hasTypesDict_TreeInt = HasTypesDict {types = iso_Rep ∘ gtypes ghasTypesDict_TreeInt}

ghasTypesDict_TreeInt :: GHasTypesDict (Rep (Tree Int)) Int
ghasTypesDict_TreeInt = GHasTypesDict {gtypes = λf l1r1 → case l1r1 of
  L l → L ‹$› gtypes ghasTypesDict_KInt f l
  R r → R ‹$› gtypes ghasTypesDict_:×: f r}

ghasTypesDict_KInt :: GHasTypesDict (K Int) Int
ghasTypesDict_KInt = GHasTypesDict {gtypes = iso_K}

ghasTypesDict_KTreeInt :: GHasTypesDict (K (Tree Int)) Int
ghasTypesDict_KTreeInt = GHasTypesDict {gtypes = iso_K ∘ types hasTypesDict_TreeInt}

ghasTypesDict_:×: :: GHasTypesDict (K (Tree Int) :×: K (Tree Int)) Int
ghasTypesDict_:×: = GHasTypesDict {gtypes = λf (l :×: r) → (:×:) ‹$›
  gtypes ghasTypesDict_KTreeInt f l ‹*› gtypes ghasTypesDict_KTreeInt f r}
```

Evidence is first generated by using the instance for $:+:$, before recursing into both branches and finding evidence for $:\times:$ and the $\mathsf{K}\ \mathsf{Int}$ nodes. As such, we have a dictionary for each type constructor. The constraint solver will terminate as it will observe that the $\mathsf{ghasTypesDict}_{\mathsf{TreeInt}}$ dictionary can be used when trying to solve the recursive case. Thus, these dictionaries form a mutually recursive group. The dictionaries generated are straightforward transcriptions of the instances, with instance constraints solved and β-reduced. The definition of $\mathsf{ghasTypesDict}_{\mathsf{TreeInt}}$ is still not as efficient as it could be, and we discuss how it can be further improved with inlining in Section 5.4.3.

We see that the process of generating evidence also unrolls definitions. If we had instead defined types as a function over a normal data type without any type direction, it would be self-recursive and hence not able to be eliminated in the same manner. This process is safe as types are finite and statically known at compile time. Without additional language pragmas, the restrictions on instance contexts guarantee that the constraint solving process terminates.

Optimising Dictionaries

We recall that our generated dictionaries are mutually recursive. This isn't surprising, as we expect gtypes to be recursive in general if we are trying to traverse a recursive data structure. Mutually recursive blocks of functions must be treated with care, as repeatedly inlining them causes the inliner to diverge. Each mutually recursive group is thus appointed a loop-breaker function, which is never inlined, but other definitions can be freely inlined into each other in order to create a single self-recursive definition. After the dictionaries are inlined into each other, we end up with the following evidence which has the correct unrolled shape we were looking for.

$$
\begin{aligned}
&\mathsf{ghasTypesDict'}_{\mathsf{TreeInt}} :: \mathsf{GHasTypesDict}\ (\mathsf{Rep}\ (\mathsf{Tree}\ \mathsf{Int}))\ \mathsf{Int}\\
&\mathsf{ghasTypesDict'}_{\mathsf{TreeInt}} = \mathsf{GHasTypesDict}\ \{\mathsf{gtypes} = \lambda \mathsf{f}\ \mathsf{l1r1} \rightarrow \mathbf{case}\ \mathsf{l1r1}\ \mathbf{of}\\
&\quad \mathsf{L}\ \mathsf{l}\ \ \rightarrow \mathsf{L}\ \langle\$\rangle\ \mathsf{iso}_{\mathsf{K}}\ \mathsf{f}\ \mathsf{l}\\
&\quad \mathsf{R}\ \mathsf{b} \rightarrow \mathsf{R}\ \langle\$\rangle\ (\lambda \mathsf{f}\ (\mathsf{l} :\times: \mathsf{r}) \rightarrow (:\times:)\\
&\qquad\qquad \langle\$\rangle\ (\mathsf{iso}_{\mathsf{K}} \circ \mathsf{iso}_{\mathsf{Rep}} \circ \mathsf{gtypes}\ \mathsf{ghasTypesDict'}_{\mathsf{TreeInt}})\ \mathsf{f}\ \mathsf{l}\\
&\qquad\qquad \langle*\rangle\ (\mathsf{iso}_{\mathsf{K}} \circ \mathsf{iso}_{\mathsf{Rep}} \circ \mathsf{gtypes}\ \mathsf{ghasTypesDict'}_{\mathsf{TreeInt}})\ \mathsf{f}\ \mathsf{r})\ \mathsf{f}\ \mathsf{b}\}
\end{aligned}
$$

In this case, $\mathsf{ghasTypesDict'}_{\mathsf{TreeInt}}$ acts as the loop-breaker.

5.4.4 Specialisation

As we have seen, the evidence generation procedure and inlining are sufficient on their own to eliminate much of the generic overhead of a statically known parameter as long as we call the class method directly. However, we use class methods inside bigger functions and when we do they give rise to class constraints. When these larger functions are called, the dictionary must be solved and the required evidenced passed to the function.

For instance, we might want to write the more general type signature for incList to be parametric over the choice of data structure contained in the list as long as it contains integers. We will call this generalised version incListGen. If we call incListGen and instantiate s to be Tree Int then we should expect that the definition would be identical to incList.

```
incListGen :: HasTypes s Int ⇒ [s] → [s]
incListGen [] = []
incListGen (x : xs) = over (types@Int) (+1) x : incListGen xs
```

This problem is not trivial. When incListGen is called, the evidence witnessing the constraint HasTypes will be passed to it. In order to eliminate this dictionary, it needs to be pushed inwards to the call of types. Since incListGen is recursive, it cannot be inlined. Instead, we rely on *specialisation*.

The specialiser looks for calls to overloaded functions called at a known type. It then creates a new type-specialised definition which does not take a dictionary argument and a rewrite rule which rewrites the old version to the new version.

Suppose that we know that the value of s is Tree Int, and that the evidence dictionary for HasTypes is called treeIntHasTypes. The naive desugaring of calling incListGen@(Tree Int) xs is:

```
incListGen treeIntHasTypes xs
```

The specialiser then observes this call to incListGen takes a dictionary argument and creates a specialised version $\text{incListGen}_{\text{TreeInt}}$ with the following definition:

```
incListGen_TreeInt :: [Tree Int] → [Tree Int]
incListGen_TreeInt xs = (λhasTypesDict xs → case xs of
  [] → []
  (x : xs) → over (types hasTypesDict) (+1) x : incListGen hasTypesDict xs)
    treeIntHasTypes xs
```

The right-hand side of the definition is the same as the right-hand side of incListGen applied to treeIntHasTypes. Then, an additional rewrite rule is generated which replaces the overloaded call with the specialised definition.

```
{-# RULES "specincListGen" forall xs . incListGen treeIntHasTypes xs
                                    = incListGen_TreeInt #-}
```

That's the whole process. After β-reduction, the dictionary selector types is now adjacent to its dictionary and hence `types` can be inlined, and the correct method from treeIntHasTypes can be selected. Notice that in the definition of $\mathsf{incListGen}_{\mathsf{TreeInt}}$ there is still an overloaded call to incListGen, this will be rewritten when the rewrite rule is applied and then $\mathsf{incListGen}_{\mathsf{TreeInt}}$ will become self-recursive. After these two steps, all occurrences of treeIntHasTypes and thus the overloading overhead is eliminated.

Once again, specialisation is an *enabling* transformation. Later optimisation passes will perform more complicated rearranging with the express goal of improving our code.

5.4.5 Internal Representation

After this unrolled pipeline of functions is created, the question remains how this can become the same as hand-written definitions later in the compilation process. How precisely do inlining and β-reduction lead to good code? How and why depends on the internal representation of lenses and traversals we choose in the library.

Lenses

In the case of lenses, the inliner does a sufficient job of combining the composition of lenses into a single lens without further intervention. The lens composition operator is not recursive and hence is readily inlined which leads to much further simplification.

$$\mathbf{data}\ \mathsf{Lens}_1\ \mathsf{s}\ \mathsf{t}\ \mathsf{a}\ \mathsf{b} = \mathsf{Lens}_1\ (\mathsf{s} \to \mathsf{a})\ (\mathsf{b} \to \mathsf{s} \to \mathsf{t})$$

$$(\circ) :: \mathsf{Lens}_1\ \mathsf{s}\ \mathsf{t}\ \mathsf{c}\ \mathsf{d} \to \mathsf{Lens}_1\ \mathsf{c}\ \mathsf{d}\ \mathsf{a}\ \mathsf{b} \to \mathsf{Lens}_1\ \mathsf{s}\ \mathsf{t}\ \mathsf{a}\ \mathsf{b}$$

$$(\mathsf{Lens}_1\ \mathsf{get}_1\ \mathsf{set}_1) \circ (\mathsf{Lens}_1\ \mathsf{get}_2\ \mathsf{set}_2) = \mathsf{Lens}_1\ (\mathsf{get}_2 \circ \mathsf{get}_1)\ (\lambda \mathsf{b}\ \mathsf{s} \to \mathsf{set}_1\ (\mathsf{set}_2\ \mathsf{b}\ (\mathsf{get}_1\ \mathsf{s}))\ \mathsf{s})$$

In fact, the naive encoding given above for lenses does not produce the best results. Whilst it does collapse a sequence of compositions appropriately, the type ensures that in order to implement a modification operation, we must perform a get followed by a set and hence deconstruct s twice. We can get around this problem by using the existential

encoding which means that we can directly implement an updating function by only deconstructing the source once.

$$\textbf{data}\ \mathsf{Lens_2\ s\ t\ a\ b} = \forall \mathsf{c}.\mathsf{Lens_2}\ (\mathsf{s} \to (\mathsf{a}, \mathsf{c}))\ ((\mathsf{b}, \mathsf{c}) \to \mathsf{t})$$

Intuitively, the get function separates s into the part we are focusing on of type a and its complement c. In turn, the set function recombines a value of type b with the complement.

$$\begin{aligned}
&(\bullet) :: \mathsf{Lens_2\ s\ t\ c\ d} \to \mathsf{Lens_2\ c\ d\ a\ b} \to \mathsf{Lens_2\ s\ t\ a\ b}\\
&(\mathsf{Lens_2\ get_1\ set_1}) \bullet (\mathsf{Lens_2\ get_2\ set_2}) = \mathsf{Lens_2\ get\ set}\ \textbf{where}\\
&\quad \mathsf{get\ s} = \textbf{let}\ (\mathsf{c}, \mathsf{com_1}) = \mathsf{get_1\ s}; (\mathsf{a}, \mathsf{com_2}) = \mathsf{get_2\ c}\ \textbf{in}\ (\mathsf{a}, (\mathsf{com_1}, \mathsf{com_2}))\\
&\quad \mathsf{set}\ (\mathsf{b}, (\mathsf{com_1}, \mathsf{com_2})) = \mathsf{set_1}\ ((\mathsf{set_2}\ (\mathsf{b}, \mathsf{com_2})), \mathsf{com_1})\\
&\mathsf{modify} :: \mathsf{Lens_2\ s\ t\ a\ b} \to (\mathsf{a} \to \mathsf{b}) \to (\mathsf{s} \to \mathsf{t})\\
&\mathsf{modify}\ (\mathsf{Lens_2\ get\ set})\ \mathsf{f\ s} = \textbf{let}\ (\mathsf{a}, \mathsf{c}) = \mathsf{get\ s}\ \textbf{in}\ \mathsf{set}\ ((\mathsf{f\ a}), \mathsf{c})
\end{aligned}$$

Using this definition, chained modifications can be fused into a single function. Thus, in our implementation, the lenses we use are of this latter encoding. Once we have fused them together, we turn them into whichever encoding that we want the library to produce. By default, it is the van Laarhoven encoding as found in the LENS library (Kmett, 2018).

Traversals

Optimising traversals in the same manner is slightly trickier as we must find an encoding of a traversal which does not require a recursive composition operator. In order to do this, we use a van Laarhoven style representation. The composition operator for these traversals is the function composition operator. However, this is not sufficient, the downside of using this composition operator is that it does not perform normalisation as happened with lenses. It is necessary to appeal to the Applicative laws in order to rearrange and normalise these compositions. The following technique is due to Eric Mertens and can be found implemented in the LENS library.

A van Laarhoven Traversal is a function with the following type.

$$\textbf{type}\ \mathsf{Traversal\ s\ t\ a\ b} = \forall \mathsf{g}.\mathsf{Applicative\ g} \Rightarrow (\mathsf{a} \to \mathsf{g\ b}) \to \mathsf{s} \to \mathsf{g\ t}$$

The result type of these functions is a value constructed using Applicative operators. Applicative expressions have a normal form of a single pure followed by a sequence of left-associated applications using the combinator $\langle * \rangle$ (McBride and Paterson, 2008). In

order to rewrite this normal form, we must re-associate all uses of $\langle * \rangle$ and then fuse together all uses of pure.

This first step is achieved by instantiating g to be Curried.

```
data Curried f a = Curried {runCurried :: ∀r.f (a → r) → f r}
instance Functor f ⇒ Functor (Curried f) where
  fmap f (Curried v) = Curried (λfar → v (fmap (∘f) far))
instance Functor f ⇒ Applicative (Curried f) where
  pure a = Curried (λfar → fmap ($a) far)
  Curried mf ‹*› Curried ma = Curried (ma ∘ mf ∘ fmap (∘))
```

It is the definition of $\langle * \rangle$ which performs the reassociation. Notice that the Applicative instance for Curried delegates all calls to pure to the underlying functor. We will fuse those together with an additional layer termed Yoneda which intercepts all the calls to fmap and fuses them together.

```
data Yoneda f a = Yoneda {runYoneda :: ∀r.(a → r) → f r}
instance Functor (Yoneda f) where
  fmap f (Yoneda v) = Yoneda (λk → v (k ∘ f))
instance Applicative f ⇒ Applicative (Yoneda f) where
  pure a = Yoneda (λf → pure (f a))
  Yoneda m ‹*› Yoneda n = Yoneda (λf → m (f∘) ‹*› n id)
```

This time, we notice that Yoneda just delegates the definitions of the Applicative. Putting this together, we instantiate g to be Curried (Yoneda g) and then use lowerCurriedYoneda in order to return to a simple type parameterised by an Applicative constraint.

```
liftCurriedYoneda :: Applicative g ⇒ g a → Curried (Yoneda g) a
lowerCurriedYoneda :: Applicative g ⇒ Curried (Yoneda g) a → g a
```

This process performs the reassociating and fusion that we desired. However, in practice, it is difficult to be sure that the compiler will remove this overhead. On the other hand, it does not affect performance in common use cases such as modifying or summarising. This is because when g is instantiated to a known Applicative, the non-recursive Applicative methods can be inlined. We usually instantiate to g to either Const or Identity which are completely eliminated.

Using similar techniques to traversals, van Laarhoven or profunctor representation of lenses and prisms could also be optimised, but these simple minded techniques are the

most reliable and very effective in generating good programs without impacting compile times significantly.

5.4.6 Compiler Flags

Given that the library has obeyed the principles of optimisation, it was still not enough to produce an optimal program. Two more modifications to the compiler's default flags are needed in order to make the small collection of benchmarks work.

- `-fspecialise-aggressively` and `-fexpose-all-unfoldings` were used to make sure cross-module inlining worked.
- `-flate-specialise` was implemented as a plugin pass to make sure that all specialisation opportunities were specialised. This made a particular difference in one benchmark.

For our specific benchmarks, this was enough, but it has been observed in the wild that sometimes it is also necessary to increase `-funfolding-use-threshold` to make sure that the relevant methods get inlined. It is a bit unfortunate that these global options have to be modified in order to make sure that using a specific library optimises correctly.

In the intervening years each GHC release has broken the test suite which asserts that definitions will optimise to the same as a hand-written version of the function. These tests are verified using the INSPECTION-TESTING library (Breitner, 2018). The test works by comparing the core of the generically derived lens to the core from an optimal hand-written variant. So any work done in order to convince the optimiser to produce a certain program has to be reassessed every time a new release is made as the behaviour usually changes in undocumented ways.

Overall despite spending a lot of time setting up our library properly, using type classes and non-recursive functions, we are still left with something which is quite brittle. It would be much better if the language itself would guarantee certain properties about the optimisation so the library author could program confidently without having to spend their time fighting something complex and ill-specified.

5.5 Case Study: Staged SOP

In this section we will reflect on the how the implementation has enabled new staged programs to be implemented. In particular the invention of CodeC constraints has been

very useful when using Typed Template Haskell in order to stage a generic programming library.

STAGED-SOP was joint work with Andres Löh. The original implementation was performed together in Regensberg, September 2019. It was then we realised how significant the issues with constraints and quotations discussed in Section 3.2 which we resolved in Bristol, January 2020. I then formalised the extension and implemented the extension in GHC whilst Andres modified the library in order to work with the branch. This section is based on:

> Pickering, M., Löh, A., and Wu, N. (2020). Staged sums of products. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, Haskell 2020, pages 122–135, New York, NY, USA. Association for Computing Machinery.

5.5.1 Generic Programming using Sums of Products

Generic programming is a natural fit for staging. The premise of generic programming is to write functions to perform a specific task such as traversal or summation based on the type of a value. From the type of the value a large amount of *static* information is known.

What has plagued authors of generic programming libraries is that generic functions are usually less efficient than their hand-written counterparts. This is disappointing as in theory it should be possible to eliminate the statically known information during compilation.

GENERICS-SOP basics

We will briefly recap the basics of the GENERICS-SOP library before discussing how to modify the interface to turn it into STAGED-SOP. Then examples of using the CodeC constraints will demonstrated in conjunction with examples of using STAGED-SOP.

The generic representation of a datatype used by GENERICS-SOP is its decomposition into sums of products.

```
data NS :: (a → Type) → [a] → Type where
  Z :: f x → NS f (x : xs)
  S :: NS f xs → NS f (x : xs)
```

```
data NP :: (a → Type) → [a] → Type where
  Nil :: NP f []
  (:*) ::: f x → NP f xs → NP f (x : xs)
newtype SOP f xss = SOP (NP (NP f) xss)
```

The library provides many functions for manipulating the SOP, NP and NS data types. Since it is possible to inject and project a datatype from the generic representation, generic functions are implemented by first converting the datatype into the SOP form, computing the generic information and then converting back to the desired datatype. In order to support these conversion operations for any algebraic datatype a class Generic is defined which includes methods to perform the conversions. This class is usually implemented using GHC.Generics or using Untyped Template Haskell.

```
class Generic a where
  type Description a :: [[Type]]
  from :: a → Rep a
  to :: Rep a → a
type Rep a = SOP I (Description a)
```

The methods from and to form an isomorphism.

For example, for the tree data type the Description of Tree is a list of length two which contains a description for each constructor of Tree. Leaf is represented by a list containing a and Branch by the list containing two copies of Tree a. Notice that the representation is *shallow* so the description includes references to the original Tree type.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
instance Generic (Tree a) where
  type Description (Tree a) = [[a], [Tree a, Tree a]]
  from :: Tree a → Rep (Tree a)
  from (Leaf a) = Z (I a :* Nil)
  from (Branch t1 t2) = S (I t1 :* I t2 :* Nil)
  to (Z (I a :* Nil)) = Leaf a
  to (S (I t1 :* I t2 :* Nil)) = Branch t1 t2
```

Generic functions are implemented by manipulations of the SOP, NS and NP types. The generic mapping function (mapNP) can be used to modify the functor that each element is wrapped in. It is defined by structural recursion over the NP type.

```
mapNP :: (∀x.f x → g x) → NP f xs → NP g xs
mapNP _ Nil = Nil
mapNP f (x :* xs) = f x :* mapNP f xs
```

The mapNP function is very much like the normal map function but with a more refined type signature. There are also generic variants of other common functions which work on lists such as folds.

```
cataNP :: r [ ] → (∀y ys.f y → r ys → r (y : ys)) → NP f xs → r xs
cataNS :: (∀y ys.f y → r (y : ys)) → (∀y ys.r ys → r (y : ys)) → NS f xs → r xs
```

Using cataNP and cataNS is it straightforward to construct a function which counts the number of fields a certain value has.

```
count :: Generic a ⇒ a → Int
count = getConst ∘ gcount ∘ from
  where
    gcount = cataNS (cataNP (K 0) (λv (K n) → K (n + 1))) (λ(K r) → K r)
```

count first converts the value into the generic representation which is then folded by the cataNS and cataNP functions in order to result in a value type K Int which is unwrapped by the getConst field accessor.

The definition is all well and good, but it isn't practical. Not for the reason that the function doesn't compute anything useful but because there is now a significant performance overhead of having to convert the value into the generic representation.

For the tree function, a user would instead write this very direct definition.

```
treeCount :: Tree a → Int
treeCount (Leaf { }) = 1
treeCount (Branch { }) = 2
```

Is there any hope that without further intervention that the optimiser will be able to convert the high-level generic definition gcount into the specialised low-level variant treeCount if the type of value is known to be Tree? Inspecting the definition of gcount you can see that there are at least two recursive traversals (cataNS and cataNP) so the likelihood of GHC's optimiser being able to make progress here is slim.

STAGED-SOP is a staged variant of GENERICS-SOP which will guarantee that the definition of count can be specialised to a specific type and result in the same program as the hand-written definition.

5.5.2 Staged Sums of Products

Now it is time to turn to staging GENERICS-SOP.

Binding-time analysis The first aspect of staging a library is performing a binding-time analysis. In a generic program, the structure of the type is *static* information. The values contained in the leaves of a type are *dynamic* information. For example, given a value of type Tree a you know the value is *either* a Leaf or Branch constructor and that if it's a Leaf, there is a single value of type a inside there or if it's a branch there are two values of type Tree a. However, you do not know statically what the value of type a or Tree a are, this is only information available at runtime.

Therefore when staging the library we want to be able to manipulate a datatype which encodes the distinction between the static and dynamic structure. The SOP type can already encode this structure because of the type parameter f which dictates how each value is wrapped inside the heterogeneous list. The first step in staging the library is to modify the representation type from wrapping the dynamic components in the Identity type constructor to the Code type constructor to indicate that they are dynamic. The CRep type synonym is the staged analogue of the Rep type synonym and will be the isomorphic generic representation that will be used to implement generic operations on values.

```
type CRep a = SOP Code (Description a)
```

The same combinators can be used as before in order to manipulate the generic representation.

Converting to CRep The second part of the staging process is to work out how to implement functions which convert from a value into the generic representation. There will have to be one slight modification to the conversion functions, but firstly a false start by naively modifying the from and to functions to convert to and from a CRep rather than Rep as before.

```
class CGeneric a where
  type Description a :: [[Type]]
  cfrom :: Code a → CRep a
  cto :: CRep a → Code a
```

The to function, which forgets static information, can be implemented without any issue. On the other hand, the cfrom function can't be implemented in this form. Now that the

argument to cfrom is dynamic information, how do we know which constructor the value will be at compile-time? In general, we will not and therefore can't select the right injection from NS in order to build the CRep value.

However, if we could instead say how we *would* deal with each case then a case expression can be generated in order to handle each case depending on which constructor is actually used at runtime. Therefore the type of cfrom is modified to take a continuation which can be invoked in each branch, where the information about the identity of the constructor is statically known.

```
class CGeneric a where
  type Description a :: [[Type]]
  cfrom :: Code a → (SOP Code (Description a) → Code r) → Code r
  cto :: CRep a → Code a
```

Using continuations is a common trick to improve the binding time properties of a program (Jones et al., 1993).

Now the CGeneric instance for Tree a can be implemented. As with the Generic class this instance can be automatically derived for datatypes that can be represented in the sums of products encoding.

```
instance CGeneric (Tree a) where
  type Description (Tree a) = [[a], [Tree a, Tree a]]
  cfrom :: Code (Tree a) → ((SOP Code (Description (Tree a)) → Code r) → Code r
  cfrom v k = [|  case $(v) of
    (Leaf a) → $(k (Z ([| a |] :* Nil)))
    (Branch t1 t2) = $(k (S ([| t1 |] :* [| t2 |] :* Nil))) |]
  cto (Z (a :* Nil)) = [| Leaf $(a) |]
  cto (S (t1 :* t2 :* Nil)) = [| Branch $(t1) $(t2) |]
```

Using the New Features Modifying GENERIC-SOP seemed quite straightforward but the work isn't done yet. Writing simple generic functions which don't require any class constraints can be achieved in older GHC versions. If you want to write functions which generate code using class constraints, which is the majority of generic programs, then an implementation which treats class constraints soundly is critical.

For example, most generic traversal functions also have a variant where the higher-order traversal function is also parametrised by a constraint. This enables more complicated

traversals of datatypes where all fields satisfy a particular constraint. A function which implements the monoidal append can be implemented if each type contained within the data type is an instance of the Monoid class. The generic function operates by monoidally combining each field.

The All c xs type family indicates that all types in xs satisfy the c constraint. The czipWithNP function is a generalisation of the mapNP function from before. It can be used to zip together two heterogeneous lists, with the assumption that each field satisfies a certain constraint c.

```
czipWithNP :: All c xs ⇒ Proxy c
           → (∀x.c x ⇒ f x → g x → h x)
           → NP f xs → NP g xs → NP h xs
```

The unstaged append function can therefore be implemented as essentially the composition of to, from and czipWithNP.

```
type IsProduct a xs = (Description a∼[xs], Generic a)
productTypeFrom :: (IsProduct a xs) ⇒ a → NP I xs
productTypeFrom = case from a of SOP (Z xs) → xs

productTypeTo :: (IsProduct a xs) ⇒ NP I xs → a
productTypeTo = to (SOP (Z xs))

gappend :: (IsProduct a xs, All Monoid xs) ⇒ a → a → a
gappend xs ys = productTypeTo
                  (czipWithNP (Proxy Monoid) go
                    (productTypeFrom xs)
                    (productTypeFrom ys))
  where
    go (I x) (I y) = I (x `mappend` y)
```

Now, to turn to the staged variant, gappend will be refactored to turn into a code generator which will generate the code to perform the appending of two specific types. Which parts will be have to change? Firstly the uses of from will have to be modified as cfrom is now using continuation-passing style. Then the mapping function in czipWithNP will be modified to combine together Code fragments rather than values wrapped in identity functors.

```
productTypeFrom' :: (IsProduct a xs) → Code a → (NP C xs → Code r) → Code r
productTypeFrom' c k = cfrom (λ(SOP (Z xs)) → k xs)
```

```
productTypeTo :: (IsProduct a xs) ⇒ NP C xs → Code a
productTypeTo = cto

sappend :: (IsProduct a xs, All (Compose CodeC Monoid) xs) ⇒ Code a → Code a → Code a
sappend cxs cys = productTypeFrom'
                    (λxs → productTypeFrom'
                      (λys → productTypeTo
                        (czipWithNP (Proxy (Compose CodeC Monoid)) go xs ys)))
  where
    go xn yn = [|| $(x) `mappend` $(yn) ||]
```

The changes to from and go were mechnanical translations. The application in go is now in terms of the Code functor rather than the Identity functor from before. The mappend operation now appears inside a quotation which means that the type of go is CodeC (Monoid a) ⇒ Code a → Code a → Code a. This explains the appearance of the All (Compose Code Monoid) xs constraint which indicates that every field must satisfy the CodeC (Monoid a) constraint.

Now, the generated code for sappend will look close to the hand-written version and the overhead of the generic machinery is eliminated at compile-time.

Rewriting the function into a staged version required some superficial changes to the unstaged version. The main work-horse of sappend remains the same czipWithNP function as in the gappend case. In a more complicated generic function, where the generic representation goes through several steps of manipulation, the heart will be bigger and the appearance of the from and to functions will be less prominent.

Reflections on CodeC The introduction of the CodeC constraint form has proved its worth already in STAGED-SOP. Since CodeC is a normal constraint, much like Code is a normal value, it can be manipulated just like other constraints. In the sappend example it was composed with the Monoid class using the Compose constraint and then distributed over a list using All. Without being able to manipulate future-stage constraints like future-stage values, it seems much more difficult to imagine how to implement a function like sappend.

The CodeC abstraction can also be mostly hidden away from a user by introducing a new type synonym Quoted c a for the combinator of CodeC (c a) and LiftT a.

More examples of writing staged generic functions can be found in the paper (Pickering et al., 2020).

Staged Generic Lenses In the background section we recounted the steps taken in order to ensure that GHC's optimiser was able to optimise the GENERIC-LENS library. It took a large amount of effort to convince the optimiser to do the right thing and with no firm guarantee it would continue to do so in the future. The promise of this thesis has been to enable generic programming libraries in Haskell such as GENERIC-LENS to ensure to their users that the generic overhead will be eliminated.

In this section we'll implement the position lens which for a given type and integer, will produce a lens which indexes into that type. The reason for implementing position rather than field lens is that position doesn't require any additional metainformation about the names of the fields.

For example, position can be used to generate lenses for the age or baldness field for the Person data type.

```
data Person = Person {age :: Int, isBald :: Bool, name :: String}

zero = SZ
one = SS SZ

getBald :: Person → Int
getBald = view $(position zero)

setBald :: Bool → Prod → Prod
setBald = set $(position one)
```

Then the derived code can be used to modify values of type Person:

```
>>> person = Person 30 False "Peggy"
>>> setBald True person
Person {age = 30, isBald = True, name = "Peggy"}
```

Rather than generate the van Laarhoven representation, the implementation will generate the simple representation which comprises of a pair of a getter and a setter. This avoids introducing complications involving data types which container higher-rank functions.

```
data Lens s a = Lens {lensGet :: s → a, lensSet :: a → s → s}

view :: Lens s a → s → a
view (Lens v _) = v

set :: Lens s a → a → s → s
set (Lens _ v) = v
```

The position index will be given by a singleton SNat n.

```
data Nat = NZ | SN Nat
data SNat n where
  SZ :: SNat NZ
  SS :: SNat n → SNat (SN n)
```

The lensGet and lensSet components are implemented using the replace and get functions which manipulate the generic product representation. replace traverses the product and replaces the suitable field. get projects the relevant field from the product.

```
type family Index n xs :: ∗where
  Index NZ [] = TypeError (Text "Empty")
  Index NZ (x : xs) = x
  Index (SN n) (_ : xs) = Index n xs
replace :: Index n xs∼x ⇒ SNat n → NP f xs → f x → NP f xs
replace SZ (x :∗ xs) v = v :∗ xs
replace (SS n) (x :∗ xs) v = x :∗ replace n xs v
get :: Index n xs∼x ⇒ SNat n → NP f xs → f x
get SZ (x :∗ xs) = x
get (SS n) (x :∗ xs) = get n xs
```

They are combined together in position, which uses sproductTypeFrom to convert a value to its generic representation before replace and get perform the necessary manipulation.

```
position :: ∀a xs n r.(SIsProductType a xs, r∼Index n xs, LiftT r)
         ⇒ SNat n → Code (Lens a (Index n xs))
position n =
  let get' :: Code a → Code (Index n xs)
      get' s = sproductTypeFrom s $ λxs → get n xs
      set' :: Code (Index n xs) → Code a → Code a
      set' a s = sproductTypeFrom s $ λxs → sproductTypeTo (replace n xs a)
  in ⟦ Lens (λs → $(get' ⟦ s ⟧)) (λa s → $(set' ⟦ a ⟧ ⟦ s ⟧)) ⟧
```

The result is that the program generated by using position will contain no trace of the generic representation. Therefore we can be much more confident that of the performance characteristics of any usage of view or set.

5.5.3 Conclusion

In this section we saw some situations where it was necessary to use the CodeC constraint form in order to write a program generator. The first-class nature of CodeC was useful when used in conjunction with the All type family. Staging is a natural fit for the sums of product style and the resulting library is an elegant way to write high-level efficient generic functions. There are more examples of writing staged generic functions using STAGED-SOP in the corresponding Haskell Symposium paper (Pickering et al., 2020).

Others have also applied staged programming to generic programming but in the SYB style (Yallop, 2017). This staging requires some more sophisticated techniques such as partially static datatypes due to how SYB usually relies on runtime type checks.

Chapter 6

Multi-Stage CBPV

In this chapter we change tack and formally tackle another issue with multi-stage programming languages: the inability to lift function values. The idea of the language will be to use a meta-representation to syntactically represent unevaluated expressions which can be interpreted into a specific stage during the execution of the program. This modification will allow a source language, which contains a lift operation, to be compiled to this core language with the ability to "lift" any value. The lifting operation will be implemented by interpreting syntax into the appropriate level, rather than having to convert a possibly opaque runtime value into its representation.

The language is an extended variant of Levy's CBPV language (Levy, 2004), which was chosen because it distinguishes between unevaluated expressions and evaluated values. Only small modifications are necessary in order to firstly modify it into a simple multi-stage language and then to add a new value type for representing unevaluated expressions.

The structure of this chapter is as follows:

- Firstly in Section 6.1 the informal motivation as to why lifting functions is desirable at all is explained with some practical examples from a staged parser combinator library.
- We then reflect in Section 6.2 on why extending Levy's CBPV to a multi-stage language would provide a good language for a solution to this problem.
- The languages and their elaborations are formally described in Section 6.3. Section 6.3.11 contains some example programs.
- We conclude with quite a long discussion about possible future formal directions

(Section 6.4) and some interesting related work (Section 6.5).

The work presented in this chapter is at a preiminary stage. Further work to implement the language and prove formal properties are necessary in order to be confident about the design.t

6.1 Motivation

The purpose of a multi-stage language is to separate a program into levels. Each level of the program can be completely evaluated before evaluation of the next level starts. The typing rules for a multi-stage language are designed to ensure this principle. The necessary modification is to usually add a level index to the typing rules which ensures that the staged evaluation is possible. The primary complication of the level indexing is binding structures. Each value in the environment is also marked with the level it is introduced so that only variables can only be used at the level they are bound. Recall the discussion of hasty and tardy variables from Section 2.1.2.

In particular, if a variable bound at level k is used at level $k+1$ then instead of rejecting a program, most type systems for a multi-stage language instead choose to interpret this reference as a request to copy the value the variable holds at level k into the generated program at level $k+1$. This mechanism is known as *cross-stage persistence by lifting* and follows the original suggestion by Taha and Sheard (2000).

For example, implementing a function which generates code for double a specific number could be written as follows:

$$\mathsf{double} :: \mathsf{Int} \to \mathsf{Code\ Int}$$
$$\mathsf{double\ x} = [\![\ \mathsf{x} * 2\ ]\!]$$

The variable x is bound at level 0 but used inside the quotation at level 1. In a typical language with cross-stage persistence the doubling function is interpreted into a combination of a splice and a lifting function. Therefore double is equivalent to double' below.

$$\mathsf{double'} :: \mathsf{Int} \to \mathsf{Code\ Int}$$
$$\mathsf{double'\ x} = [\![\ \$(\mathsf{lift\ x}) * 2\ ]\!]$$

Now when double' is applied to a concrete integer, for example 0, it will produce the code which doubles that number but does not evaluate it. The result of evaluating double 0 is $[\![\ 0 * 2\ ]\!]$.

Typically not all values are liftable, the Lift class is implemented for liftable types.

```
class Lift a where
  lift :: a → Code a
```

An implementation of the Lift class is expected to obey the following law:

$$\$(\mathsf{lift}\ \mathsf{x}) \equiv \mathsf{x}$$

In a typical implementation only simple base types are liftable such as integers, strings and normal algebraic data types. Types which are liftable under this scheme are ones which from the runtime representation you can construct the syntactic representation. For example, given an integer you can construct the syntax representing that integer literal instead. The specific exception to the liftable values that we care about are functions. At runtime in most programming languages functions are not inspectable and the only operation which be performed with a function is applying it to an argument. Therefore the internal representation is optimised for this use case and any specific information about the definition is lost by the runtime. There is no way to create a syntactic representation of a function given the runtime value. As a result, in Haskell, the Lift type class is not implemented for function types.

This is not a theoretical concern. Take one particular example that we have run into, in a staged parser combinator library (Willis et al., 2020) it is common to provide combinators which use functions as arguments. The act of staging a library is the act of turning a runtime interpreter of the combinators into a code generator. The code which gets generated needs to contain the functions that the user has supplied as an argument and in order to provide the same interface this amounts to being able to persist functions. A typical example of this is the mapping combinator ‹$› which applies a function to the result of a parser.

```
(‹$›) :: (a → b) → Parser a → Parser b
```

In the case of compilation, the denotation of a Parser is a value of type Code (String → Maybe a) so therefore the meaning of ‹$› should be to apply the function of type (a → b) to the value of type Code (String → Maybe a) in order to produce a value of type Code (String → Maybe b). The compile function computes this denotation from the Parser AST.

```
compile :: Parser a → Code (String → Maybe a)
```

Without the ability to lift function values, it should be clear that there is no way to implement this function since the only way it is possible to interact with the function f is by applying it to an argument.

The workaround for this issue in practical programs is modify the type of ‹$› so that as well as storing a present stage value, its future stage representation is also stored. This necessitates modifying the type of the ‹$› combinator to ensure that the user supplies both the present and future stage representations. The WQ wrapper is introduced to store a pair of a value of type a and also its future stage representation of type Code a.

```
data WQ a = WQ a (Code a)
(‹$›) :: WQ (a → b) → Parser a → Parser b
```

(‹$›) is then modified to take a value of type WQ (a → b) as the first argument so that when passed to compile, the future stage representation can be extracted from the pair to be used in the generated program. If we compare this to the combinator which parses a specific string,

```
string :: String → Parser a
```

the same interface can be used in both the staged and unstaged variants of the library because values of type String are liftable as it is a instance of the Lift type class. Therefore the conversion from String to Code String can be performed by calling lift. For functions, as it is not an instance of Lift, the WQ data type stores both the function f and also its representation so either can be projected as necessary.

In order to use the new interface, all calls to the mapping combinator must be modified to pass a WQ value rather than the function directly.

```
addOne :: Parser Int → Parser Int
addOne p = WQ (+1) ⟦ (+1) ⟧ ‹$› p
```

Unfortunately this approach now leaks information about the implementation of compile to the user of the library. Users of the library have to firstly understand why they must treat functions differently and then jump through hoops to use functions. For functional programming languages this second class nature of abstracting over functions is unfortunate. We have taken a specialised mapping function as an example from a parser combinator library but this issue affects any library which abstracts over function values. The restriction also means common interfaces such as Functor, Applicative, Profunctor and so on are not suitable for usage in combination with multi-stage features. When refactoring a library or application into a staged style this is one particular pain-point because any usage of these common abstractions has to be modified due to the interface changing.

It is even more unfortunate because any function we want to persist arises in one of three different ways.

- A user explicitly writes a lambda in their program.
- The function is imported from another module.
- The function is the result of partially applying a function (arising from the first two possibilities) to an argument.

In each of the three cases, it is possible automatically to create a WQ yet the user must manually create it themselves by explicitly writing a quotation. Therefore, if the compiler could perform a similar job for us then functions would also be liftable and the lifting could be implemented by projection from the WQ value. So, we have seen two possible strategies so far for implementing lifting. The first one which is how lifting is normally implemented by a runtime function which uses the host language pattern matching facilities to create a syntactic representation of the value. The second method is if the language can always keep track of the representation which would be used as evaluation proceeds then the primary computational aspect of lifting can be removed.

It is a variant of the second approach that we take in this thesis. All values are represented as uninterpreted expressions until the moment they are needed for computation. Therefore, as each expression remains uninterpreted into a specific stage until the last possible moment, any value can be persisted into any future stage as necessary. To make the idea concrete, consider again a modification to the interface which replaces the WQ argument with an unevaluated syntactic representation of a term (Syntax). Then Syntax can be interpreted into either Code or a present stage value.

```
data Syntax a where
  ...
  AddOne :: Syntax (Int → Int)
  ...
interpCode :: Syntax a → Code a
interpCode AddOne = ⟦ (+1) ⟧
interpNormal :: Syntax a → a
interpNormal AddOne = (+1)
(‹$›) :: Syntax a → Parser a → Parser a
addOne' :: Parser Int → Parser Int
addOne' p = AddOne ‹$› p
```

$$\tau \quad ::= \quad Int \mid () \mid \tau_1 \times \tau_2 \mid Void \mid \tau_1 + \tau_2 \mid U\chi$$

Figure 6.1: CBPV Value Types

$$\chi \quad ::= \quad F\tau \mid \tau \to \chi \mid \top \mid \chi_1 \& \chi_2$$

Figure 6.2: CBPV Computation Types

The design for our language will formalise this idea and make the usage of the representation transparent to the user.

The starting point for designing this language with the ability to lift function values is the observation that meta-programming where we are trying to delay interpretation as long as possible is very similar in flavour to a call-by-name evaluation strategy where expressions are only evaluated when they are needed. The natural starting point for studying a language where we want to be able to describe precise properties of the evaluation order is a CBPV style language which allows us to precisely express when expressions are evaluated. Therefore we will use a level indexed variant of CBPV in order to firstly answer the question we have presented here about persisting function values but also as a candidate for a more general core language for multi-stage programs in which many different ideas are possible to express. We will sketch some of these possibilities in Section 6.4.

6.2 CBPV Background

In this section we recall the definition and motivation for a CBPV language before explaining our extensions informally before moving onto a formalism in Section 6.3.

Call-By-Push-Value (CBPV) is a calculus which makes evaluation order explicit. The slogan of CBPV is: values are, computations do. This is reflected by the fact that computations are given reduction semantics but values are inert. The difference between a value and a computation is evidenced in the types, values of type τ are embedded in a computation of type $F\tau$ by using **return** and computations can be embedded into values using the thunking operation $\{e\}$. Effectful computations are sequenced by **let** and computation values forced by $!()$. $F\tau$ is the type of computations which produce a value of type τ. A value of type $U\chi$ is a thunk, which when forced will produce a

$$v \quad ::= \quad \mathrm{Z} \mid \mathrm{S}\ v \mid () \mid (v_1, v_2) \mid L\ v_1 \mid R\ v_1 \mid \{e\}$$

Figure 6.3: CBPV Values

$$
\begin{array}{rcl}
e & ::= & \textbf{split}(v, x_1.x_2.e) \mid \textbf{case}_0(v) \mid \langle\rangle \mid \textbf{case}(V, x_1.e_1, x_2.e_2) \mid !(v) \\
& & \mid \textbf{return}\ v \mid \textbf{let}\ x \leftarrow e\ \textbf{in}\ e' \mid \lambda x.e \mid e\ v \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e
\end{array}
$$

Figure 6.4: CBPV Computations

$$
\begin{array}{rcl}
\tau_L & ::= & Unit \mid \tau_L \to \tau_L \\
L & ::= & () \mid x \mid \lambda x.L \mid L_1 L_2
\end{array}
$$

Figure 6.5: STLC Syntax

computation of type χ.

Since the calculus is very explicit about evaluation order and the difference between an evaluated expression (a value) and an unevaluated expression (a computation) different evaluation strategies can be expressed within the calculus.

This distinction allows faithful translations from both call-by-name and call-by-value into call-by-push-value. The translations are faithful in the sense that the reduction order is preserved by the translation. For additional details readers are encouraged to consult Levy (2004) where many additional properties of CBPV are considered. Forster et al. (2019) provide a more accessible introduction with a more familiar syntax.

CBPV was introduced in order to understand effects in a more fine-grained manner. Evaluation order is especially important when dealing with effects as different evaluation strategies lead to the program having different results. By designing a calculus where different evaluation strategies are possible to express it is possible to be very precise about which evaluation strategy you mean within the language itself.

It will be instructive to recall the translation of both a CBV and a CBN calculus into the CBPV calculus. Our translation will have a similar flavour to the CBN translation but later we will also reflect on what would have been different with a CBV translation style in Section 6.4.

CBV Translation

In a CBV lambda calculus the arguments to functions are evaluated to values before β-reduction substitutes the value. In other words, variables denote values. When combined with effects this means that effects take place before a lambda is reduced. Figure 6.6 gives the translation rules of a CBV language into CBPV. Of the translation it is important to notice that variables are bound to normal CBPV value types and hence CBV values are mapped directly to CBPV values. CBV expressions are mapped to CBPV computations of type $F\ \tau$. The **let** expression is used to evaluate the arguments

$$\boxed{CBV(\tau_L), CBV(L)}$$

$$\begin{aligned}
CBV(Unit) &:= Unit \\
CBV(\tau_1 \to \tau_2) &:= U(CBV(\tau_1) \to F\,(CBV(\tau_2))) \\
CBV(()) &:= () \\
CBV(x) &:= x \\
CBV(\lambda x.L) &:= \{\lambda x.CBV(L)\} \\
CBV(L_1 L_2) &:= \textbf{let}\ x_1 \leftarrow CBV(L_1)\ \textbf{in let}\ x_2 \leftarrow CBV(L_2)\ \textbf{in}\ !(x_1)x_2
\end{aligned}$$

Figure 6.6: CBV Translation

$$\boxed{CBN(\tau_L), CBN(L)}$$

$$\begin{aligned}
CBN(Unit) &:= F\,Unit \\
CBN(\tau_1 \to \tau_2) &:= U(CBN(\tau_1)) \to CBN(\tau_2) \\
CBN(()) &:= \mathbf{return}\ () \\
CBN(x) &:= !(x) \\
CBN(\lambda x.L) &:= \lambda x.CBN(L) \\
CBN(L_1 L_2) &:= CBN(L_1)\{CBN(L_2)\}
\end{aligned}$$

Figure 6.7: CBN Translation

of a function before the resulting values are applied to one another. In particular in a CBV calculus, an abstraction is an example of a value and therefore must map to a CBPV value as well. In CBPV an abstraction is a computation therefore the translation must embed the computation into a value using the thunk operation ($\{\}$).

CBN Translation

In the CBN strategy (Figure 6.7), variables are bound to expressions, and each time a variable is evaluated the expression is evaluated again. In a CBPV calculus, variables always denote values and therefore to faithfully represent a CBN strategy the translation uses uses *thunks* and all variables are bound to values of type $U\ \rho$ for some computation type ρ. Variables are therefore translated to $!(x)$ (as x will have type $U\ \rho$). In opposition to the CBV case, the application case does not use **let** in order to evaluate the arguments before performing β-reduction and therefore it is clear that the effects of a particular expression will be repeated at each usage of the variable rather than once before the application is reduced.

The CBN translation is the one which is more interesting to us and will form the basis of the ideas behind our extension to CBPV. If we recall the problem at hand, we wanted to design a language which delayed the evaluation of specific fragments until we knew which stage the fragment needed to be evaluated into. The idea is therefore to adopt a similar elaboration as the CBN translation, variables are bound to unevaluated expressions but the forcing operation will also be able to interpret an unevaluated expression into any future stage. A CBV style translation will be briefly considered in Section 6.4.

6.2.1 Making CBPV Multi-Stage

Multi-stage programming is also concerned with the evaluation order of terms. A multi-stage program is separated into levels so that certain parts of the program can be evaluated before the remainder of the program. Multi-stage calculi are also already equipped with quotation and splicing in order to control which level each expression is at and hence in which order they should be evaluated.

Further to this analogy, the reduction semantics of the thunking and forcing operation of CBPV are similar to the normal formalisms of quote and splice from multi-stage programming. For example, the EFORCE rule from a typical CBPV language describes how the force and thunk operations, when combined, reduce.

$$\frac{}{!\{\mathsf{e}\} \rightsquigarrow \mathsf{e}}\ \textsc{EForce}$$

Forcing a delayed computation allows the computation to start again. The evaluation rule is precisely the same in a typical multi-stage calculus for the ESPLICE rule which cancels the combination of a splice and a quote.

$$\frac{}{\$(\llbracket\, \mathsf{e} \,\rrbracket) \rightsquigarrow \mathsf{e}}\ \textsc{ESplice}$$

The difference in CBPV is that the calculus is not level indexed in the original presentation. Creating a thunk does not increase the current level in the same way as in multi-stage calculus a quote increases the level of the term.

The purpose of level indexing in multi-stage programming is to ensure that variables are only used at the level they are bound. Once this is enforced, it's possible to fully

evaluate each level one at a time so that each stage generates the next stage.

Level indexing only affects variable binding, quotations increase the current level and variables introduced by binding structures are assigned this level. All variable uses are then checked against the current level in order to determine that the program is level separated.

Making CBPV a levelled language is as simple as adding a level index which is incremented by a thunk operator and decreased by a forcing operator. Most pieces of syntax just propagate the index, it is only the binding structures which need to be careful with the levels.

For example the following term is well typed in a normal CBPV presentation but is ill-staged in our language.

$$\lambda \mathsf{x} : \mathsf{Int}.(\mathbf{return}\ [\![\ \mathbf{return}\ \mathsf{x}\ ]\!])$$

The expression is ill-staged because the variable x is bound at an earlier level than it is used inside the thunk. Therefore the first modification to the CBPV language is to index typing judgements by a level index to ensure that variables are only used at the level they are bound and therefore a staged interpretation is possible.

Interpretation Functions

The distinction between values and computations also gives us an elegant way to understand the concepts of cross-stage persistence.

The idea is to embed a syntactic representation of a common simple language into values, think of the terms of the untyped lambda calculus. This syntax is not only unevaluated but uninterpreted. Interpretation functions are added as computations to the source language and translate the values of the simple embedded language into computations themselves. The meta syntax is embedded into the CBPV values using the **meta** m syntax. It should be thought of as similar to the $\{e\}$ syntax from a usual CBPV presentation but the argument is even less evaluated than the frozen computation from CBPV as the argument has not yet been translated into the proper CBPV language.

Translation is the role of interpretation functions. For example, there could be one interpretation function to interpret the simple lambda calculus in a CBV manner and one into a CBN interpretation. The choice about how to perform the interpretation is moved to runtime rather than an a priori decision based on a meta-theoretic translation function as in the original presentation. An uninterpreted expression is evident in the type. How long the argument is delayed for depends on the definition of the interpretation function.

The primary example of an interpretation function is one which interprets the generic syntax into different stages. This function is denoted by $\downarrow^{\mathsf{i}}_{\mathsf{g}}\ \mathsf{m}$ and indicates that the piece of meta syntax m should be interpreted at level i and lifted g levels. This function can interpret a piece of generic syntax into an arbitrary future stage. For example, $\downarrow^{0}_{0} \mathsf{m}$ interprets a piece of meta syntax directly into the current stage but $\downarrow^{0}_{1} \mathsf{m}$ will produce a value at level 0 which represents a value at level 1 as the normal one-stage lifting function. The purpose of this operation will be to support a lifting operation.

Once it is possible to interpret the meta syntax into any future stage, it provides a suitable core language for compiling a language which supports cross stage persistence of any term to any stage. The central idea of the compilation is that all terms of the source language are compiled to the generic meta syntax and further interpretation is delayed until necessary. The interpretation is similar to the CBN interpretation.

In order to do this, the translation will map all variables to values which represent unevaluated meta expressions. This can be seen in the definition of $\mathsf{Interpret}$, as the function type maps to a function from $\mathsf{Meta_s\ a}$ to the result type. This is similar to how in the CBN interpretation all variables map to values of type $\mathsf{U\ a}$. The translation is modified so that an unevaluated expression is represented by $\mathsf{Meta_s\ a}$ rather than $\mathsf{U\ a}$.

The addition of levels and the introduction of meta syntax and interpretation are the two key innovations in the calculus.

6.2.2 Lifting

The final flourish to the calculus is to add the $\uparrow\ \cdot$ operation which converts a CBPV value into the meta syntax. The operation is not supported for all types, just as lift is not normally supported for all types. This operation can be used to decompose the traditional lift operation from multi-stage languages into a composition of the new lift and an interpretation function. As an example, value integers can be translated to meta integers and then interpreted into future stage integers at a later point.

This operator can be used to implement a more efficient but less powerful lifting operator in the style found in most multi-stage languages. If your language does not support arbitrary cross-stage references then it is preferable to reduce the runtime costs of interpretation by compiling to the fragment of the core language which doesn't do runtime syntax interpretation.

Intermediate Summary In this section we have introduced the idea behind the modifications to the CBPV language which form the heart of our calculus. In the next section these modifications will be presented formally before we will discuss other points in the design space that we could have chosen in Section 6.4.

To recap the ideas that were introduced in this section:

- CBPV is natural to consider for a multi-stage language as you can already precisely express evaluation order.
- With the addition of level indexing, it is a suitable basis for a multi-stage language.
- By adding more structure to the "thunk" type and moving the interpretation function into the language, it becomes a flexible model of cross-stage persistence as well.
- Traditional lifting can also be understood in the CBPV manner as converting a value into an unevaluated meta expression.

6.3 Formalism

The syntax for four different languages is given in Figure 6.8. The source language defined by S, is the language which will allow you to use any variable at any future stage. The meaning of S is given by elaborating S into the CBPV core language which contains by computations C and values V. The elaboration firstly interprets S into the meta language M before being embedded into the computation language C using $\downarrow_0^0 (\textbf{meta}\ \mathsf{m})$. Some examples of how some source programs are compiled and evaluated are given in Section 6.3.11.

Along with the expression syntax, there are program theories for both the source ($\mathbb{S}$) and core language ($\mathbb{P}$). Each language also has a corresponding simple type system (τ_V, τ_C, τ_M and τ_S) there are also numerous environments which will be introduced as necessary.

6.3.1 The Source Language

The source language (Figure 6.9) is a simple lambda calculus containing quoting, splicing and cross-stage persistence. The key difference in the language compared to the normal presentation is that variables can be used at future levels other than the one they are bound without restriction. This includes function types which are typically not

$$\mathbb{P} ::= \textbf{main}\ \mathsf{C} \mid \mathsf{v} = \mathsf{C}; \mathbb{P}$$

$$\begin{aligned}
\mathsf{C} ::= {} & \textbf{split}\ \mathsf{V}\ \textbf{by}\ \mathsf{v}.\mathsf{v}.\mathsf{C} \\
\mid {} & \textbf{case0}\ \mathsf{V} \\
\mid {} & \langle\rangle \\
\mid {} & \textbf{case}\ \mathsf{V}\ \textbf{of}\ \mathsf{v} \to \mathsf{C}; \mathsf{v} \to \mathsf{C} \\
\mid {} & !\mathsf{V} \\
\mid {} & \downarrow^{\mathsf{n}}_{\mathsf{n}} \mathsf{V} \\
\mid {} & \uparrow \mathsf{V} \\
\mid {} & \textbf{return}\ \mathsf{V} \\
\mid {} & \textbf{let}\ \mathsf{v} \leftarrow \mathsf{C}\ \textbf{in}\ \mathsf{C} \\
\mid {} & \lambda \mathsf{v} : \tau_V . \mathsf{C} \\
\mid {} & \mathsf{C}\ \mathsf{V} \\
\mid {} & \langle \mathsf{C}, \mathsf{C} \rangle \\
\mid {} & \iota_L\ \mathsf{C} \\
\mid {} & \iota_R\ \mathsf{C}
\end{aligned}$$

$$\begin{aligned}
\mathsf{V} ::= {} & \mathsf{v} \\
\mid {} & 1 \\
\mid {} & (\mathsf{V}, \mathsf{V}) \\
\mid {} & \mathsf{L}\ \mathsf{V} \\
\mid {} & \mathsf{R}\ \mathsf{V} \\
\mid {} & [\![\ \mathsf{C}\ ]\!] \\
\mid {} & \textbf{meta}\ \mathsf{M} \\
\mid {} & \textbf{num}_v\ \mathsf{Int}
\end{aligned}$$

$$\mathbf{V} ::= \mathsf{V} \mid [\![\ \mathsf{C}\ ]\!]_\Sigma$$

$$\begin{aligned}
\mathsf{T} ::= {} & \textbf{val}\ \mathbf{V} \mid \textbf{clos}\ \Sigma\ \mathsf{v}\ \mathsf{C} \\
\mid {} & () \mid (\mathsf{C} \times \mathsf{C})
\end{aligned}$$

$$\begin{aligned}
\mathsf{M} ::= {} & \textbf{num}_m\ \mathsf{Int} \\
\mid {} & \lambda \mathsf{m} : \tau_M . \mathsf{M} \\
\mid {} & \mathsf{M}\ \mathsf{M} \\
\mid {} & \Uparrow_{\mathsf{n}}\ \mathsf{m} \\
\mid {} & \mathsf{m} \\
\mid {} & [\![\ \mathsf{M}\ ]\!]_\Theta
\end{aligned}$$

$$\Theta ::= \bullet \mid \mathsf{m} : \tau_M \mapsto \mathsf{M}, \Theta \mid \Theta \cup \Theta$$

$$\begin{aligned}
\mathsf{S} ::= {} & \textbf{num}_S\ \mathsf{Int} \\
\mid {} & \lambda \mathsf{s}.\mathsf{S} \\
\mid {} & \mathsf{S}\ \mathsf{S} \\
\mid {} & \mathsf{s} \\
\mid {} & [\![\ \mathsf{S}\ ]\!] \\
\mid {} & \$(\mathsf{S})
\end{aligned}$$

$$\mathbb{S} ::= \textbf{main}\ \mathsf{S} \mid \mathsf{s} = \mathsf{S}; \mathbb{S}$$

$$\tau_C ::= \mathsf{F}\ \tau_V \mid \tau_V \to \tau_C \mid \top \mid \tau_C \times \tau_C$$

$$\begin{aligned}
\tau_V ::= {} & \mathbf{1} \mid (\tau_V, \tau_V) \mid \mathbf{0} \mid \tau_V + \tau_V \mid \mathsf{U}\ \tau_C \\
\mid {} & \mathsf{Int} \mid \mathsf{Meta}_{\mathsf{n}}\ \tau_M
\end{aligned}$$

$$\tau_M ::= \mathsf{Int}_m \mid \tau_M \to_m \tau_M \mid \mathsf{Code}_m \tau_M$$

$$\tau_S ::= \mathsf{Int}_s \mid \tau_S \to_s \tau_S \mid \mathsf{Code}_s \tau_S$$

$$\Xi ::= \bullet \mid \Xi, \mathsf{s} : (\mathsf{n}, \tau_S)$$

$$\Gamma ::= \bullet \mid \Gamma, \mathsf{v} : (\mathsf{n}, \tau_V)$$

$$\begin{aligned}
\Sigma ::= {} & \bullet \mid \Sigma, \mathsf{v} \to \mathbf{V} \\
\mid {} & \Sigma, \mathsf{m} \to \mathsf{C}
\end{aligned}$$

$$\Delta ::= \bullet \mid \Delta, \mathsf{m} : (\mathsf{n}, \tau_M)$$

$$\mathbb{S} ::= \bullet \mid \mathbb{S}, \mathsf{s} : \tau_S$$

$$\mathbb{P} ::= \bullet \mid \mathbb{P}, \mathsf{v} : \tau_V$$

$$\mathbb{P}_\Downarrow ::= \bullet \mid \mathbb{P}_\Downarrow, \mathsf{v} \mapsto \mathbf{V}$$

$$\begin{aligned}
\mathsf{CE} ::= {} & \bullet \\
\mid {} & |< \mathsf{Splice}\ \mathsf{m}\ \tau_M >\ \mathsf{M}\ \mathsf{CE} \\
\mid {} & |< \mathsf{Lift}\ \mathsf{n}\ \mathsf{m}\ \tau_M\ \mathsf{M} > \\
\mid {} & \mathsf{CE} \cup \mathsf{CE}
\end{aligned}$$

$$\Upsilon ::= \bullet \mid \mathsf{m} \to \mathsf{m}, \Upsilon$$

$$\mathsf{SS} ::= \bullet \mid \Theta : \mathsf{SS}$$

Figure 6.8: Syntax

$$\boxed{\Xi \vdash_{\mathsf{n}} \mathsf{s} : \tau^s \rightsquigarrow \mathsf{m}; \mathsf{CE}}$$

$$\frac{\mathsf{v} : (\tau^s, \mathsf{k}) \in \Xi \qquad k < n \qquad \text{fresh } \mathsf{v}'_m}{\Xi \vdash_{\mathsf{n}} \mathsf{v} : \tau^s \rightsquigarrow \mathsf{v}_m; < \mathsf{Lift}\ (\mathsf{n} - \mathsf{k})\ \mathsf{v}'_m\ (\mathsf{Interpret}_m\ \tau^s)\ (\Uparrow_{(\mathsf{n}-\mathsf{k})}\ \mathsf{v}_m) >}\ \text{S\_VAR\_CSP}$$

$$\frac{\mathsf{v} : (\tau^s, \mathsf{k}) \in \Xi \qquad k = n}{\Xi \vdash_{\mathsf{n}} \mathsf{v} : \tau^s \rightsquigarrow \mathsf{v}_m; \bullet}\ \text{S\_VAR} \qquad \frac{\mathsf{v} : \tau^s \in \mathbb{S}}{\Xi \vdash_{\mathsf{n}} \mathsf{v} : \tau^s \rightsquigarrow \mathsf{v}_m; \bullet}\ \text{S\_PROG\_VAR}$$

$$\frac{}{\Xi \vdash_{\mathsf{n}} \mathbf{num}_S\ \mathsf{n} : \mathsf{Int}_s \rightsquigarrow \mathbf{num}_m\ \mathsf{n}; \bullet}\ \text{S\_NUM}$$

$$\frac{\Xi, \mathsf{v} : (\mathsf{n}, \tau_1^s) \vdash_{\mathsf{n}} \mathsf{s} : \tau_2^s \rightsquigarrow \mathsf{m}; \mathsf{CE}}{\Xi \vdash_{\mathsf{n}} \lambda \mathsf{v}.\mathsf{s} : \tau_1^s \to_s \tau_2^s \rightsquigarrow \lambda \mathsf{v}_m : \mathsf{Interpret}_m\ \tau_1^s . \mathsf{m}; \mathsf{CE}}\ \text{S\_LAM}$$

$$\frac{\Xi \vdash_{\mathsf{n}} \mathsf{s}_1 : \tau_1^s \to_s \tau_2^s \rightsquigarrow \mathsf{m}_1; \mathsf{CE}_1 \qquad \Xi \vdash_{\mathsf{n}} \mathsf{s}_2 : \tau_1^s \rightsquigarrow \mathsf{m}_2; \mathsf{CE}_2}{\Xi \vdash_{\mathsf{n}} \mathsf{s}_1\ \mathsf{s}_2 : \tau_2^s \rightsquigarrow \mathsf{m}_1\ \mathsf{m}_2; \mathsf{CE}_1 \cup \mathsf{CE}_2}\ \text{S\_APP}$$

$$\frac{\Xi \vdash_{(\mathsf{n}+1)} \mathsf{s} : \tau^s \rightsquigarrow \mathsf{m}; \mathsf{CE}' \qquad \mathsf{SplitEnv}(\mathsf{CE}) = (\mathsf{SPE}, \mathsf{CE}')}{\Xi \vdash_{\mathsf{n}} [\![\ \mathsf{s}\ ]\!] : \mathsf{Code}_s \tau^s \rightsquigarrow [\![\ \mathsf{m}\ ]\!]_{\mathsf{SPE}}; \mathsf{CE}'}\ \text{S\_QUOTE}$$

$$\frac{\Xi \vdash_{(\mathsf{n}-1)} \mathsf{s} : \mathsf{Code}_s \tau^s \rightsquigarrow \mathsf{m}; \mathsf{CE} \qquad \text{fresh } \mathsf{v}_m}{\Xi \vdash_{\mathsf{n}} \$(\mathsf{s}) : \tau^s \rightsquigarrow \mathsf{v}_m; < \mathsf{Splice}\ \mathsf{v}_m\ (\mathsf{Interpret}_m\ \tau^s) >\ \mathsf{m}\ \mathsf{CE}}\ \text{S\_SPLICE}$$

Figure 6.9: Source Expression Typing

$$\begin{aligned}
&\mathsf{Interpret}_m :: \tau_S \to \tau_M \\
&\mathsf{Interpret}_m\ (\mathsf{a} \to_s \mathsf{b}) = (\mathsf{Interpret}_m\ \mathsf{a}) \to_m (\mathsf{Interpret}_m\ \mathsf{b}) \\
&\mathsf{Interpret}_m\ \mathsf{Int}_s \quad\ = \mathsf{Int}_m \\
&\mathsf{Interpret}_m\ (\mathsf{Code}_s \mathsf{m}) = \mathsf{Code}_m (\mathsf{Interpret}_m\ \mathsf{a})
\end{aligned}$$

Figure 6.10: Definition of $\mathsf{Interpret}_m$

$$\boxed{\mathsf{SplitEnv}(\mathsf{CE}) = (\mathsf{SPE}, \mathsf{CE}')}$$

$$\frac{}{\mathsf{SplitEnv}(\bullet) = (\bullet, \bullet)}\ \textsc{Split\_Empty}$$

$$\frac{\mathsf{SplitEnv}(\mathsf{CE}_1) = (\mathsf{SPE}_1, \mathsf{CE}_1') \qquad \mathsf{SplitEnv}(\mathsf{CE}_2) = (\mathsf{SPE}_2, \mathsf{CE}_2')}{\mathsf{SplitEnv}(\mathsf{CE}_1 \cup \mathsf{CE}_2) = (\mathsf{SPE}_1 \cup \mathsf{SPE}_2, \mathsf{CE}_1' \cup \mathsf{CE}_2')}\ \textsc{Split\_Union}$$

$$\frac{}{\mathsf{SplitEnv}(< \mathsf{Splice}\ \mathsf{v}_m\ \tau^m >\ \ \mathsf{m}\ \mathsf{CE}) = (\mathsf{v}_m : \tau^m \mapsto \mathsf{m}, \bullet, \mathsf{CE})}\ \textsc{Split\_Splice}$$

$$\frac{}{\mathsf{SplitEnv}(< \mathsf{Lift}\ 0\ \mathsf{v}_m\ \tau^m\ \mathsf{m} >) = (\mathsf{v}_m : \tau^m \mapsto \mathsf{m}, \bullet, \bullet)}\ \textsc{Split\_Lift\_Zero}$$

$$\frac{\mathsf{fresh}\ \mathsf{v}_m' \qquad \mathsf{SPE}' = (\mathsf{v}_m : \tau^m \mapsto (\Uparrow_0 \mathsf{v}_m'), \bullet)}{\mathsf{SplitEnv}(< \mathsf{Lift}\ (\mathsf{k}+1)\ \mathsf{v}_m\ (\mathsf{Code}_m \tau^m)\ \mathsf{m} >) = (\mathsf{SPE}', < \mathsf{Lift}\ \mathsf{k}\ \mathsf{v}_m'\ \tau^m\ \mathsf{m} >)}\ \textsc{Split\_Lift\_N}$$

Figure 6.11: Splitting a Compilation Environment

possible to be lifted. Otherwise the typing rules for abstraction, application, quotation and splicing are standard as found in other multi-stage languages.

The source language is elaborated into the meta language where any splices are replaced with a splice environment and any implicit lifting is replaced by explicit lifting. The elaboration is given in Figure 6.9 and distinguished by a grey box around the relevant part of the rule. The elaboration judgement is of the form $\Xi \vdash_{\mathsf{n}} \mathsf{s} : \tau^s \rightsquigarrow \mathsf{m}; \mathsf{CE}$ indicates that in environment Ξ that the expression s has types τ^s at level n and elaborates to meta expression m with the compile environment CE. The compile environment is used to remove nested splices from the source language which are convenient for a programmer but inconvenient formally. The elaboration is described in Section 6.3.3.

6.3.2 The Meta Language

There is a lot of scope for designing a meta language (Figure 6.12), it acts as a compilation target for the source language and therefore has been designed in this case to mirror the source language closely. The second criteria is that it can be interpreted into the CBPV core language. There are two points to note in particular about the meta language. Both of these modifications are intended to make specifying reduction semantics for the language easier.

$$\boxed{\Delta \vdash_{\mathsf{n}} \mathsf{m} : \tau^m}$$

$$\frac{}{\Delta \vdash_{\mathsf{n}} \mathbf{num}_m\ \mathsf{n} : \mathsf{Int}_m}\ \text{M\_NUM} \qquad \frac{\Delta, \mathsf{v}_m : (\mathsf{n}, \tau_1^m) \vdash_{\mathsf{n}} \mathsf{m} : \tau_2^m}{\Delta \vdash_{\mathsf{n}} \lambda \mathsf{v}_m : \tau_1^m . \mathsf{m} : \tau_1^m \to_m \tau_2^m}\ \text{M\_LAM}$$

$$\frac{\Delta \vdash_{\mathsf{n}} \mathsf{m}_1 : \tau_1^m \to_m \tau_2^m \qquad \Delta \vdash_{\mathsf{n}} \mathsf{m}_2 : \tau_1^m}{\Delta \vdash_{\mathsf{n}} \mathsf{m}_1\ \mathsf{m}_2 : \tau_2^m}\ \text{M\_APP} \qquad \frac{\mathsf{v}_m : (\tau^m, \mathsf{k}) \in \Delta}{\Delta \vdash_{\mathsf{k}} \Uparrow_{\mathsf{i}} \mathsf{v}_m : \mathsf{C}\ \mathsf{i}\ \tau^m}\ \text{M\_LIFT}$$

$$\frac{\Delta \cup \mathsf{SPE} \vdash_{(\mathsf{k}+1)} \mathsf{m} : \tau^m \qquad \Delta \vdash^{\mathsf{k}}_{\mathsf{SP}} \mathsf{SPE}}{\Delta \vdash_{\mathsf{k}} [\![\ \mathsf{m}\ ]\!]_{\mathsf{SPE}} : \mathsf{Code}_m \tau^m}\ \text{M\_QUOTE}$$

Figure 6.12: Meta Expression Typing

$$\mathsf{C} :: \mathsf{Level} \to \tau_M \to \tau_M$$
$$\mathsf{C}\ 0\ \mathsf{t} = \mathsf{t}$$
$$\mathsf{C}\ \mathsf{n}\ \mathsf{t} = \mathsf{Code}_m(\mathsf{C}\ (\mathsf{n} - 1)\ \mathsf{t})$$

Figure 6.13: Definition of C

$$\boxed{\Delta \vdash^{\mathsf{k}}_{\mathsf{SP}} \mathsf{SPE}}$$

$$\frac{}{\Delta \vdash^{\mathsf{k}}_{\mathsf{SP}} \bullet}\ \text{SP\_EMPTY} \qquad \frac{\Delta \vdash_{\mathsf{k}} \mathsf{m} : \tau^m \qquad \Delta \vdash^{\mathsf{k}}_{\mathsf{SP}} \mathsf{SPE}}{\Delta \vdash^{\mathsf{k}}_{\mathsf{SP}} \mathsf{v}_m : \tau^m \mapsto \mathsf{m}, \mathsf{SPE}}\ \text{SP\_CONS}$$

Figure 6.14: Typing a Meta Splice Environment

- Splices are replaced in favour of splice environments attached to a quotation. A splice environment binds variables available inside the quotation to values computed by executing code at a prior stage. A splice in the source language is replaced by a variable bound in a splice environment.

- The lifting combinator $\Uparrow_{\mathsf{k}} \mathsf{v}_m$ lifts a meta expression k levels into the future. It replaces variables in the language, and as a result the argument can only be a variable. Lifting a variable 0 levels is the identity function.

These two changes highlight the key difference between the source language and meta language. The source syntax has both implicit cross-stage persistence and nested splices. Evaluation semantics with these two features are more difficult to specify so they are both removed during compilation. Cross-stage references are replaced with the standard combination of lifts and splices. The meta-language has a construct to directly lift a value n levels into the future to avoid needing to nest lifts. Nested splices are replaced in favour of a splice environment. Each quotation in the meta syntax has an associated splice environment which is a map from a variable at level $n + 1$ to an expression of type $\mathsf{Code}_m\mathsf{a}$ at level n. Each of the variables is in scope inside inside the body of the quotation, and when the quote is evaluated, the definition will be evaluated to a quotation and then the quoted expression bound to the variable at the next stage.

The way to use a variable is with the $\Uparrow_{\mathsf{k}} \mathsf{v}_m$ syntax. A 0 level ($\Uparrow_0 \mathsf{v}_m$) lifting is interpreted as the identity, and allows a variable to be used as normal without performing any promotion. The argument to $\Uparrow_{\mathsf{k}} \cdot$ must be a variable, you may expect it to be any meta expression. This decision is motivated by the compilation to the CBPV calculus where ensuring the argument to the lifting constructor is a variable means it must be an evaluated value. Also in the source language, there is only implicit lifting so it is only variables which are ever lifted.

6.3.3 Compilation

The overall idea of compilation is as follows: firstly a term in the source language is compiled into a meta syntax term. The meta syntax term is then embedded into the core calculus using the interpretation function $(\downarrow^0_0 \cdot)$ which interprets the meta syntax into the CBPV calculus before the standard CBPV evaluation rules apply.

The first stage of compilation is given in Figure 6.9, the elaboration takes a source language term (which we have already seen) and translates it into a meta syntax term along with a environment which records information about the splices.

The addition of splice environments is exploited in Section 3.4 and reflects common practice in implementations of multi-stage languages where ensuring a staged evaluation is simplified by removing nested splices from a term. This means the operational semantics do not have to traverse inside quotations whilst performing evaluation.

The Compile Environment

In order to convert a language with nested splices into one with splice environments, when compiling the language, it is necessary to collect the splices contained within a quotation and convert them into a splice environment. Therefore the compilation function returns both a compiled expression and what we call a *compile environment* (CE) which is translated into the splice environment for a particular quotation.

Splices arise in two different ways in the source language. Either explicitly as a nested splice or implicitly as we compile a variable used across stages to a combination of a splice and a lift. Therefore the compile environment contains two cases for these two scenarios.

During compilation, fresh variables are generated to replace anywhere where a splice would occur and a new entry is added into the compile environment. In the case of a nested splice, the compiled expression and the compilation environment resulting from the compilation is stored in the environment. This is important as any nested splices need to added to splice environments of outer quotations.

In the case of implicit lifting, the situation is a little more complicated because if a variable is used 2 levels after it is bound, then bindings need to be created in the splice environments of both surrounding quotations. Therefore the case for an implicit lift stores how many levels to lift and also the expression to insert when we get to that point. For example, consider the source program:

$$\mathsf{e}_1 = \lambda \mathsf{v}.[\![\, [\![\, \mathsf{v} \,]\!] \,]\!]$$

Then this will be compiled to:

$$\mathsf{e}'_1 = \lambda \mathsf{v}_m : \mathsf{Int}_m.[\![\, [\![\, \mathsf{mv''} \,]\!]_{(\mathsf{mv''}:\mathsf{Int}_m \mapsto \mathsf{v}'_m, \bullet)} \,]\!]_{(\mathsf{v}'_m : (\mathsf{Code}_m \mathsf{Int}_m) \mapsto (\Uparrow_2 \mathsf{v}_m), \bullet)}$$

As the SPLIT_LIFT_N (Figure 6.11) judgement will create the intermediate binding on the nested quotation.

After the body of a quotation has been compiled, the compile environment is consulted in order to build the splice environment. This is what the $\mathsf{SplitEnv}(\cdot) = (\cdot, \cdot)$ function

$$
\begin{aligned}
&\mathsf{Interpret} :: \mathsf{Level} \to \tau_M \to \tau_C \\
&\mathsf{Interpret\ s\ (a} \to_m \mathsf{b)} \ = (\mathsf{Meta_s\ a}) \to (\mathsf{Interpret\ s\ b}) \\
&\mathsf{Interpret\ s\ Int}_m \qquad\quad = \mathsf{F\ Int} \\
&\mathsf{Interpret\ s\ (Code}_m\mathsf{m)} = \mathsf{F\ (U\ (F\ (Meta}_{(\mathsf{S\ s})}\ \mathsf{m})))
\end{aligned}
$$

Figure 6.15: Definition of Interpret

$$
\begin{aligned}
&\mathsf{Uninterpret} :: \tau_V \to \tau_M \\
&\mathsf{Uninterpret\ Int} = \mathsf{Int}_m
\end{aligned}
$$

Figure 6.16: Definition of Uninterpret

defined in Figure 6.11 is for. It takes a compile environment and splits it into a new compile environment and a splice environment which is attached to the current quotation.

The new compile environment can contain new nested splices from either of the two cases. Either there are further nested splices inside an explicit quotation, which are within the nested compile environment or the implicitly lifted variable still needs to be lifted more levels before it should be inserted. In this latter case a new fresh variable is created for the current splice environment and the target number decreased by one.

The typing rules for the source language ensure that each nested splice can be bound on a quotation in this way because splices are only permitted to appear at positive levels. Unlike in our previous formalism, there are no negative levels in the source language.

Once a source language term has been compiled to a meta term, it is embedded into the core language using the interpretation function $\downarrow_0^0$ (**meta** m). Further evaluation of the meta syntax, into an eventual value happens during normal interpretation which we will discuss in detail in the next section.

$$\boxed{\mathsf{Lift}\ \tau^v}$$

$$\frac{}{\mathsf{Lift\ Int}}\ \textsc{Lift\_Int}$$

Figure 6.17: Definition of Lift

$$\boxed{\Gamma \vdash_{\mathsf{n}} \mathsf{v} : \tau^v}$$

$$\frac{\mathsf{x} : (\tau^v, \mathsf{n}) \in \Gamma}{\Gamma \vdash_{\mathsf{n}} \mathsf{x} : \tau^v}\ \text{V\_VAR} \qquad \frac{\mathsf{x} : \tau^v \in \mathbb{P}}{\Gamma \vdash_{\mathsf{n}} \mathsf{x} : \tau^v}\ \text{V\_PROG\_VAR} \qquad \frac{}{\Gamma \vdash_{\mathsf{n}} 1 : \mathbf{1}}\ \text{V\_UNIT}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v}_1 : \tau_1^v \qquad \Gamma \vdash_{\mathsf{n}} \mathsf{v}_2 : \tau_2^v}{\Gamma \vdash_{\mathsf{n}} (\mathsf{v}_1, \mathsf{v}_2) : (\tau_1^v, \tau_2^v)}\ \text{V\_PROD} \qquad \frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v}_1 : \tau_1^v}{\Gamma \vdash_{\mathsf{n}} \mathsf{L}\ \mathsf{v}_1 : \tau_1^v + \tau_2^v}\ \text{V\_SUM\_L}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v}_2 : \tau_2^v}{\Gamma \vdash_{\mathsf{n}} \mathsf{R}\ \mathsf{v}_2 : \tau_1^v + \tau_2^v}\ \text{V\_SUM\_R} \qquad \frac{\Gamma \vdash_{(\mathsf{n}+1)} \mathsf{c} : \tau^c}{\Gamma \vdash_{\mathsf{n}} [\![\, \mathsf{c} \,]\!] : \mathsf{U}\ \tau^c}\ \text{V\_THUNK}$$

$$\frac{\Delta \vdash_{\mathsf{k}} \mathsf{m} : \tau^m}{\Gamma \vdash_{\mathsf{n}} \mathbf{meta}\ \mathsf{m} : \mathsf{Meta}_{\mathsf{k}}\ \tau^m}\ \text{V\_META}$$

Figure 6.18: Value Typing

6.3.4 Core Language

The core language (Figure 6.19, Figure 6.18) is mostly a standard CBPV calculus with the addition of the extensions sketched in Section 6.2.1. In particular the value syntax is extended with **meta** m to embed the meta language into the value language. The language is level indexed and hence quotation $[\![\cdot]\!]$ and splicing $!\cdot$ modify the level whilst variables in the environment are level indexed. The computation language also includes two new pieces of syntax $\downarrow^{\cdot} \cdot$ for interpreting meta-syntax as a computation and $\uparrow \cdot$ for converting values to meta-syntax. The remainder of the rules are standard for a CBPV language and are not affected by the level indexing apart from making sure to introduce binders at the correct level in the environment.

Typing and Interpreting Meta Syntax

The most important extension to the calculus is the typing rules for $\downarrow^{\cdot} \cdot$ (C_INTERPRET, Figure 6.19) and the corresponding evaluation of this construct defined by $\Sigma; \mathsf{m} \mathbin{@} \mathsf{i} \Downarrow_m \mathsf{T}$ (Figure 6.27). The judgement says that in environment Σ the meta expression m interpreted into level i results in a terminal value T.

The typing rule is given by C_INTERPRET and shows how the internal level of the meta syntax interacts with the ambient level of the computation. Both the CBPV language and meta language typing rules are indexed by a level. The CBPV typing judgement is indexed by a level which acts to ensure that expressions at level k are evaluated before expressions at level $k + 1$. The meta syntax is also indexed by a level, which

$$\boxed{\Gamma \vdash_{\mathsf{n}} \mathsf{c} : \tau^c}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v} : (\tau_1^v, \tau_2^v) \qquad (\Gamma, \mathsf{v}_1 : (\mathsf{n}, \tau_1^v), \mathsf{v}_2 : (\mathsf{n}, \tau_2^v)) \vdash_{\mathsf{n}} \mathsf{c} : \tau^c}{\Gamma \vdash_{\mathsf{n}} \textbf{split } \mathsf{v} \textbf{ by } \mathsf{v}_1.\mathsf{v}_2.\mathsf{c} : \tau^c} \text{ C\_SPLIT}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v} : \mathbf{0}}{\Gamma \vdash_{\mathsf{n}} \textbf{case0 } \mathsf{v} : \top} \text{ C\_CASE\_ZERO}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v} : \tau_1^v + \tau_2^v \qquad (\Gamma, \mathsf{v}_1 : (\mathsf{n}, \tau_1^v)) \vdash_{\mathsf{n}} \mathsf{c}_1 : \tau^c \qquad (\Gamma, \mathsf{v}_2 : (\mathsf{n}, \tau_2^v)) \vdash_{\mathsf{n}} \mathsf{c}_2 : \tau^c}{\Gamma \vdash_{\mathsf{n}} \textbf{case } \mathsf{v} \textbf{ of } \mathsf{v}_1 \to \mathsf{c}_1; \mathsf{v}_2 \to \mathsf{c}_2 : \tau^c} \text{ C\_CASE}$$

$$\frac{\Gamma \vdash_{(\mathsf{n}-1)} \mathsf{v} : \mathsf{U}\ \tau^c}{\Gamma \vdash_{\mathsf{n}} !\mathsf{v} : \tau^c} \text{ C\_FORCE} \qquad \frac{\Gamma \vdash_{(\mathsf{n}+\mathsf{i})} \mathsf{v} : \mathsf{Meta}_{\mathsf{n}}\ \tau^m}{\Gamma \vdash_{(\mathsf{n}+\mathsf{i})\downarrow_{\mathsf{g}}^{\mathsf{i}}} \mathsf{v} : \mathsf{Interpret\ n}\ (\mathsf{C\ g}\ \tau^m)} \text{ C\_INTERPRET}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v} : \tau^v \qquad \mathsf{Lift}\ \tau^v}{\Gamma \vdash_{\mathsf{n}}\uparrow \mathsf{v} : \mathsf{F}\ (\mathsf{Meta}_{\mathsf{n}}\ (\mathsf{Uninterpret}\ \tau^v))} \text{ C\_LIFT}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{c}_1 : \mathsf{F}\ \tau^v \qquad (\Gamma, \mathsf{v} : (\mathsf{n}, \tau^v)) \vdash_{\mathsf{n}} \mathsf{c}_2 : \tau^c}{\Gamma \vdash_{\mathsf{n}} \textbf{let } \mathsf{v} \leftarrow \mathsf{c}_1 \textbf{ in } \mathsf{c}_2 : \tau^c} \text{ C\_LET}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v} : \tau^v}{\Gamma \vdash_{\mathsf{n}} \textbf{return } \mathsf{v} : \mathsf{F}\ \tau^v} \text{ C\_RETURN} \qquad \frac{(\Gamma, \mathsf{v} : (\mathsf{n}, \tau^v)) \vdash_{\mathsf{n}} \mathsf{c} : \tau^c}{\Gamma \vdash_{\mathsf{n}} \lambda \mathsf{v} : \tau^v.\mathsf{c} : \tau^v \to \tau^c} \text{ C\_LAM}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{v} : \tau^v \qquad \Gamma \vdash_{\mathsf{n}} \mathsf{c} : \tau^v \to \tau^c}{\Gamma \vdash_{\mathsf{n}} \mathsf{c}\ \mathsf{v} : \tau^v \to \tau^c} \text{ C\_APP} \qquad \frac{\Gamma \vdash_{\mathsf{n}} \mathsf{c}_1 : \tau_1^c \qquad \Gamma \vdash_{\mathsf{n}} \mathsf{c}_2 : \tau_2^c}{\Gamma \vdash_{\mathsf{n}} \langle \mathsf{c}_1, \mathsf{c}_2 \rangle : \tau_1^c \times \tau_2^c} \text{ C\_PROD}$$

$$\frac{\Gamma \vdash_{\mathsf{n}} \mathsf{c}_1 : \tau_1^c \times \tau_2^c}{\Gamma \vdash_{\mathsf{n}} \iota_L\ \mathsf{c}_1 : \tau_1^c} \text{ C\_PROJ\_L} \qquad \frac{\Gamma \vdash_{\mathsf{n}} \mathsf{c}_2 : \tau_1^c \times \tau_2^c}{\Gamma \vdash_{\mathsf{n}} \iota_R\ \mathsf{c}_2 : \tau_2^c} \text{ C\_PROJ\_R}$$

Figure 6.19: Computation Typing

is internal to the meta syntax. It must also be levelled because the meta syntax contains multi-stage language features (quotes and splices). Therefore in order to ensure a well-levelled translation into the well-levelled core language the interpretation function must faithfully map levels of the meta syntax into levels of the core language.

Therefore the typing rule for $\downarrow^{\mathsf{i}}_{\mathsf{g}} \cdot$ explains that if the argument is a piece of meta syntax at level n, then interpreting that into level i will result in a computation at level $\mathsf{n} + \mathsf{i}$. In addition, the $\downarrow^{\mathsf{i}}_{\mathsf{g}} \cdot$ will also shift the syntax g levels forward so the result is of type $\mathsf{Interpret}\ \mathsf{n}\ (\mathsf{C}\ \mathsf{g}\ \tau^m)$. C (Figure 6.13) is a function which wraps g $\mathsf{Code}_m \cdot$ type constructors around the τ^m type.

The $\mathsf{Interpret}$ function (Figure 6.15) describes how to map a τ^m to a τ^c. A τ^m is mapped to a computation type and hence meta values are interpreted into computations. Each case of the definition is interesting. Numbers are mapped to a computation $\mathsf{F}\ \mathsf{Int}$, a computation which returns a number. Functions are interpreted into a CBPV function, the argument to a function is an unevaluated **meta** m value. A $\mathsf{Code}_m \mathsf{a}$ type is interpreted into $\mathsf{F}\ (\mathsf{U}\ (\mathsf{F}\ (\mathsf{Meta}_{(\mathsf{S}\ \mathsf{s})}\ \mathsf{a})))$, a computation which can perform some effects to return an unevaluated meta expression. The fact that the interpretation of the quote constructor could be effectful, means that we have to pay particular care to where the quotation is evaluated so that the effects from the outer F happen at a prior level to the effects of the delayed computation.

The **meta** m type is indexed by the internal stage of the meta syntax. This also explains why the $\mathsf{Interpret}$ function increases the level of enclosed expressions by one in the $\mathsf{Code}_m \cdot$ case, because any variables introduced inside a quotation will be bound at the subsequent meta level and hence translated to a computation where the argument is bound to a meta syntax expression at the subsequent level.

If the meta syntax did not contain levelled constructs then the typing of $\downarrow^{\mathsf{g}}_{\mathsf{k}} \mathsf{v}$ would be much more straightforward as the interpretation would not have combine together the two different notions of levels.

Evaluation of Meta Expressions

The C_EVAL_INTERPRET rule gives the evaluation of $\downarrow^{\mathsf{i}}_{\mathsf{g}} \mathsf{m}$. The evaluation proceeds in three stages. Firstly the meta syntax m is shifted g levels into the future which turns a meta expression at level n into an expression at level $\mathsf{n} + \mathsf{g}$. The process of shifting will be discussed in much more detail in the next section. Then $\mathsf{WrapShifted}(\mathsf{m}, \mathsf{SS}) = \mathsf{m}'$ (Figure 6.20) wraps the shifted expression in enough quotations to correct the level back to level i. This is the value analogue to the C function. Finally the main work is done

$$\boxed{\mathsf{WrapShifted}(\mathsf{m}, \mathsf{SS}) = \mathsf{m}'}$$

$$\frac{}{\mathsf{WrapShifted}(\mathsf{m}, \bullet) = \mathsf{m}}\ \textsc{WrapShifted\_Zero}$$

$$\frac{\mathsf{WrapShifted}(\mathsf{m}, \mathsf{SS}) = \mathsf{m}'}{\mathsf{WrapShifted}(\mathsf{m}, (\mathsf{SPE} : \mathsf{SS})) = [\![\ \mathsf{m}'\ ]\!]_{\mathsf{SPE}}}\ \textsc{WrapShifted\_N}$$

Figure 6.20: Applying a Splice Stack to a shifted expression

evaluating the meta expression into level i by $\cdot; \cdot \mathbin{@} \cdot \Downarrow_m \cdot$.

Shifting Shifting ($\Upsilon; \circlearrowleft^{\mathsf{i}}\ \mathsf{m} \mapsto \mathsf{m}'; \mathsf{SS}$, Figure 6.21), is an operation which takes a meta expression at level k and moves it i levels into the future to result in a meta expression at level $\mathsf{k} + \mathsf{i}$ along with any splices which need to be inserted into parent quote environments which arose as a result of the shifting.

There are three main points to consider when inspecting the definition of shifting. Shifting of some meta language constructs such as numbers or application is straightforward, because numbers and application can be introduced at any level given suitable level-correct arguments. Constructs which bind and use variables need to be more carefully considered. In order to shift a construct which binds a variable then any references to this variable must also be shifted. This is the purpose of the Υ. As a meta expression is traversed and shifted, the fresh variables which are created are added to the shift environment so that when a raw variable is encountered it can be replaced with its new binding. The Υ has a secondary purpose as well, if the meta syntax has a free meta variable in it, which may arise as computation proceeds, then the variable will not be in the Υ but still needs to be shifted. We can't change the binding site for the unbound metavariable but we know that it will be bound to a value of type **meta** m at level k in the program, this is because all metavariables end up being bound to variable of this type, as ensured by the definition of $\downarrow^{:} \cdot$. Therefore an unbound value is shifted i levels into the future by inserting a lift of i levels at level k. Splices are then inserted into each surrounding quotation in order to transport the value down to the correct stage where it will be used. The diagram in Figure 6.25 shows this in action. The expression to be shifted is represented abstracted by the boxed portion of the diagram. It contains the bound variable y and used variables x and y. Therefore y is rebound to the shifted y, as it is bound inside the same expression but x is shifted by additionally lifting it at the ambient level i. The rebound variable will end up in a splice environment and then can be used in the shifted expression.

$$\boxed{\Upsilon; \circlearrowleft^{\mathsf{k}}\ \mathsf{m} \mapsto \mathsf{m}'; \mathsf{SS}}$$

$$\frac{}{\Upsilon; \circlearrowleft^{\mathsf{k}}\ \mathbf{num}_m\ \mathsf{k} \mapsto \mathbf{num}_m\ \mathsf{k}; ([\,]_{\mathsf{k}})}\ \textsc{Shift\_Num}$$

$$\frac{(\mathsf{v}_m \to \mathsf{v}'_m, \Upsilon); \circlearrowleft^{\mathsf{k}}\ \mathsf{m} \mapsto \mathsf{m}'; \mathsf{SS}}{\Upsilon; \circlearrowleft^{\mathsf{k}}\ \lambda \mathsf{v}_m : \tau^m.\mathsf{m} \mapsto \lambda \mathsf{v}'_m : \tau^m.\mathsf{m}'; \mathsf{SS}}\ \textsc{Shift\_Lam}$$

$$\frac{\Upsilon; \circlearrowleft^{\mathsf{i}}\ \mathsf{m}_1 \mapsto \mathsf{m}'_1; \mathsf{SS}_1 \qquad \Upsilon; \circlearrowleft^{\mathsf{i}}\ \mathsf{m}_2 \mapsto \mathsf{m}'_2; \mathsf{SS}_2}{\Upsilon; \circlearrowleft^{\mathsf{k}}\ \mathsf{m}_1\ \mathsf{m}_2 \mapsto \mathsf{m}'_1\ \mathsf{m}'_2; (\mathsf{SS}_1 \cup \mathsf{SS}_2)}\ \textsc{Shift\_App}$$

$$\frac{\mathsf{v}_m \to \mathsf{v}'_m \in \Upsilon}{\Upsilon; \circlearrowleft^{\mathsf{k}} \Uparrow_{\mathsf{g}} \mathsf{v}_m \mapsto \Uparrow_{\mathsf{g}} \mathsf{v}'_m; ([\,]_{\mathsf{k}})}\ \textsc{Shift\_Lift\_In}$$

$$\frac{\mathsf{v}_m \notin \Upsilon \qquad \mathsf{MkSpliceStack}_{\mathsf{i}}(\Uparrow_{(\mathsf{g}+\mathsf{i})} \mathsf{v}_m) = (\mathsf{m}', \mathsf{SS})}{\Upsilon; \circlearrowleft^{\mathsf{i}} \Uparrow_{\mathsf{g}} \mathsf{v}_m \mapsto \mathsf{m}'; \mathsf{SS}}\ \textsc{Shift\_Lift\_Out}$$

$$\frac{\begin{array}{c}\Upsilon; \circlearrowleft^{\mathsf{i}}_{SPE}\ \mathsf{SPE} \mapsto \mathsf{SS}; \mathsf{SPE}'; \Upsilon' \qquad \Upsilon'; \circlearrowleft^{\mathsf{i}}\ \mathsf{m} \mapsto \mathsf{m}'; \mathsf{SS} \\ \mathsf{NewLevel}(\mathsf{SS}) = (\mathsf{SS}', \mathsf{SPE})\end{array}}{\Upsilon; \circlearrowleft^{\mathsf{i}}\ [\![\ \mathsf{m}\ ]\!]_{\mathsf{SPE}} \mapsto [\![\ \mathsf{m}'\ ]\!]_{(\mathsf{SPE}' \cup \mathsf{SPE})}; \mathsf{SS}'}\ \textsc{Shift\_Quote}$$

Figure 6.21: Shifting Meta Syntax

$$\boxed{\mathsf{NewLevel}(\mathsf{SS}) = (\mathsf{SS}', \mathsf{SPE})}$$

$$\frac{}{\mathsf{NewLevel}(\bullet) = (\bullet, \bullet)}\ \textsc{NewLevel\_Zero}$$

$$\frac{\mathsf{SplitSS}(\mathsf{SPE}, \mathsf{SS}) = (\mathsf{SPE}', \mathsf{SS}')}{\mathsf{NewLevel}(\mathsf{SPE} : \mathsf{SS}) = ((\bullet : \mathsf{SS}'), \mathsf{SPE}')}\ \textsc{NewLevel\_Succ}$$

Figure 6.22: Remove the last level from a splice stack

$$\boxed{\mathsf{SplitSS}(\mathsf{SPE}, \mathsf{SS}) = (\mathsf{SPE}', \mathsf{SS}')}$$

$$\frac{}{\mathsf{SplitSS}(\mathsf{SPE}, \bullet) = (\mathsf{SPE}, \bullet)}\ \textsc{SplitSS\_End}$$

$$\frac{\mathsf{SplitSS}(\mathsf{SPE}', \mathsf{SS}) = (\mathsf{SPE}'', \mathsf{SS}')}{\mathsf{SplitSS}(\mathsf{SPE}, (\mathsf{SPE}' : \mathsf{SS})) = (\mathsf{SPE}'', (\mathsf{SPE} : \mathsf{SS}'))}\ \textsc{SplitSS\_Next}$$

Figure 6.23: Retrieve the last splice environment in a splice stack

$$\boxed{\Upsilon; \circlearrowleft^{\mathsf{i}}_{SPE}\ \mathsf{SPE} \mapsto \mathsf{SS}; \mathsf{SPE}'; \Upsilon'}$$

$$\frac{}{\Upsilon; \circlearrowleft^{\mathsf{i}}_{SPE}\ \bullet \mapsto ([\,]_{\mathsf{i}}); \bullet; \bullet}\ \textsc{Shift\_Env\_Empty}$$

$$\frac{\Upsilon; \circlearrowleft^{\mathsf{i}}\ \mathsf{m} \mapsto \mathsf{m}'; \mathsf{SS} \qquad \Upsilon; \circlearrowleft^{\mathsf{i}}_{SPE}\ \mathsf{SPE} \mapsto \mathsf{SS}'; \mathsf{SPE}; \Upsilon' \qquad \text{fresh } \mathsf{v}_{m2} \qquad \mathsf{SPE}'' = (\mathsf{v}_{m2} : \tau^m \mapsto \mathsf{m}', \mathsf{SPE}') \qquad \Upsilon'' = (\mathsf{v}_{m1} \to \mathsf{v}_{m2}, \Upsilon')}{\Upsilon; \circlearrowleft^{\mathsf{i}}_{SPE}\ (\mathsf{v}_{m1} : \tau^m \mapsto \mathsf{m}, \mathsf{SPE}) \mapsto (\mathsf{SS} \cup \mathsf{SS}'); \mathsf{SPE}''; \Upsilon''}\ \textsc{Shift\_Env\_Cons}$$

Figure 6.24: Shifting A Splice Environment

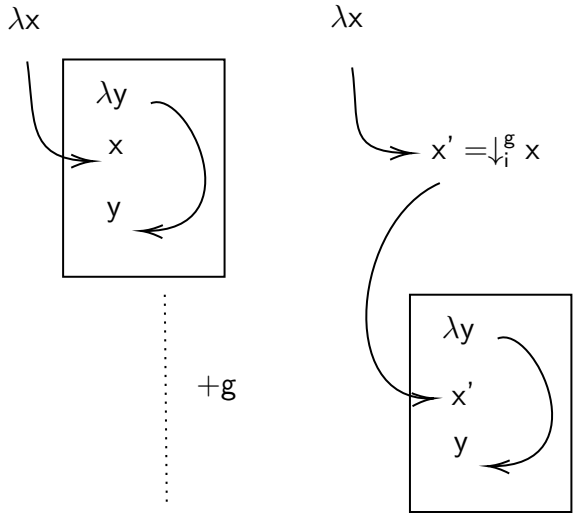

Figure 6.25: An example of $\Upsilon; \circlearrowleft^{\mathsf{i}}\ \mathsf{m} \mapsto \mathsf{m}'; \mathsf{SS}$

$$\boxed{\mathsf{MkSpliceStack_i(m)} = (\mathsf{m}', \mathsf{SS})}$$

$$\frac{}{\mathsf{MkSpliceStack_0(m)} = (\mathsf{m}, \bullet)}\ \textsc{SpliceStack\_Zero}$$

$$\frac{\text{fresh } \mathsf{v}_m \qquad \mathsf{MkSpliceStack_k}(\mathsf{v}_m) = (\mathsf{m}', \mathsf{SS})}{\mathsf{MkSpliceStack_{k+1}(m)} = (\mathsf{m}', (\mathsf{v}_m \mapsto \mathsf{m}, \bullet) : \mathsf{SS})}\ \textsc{SpliceStack\_N}$$

Figure 6.26: Making a Splice Stack

One further complication to this scheme is that if a variable bound at level $k+1$ is shifted i levels into the future, into level $k + i + 1$, then some manipulation of the returned splice stack is needed in order to bind the lowest splice environment to the newly shifted quotation. This is the purpose of $\mathsf{SplitSS}(\cdot, \cdot) = (\cdot, \cdot)$ and $\mathsf{NewLevel}(\cdot) = (\cdot, \cdot)$ which take a splice stack of size i, and removes the innermost splice environment in order to attach it to the quotation. Finally $\mathsf{NewLevel}(\cdot) = (\cdot, \cdot)$ extends the splice stack with an extra empty environment to make it the correct size before it is unioned with the result of also shifting the splices.

Each definition returns a splice stack which is a stack of k splice environments, one for each level. $[]_\mathsf{k}$ creates a stack of k empty splice environments. $\mathsf{MkSpliceStack_i(m)} = (\mathsf{v}_m, \mathsf{SS})$ creates a a splice stack which binds a specific expression (m) i levels prior and creates appropriate splices to transport it through subsequent levels. This definition is given in Figure 6.26. The $\mathsf{SS}_1 \cup \mathsf{SS}_2$ performs pairwise union of two splice stacks of size k.

Wrapping After the expression has been shifted, it is wrapped up with i quotations in order to correct the level of the expression from $\mathsf{k} + \mathsf{i}$ to k. Whilst it is being wrapped in i quotations the i levels of the returned splice stack are also unrolled and the appropriate splice environment is attached onto each quotation. Starting from a meta expression at level k, the result is a meta expression at level k but with type C i a. Now shifted and wrapped, when interpreted it will produce a CBPV computation at the desired level. This operation is specified in Figure 6.20.

Evaluating The $\Sigma; \mathsf{m} \mathbin{@} \mathsf{i} \Downarrow_m \mathsf{T}$ judgement (Figure 6.27) maps a meta expression m in environment Σ to a terminal value by interpreting a meta expression into a CBPV computation at level i and then appealing to $\cdot; \cdot \Downarrow \cdot$. The environment is extended in the course of the translation to map metavariables to CBPV computations. When the

$$\boxed{\Sigma; \mathsf{m} \mathbin{@} \mathsf{i} \Downarrow_m \mathsf{T}}$$

$$\frac{}{\Sigma; \mathbf{num}_m\ \mathsf{k} \mathbin{@} \mathsf{i} \Downarrow_m \mathbf{val}\ \mathbf{num}_v\ \mathsf{k}}\ \textsc{M\_Eval\_Num}$$

$$\frac{(\Sigma, \mathsf{v}_m \to \mathbf{return}\ \mathsf{v}); \lambda \mathsf{v} : (\mathsf{Meta}_\mathsf{n}\ \tau^m).(\downarrow_0^\mathsf{i}\ (\mathbf{meta}\ \mathsf{m})) \Downarrow \mathsf{t}}{\Sigma; \lambda \mathsf{v}_m : \tau^m.\mathsf{m} \mathbin{@} \mathsf{i} \Downarrow_m \mathsf{t}}\ \textsc{M\_Eval\_Lam}$$

$$\frac{\Sigma; ((\downarrow_0^\mathsf{i}\ (\mathbf{meta}\ \mathsf{m}_1))\ (\mathbf{meta}\ \mathsf{m}_2)) \Downarrow \mathsf{t}}{\Sigma; \mathsf{m}_1\ \mathsf{m}_2 \mathbin{@} \mathsf{i} \Downarrow_m \mathsf{t}}\ \textsc{M\_Eval\_App}$$

$$\frac{\mathsf{v}_m \to \mathsf{c} \in \Sigma \qquad \text{fresh } \mathsf{v}' \qquad \Sigma; \mathbf{let}\ \mathsf{v}' \leftarrow \mathsf{c}\ \mathbf{in}\ (\downarrow_\mathsf{g}^\mathsf{i}\ \mathsf{v}') \Downarrow \mathsf{t}}{\Sigma; \Uparrow_\mathsf{g} \mathsf{v}_m \mathbin{@} \mathsf{i} \Downarrow_m \mathsf{t}}\ \textsc{M\_Eval\_Lift}$$

$$\frac{\mathsf{mkLets}(\mathsf{n}, \mathsf{SPE}, \Sigma, (\mathbf{return}\ [\![\ \mathbf{return}\ (\mathbf{meta}\ \mathsf{m})\ ]\!])) = (\mathsf{c}, \Sigma') \qquad \Sigma'; \mathsf{c} \Downarrow \mathsf{t}}{\Sigma; [\![\ \mathsf{m}\ ]\!]_\mathsf{SPE} \mathbin{@} \mathsf{i} \Downarrow_m \mathsf{t}}\ \textsc{M\_Eval\_Quote}$$

Figure 6.27: Meta Evaluation

$$\boxed{\mathsf{mkLets}(\mathsf{n}, \mathsf{SPE}, \Sigma, \mathsf{c}_1) = (\mathsf{c}_2, \Sigma')}$$

$$\frac{}{\mathsf{mkLets}(\mathsf{n}, \bullet, \Sigma, \mathsf{c}) = (\mathsf{c}, \Sigma)}$$

$$\frac{\text{fresh } \mathsf{v} \qquad \mathsf{mkLets}(\mathsf{n}, \mathsf{SPE}, (\Sigma, \mathsf{v}_m \to !\mathsf{v}), (\mathbf{let}\ \mathsf{v} \leftarrow (\downarrow_0^\mathsf{n}\ (\mathbf{meta}\ \mathsf{m}))\ \mathbf{in}\ \mathsf{c}_1)) = (\mathsf{c}_2, \Sigma')}{\mathsf{mkLets}(\mathsf{n}, (\mathsf{v}_m \mapsto \mathsf{m}, \mathsf{SPE}), \Sigma, \mathsf{c}_1) = (\mathsf{c}_2, \Sigma')}$$

Figure 6.28: Definition of mkLets

translation encounters a variable, the environment is consulted to decide how to interpret the variable.

The translation is easiest to understand as similar to the translation of a CBN calculus into CBPV. The structure of both translations follows the same pattern and abstractions are interpreted in an analogous manner. Variables are bound to uninterpreted expressions.

The abstraction case (M_EVAL_LAM) demonstrates the extension of the environment with a mapping from the metavariable, which was bound by the meta lambda to the new variable which is now bound by the CBPV lambda. The computation which should be used to replace this newly bound variable is just **return** v in this case. The other case where binders are introduced during the interpretation is when a quotation is interpreted and mkLets will extend the environment when binding splice variables. In the case for splice variables, the computation is instead $!\cdot$, because the level of the variable needs to be corrected at the use site and the variable substituted for the contents of the quotation rather than the quotation itself.

Variables are interpreted by M_EVAL_LIFT, a 0 level lifting performs no lifting and is like using the variable normally. Otherwise, the value held by the variable is lifted the amount specified by $\Uparrow_{\mathsf{n}}^{\mathsf{g}} \cdot$. This is achieved by appealing to $\downarrow_{\mathsf{n}}^{\mathsf{g}} \cdot$ to interpret the argument into level n but shifted g levels forward.

The main work for interpreting quotations is to bind the splice environment by using mkLets as described previously. The body of the quotation is not translated itself until the quotation is used and it is known which level the body needs to be interpreted at is.

6.3.5 The $\uparrow \cdot$ Operation

In the computation language we included the $\uparrow \cdot$ operation but have neglected it so far in the discussion. The intention of $\uparrow \cdot$ is to implement lifting in the style found in existing multi-stage programming languages where values are lifted at runtime by matching on the runtime value. Using $\uparrow \cdot$ a standard multi-stage language can be translated into our calculus by interpreting lifting as a combination of $\uparrow \cdot$ and $\downarrow_1^0 \cdot$. In other words, first by converting the value into the meta syntax and then interpreting it one level in the future.

Not all values are liftable with $\uparrow \cdot$, Figure 6.17 gives the typing rules for $\mathsf{Lift}\ \tau^v$ which indicates whether a specific value is liftable. In this language only the Int type is liftable as no other values can be represented faithfully in the meta-syntax. Figure 6.29 gives the definition of $\mathsf{LiftComp}(\cdot) = \cdot$ which says how to evaluate the $\uparrow \cdot$ instruction. For

$$\boxed{\mathsf{LiftComp}(\mathsf{v}) = \mathsf{m}}$$

$$\frac{}{\mathsf{LiftComp}(\mathbf{num}_v\ \mathsf{k}) = \mathbf{num}_m\ \mathsf{k}}\ \textsc{Lift\_Comp\_Int}$$

Figure 6.29: Definition of $\mathsf{LiftComp}(\cdot) = \cdot$

integers the lifting translates a $\mathsf{Int}\ \mathsf{k}$ into an $\mathsf{Int}_m\ \mathsf{k}$. The Uninterpret function maps value types to meta syntax types.

6.3.6 Evaluation

The reduction semantics of the language are given by the $\cdot; \cdot \Downarrow \cdot$ judgement defined in Figure 6.30. The normal CBPV evaluation rules are unchanged and standard. The new rules are C_EVAL_INTERPRET, C_EVAL_FORCE C_EVAL_LIFT which describe how $\downarrow\!\!: \cdot$, $!\cdot$ and $\uparrow \cdot$ are evaluated. These three parts of the evaluation have been already described in the previous discussion.

As is standard in a CBPV calculus, values do not have an operational semantics which performs meaningful computation. The judgement $\cdot; \cdot \Downarrow_v \cdot$ performs "evaluation" of values, which amounts to just substituting variables for there definition in the environment.

6.3.7 Environments

During interpretation there are two environments. The first is a traditional environment which delays substitution by mapping CBPV variables to CBPV values. The second maps meta variables to CBPV computations which is used when translating meta syntax into CBPV syntax.

It is useful to take stock and think about all the different environments used during the compilation of a program.

- During compilation there is compilation environment (CE) which is used to remove splices from the source language.
- Computations have an environment (Σ) which maps CBPV variables to values and meta variables to CBPV computations.

$$\boxed{\Sigma; \mathsf{c} \Downarrow \mathsf{T}}$$

$$\frac{\Sigma; \mathsf{v} \Downarrow_v (\mathsf{v}_1, \mathsf{v}_2) \qquad (\Sigma, \mathsf{v}_1 \to \mathsf{v}_1, \mathsf{v}_2 \to \mathsf{v}_2); \mathsf{c} \Downarrow \mathsf{t}}{\Sigma; \textbf{split } \mathsf{v} \textbf{ by } \mathsf{v}_1.\mathsf{v}_2.\mathsf{c} \Downarrow \mathsf{t}} \text{ C\_EVAL\_SPLIT}$$

$$\frac{}{\Sigma; \textbf{case0 } \mathsf{v} \Downarrow ()} \text{ C\_EVAL\_CASE\_ZERO}$$

$$\frac{\Sigma; \mathsf{v} \Downarrow_v \mathsf{L}\ \mathsf{ivv} \qquad (\Sigma, \mathsf{v}_1 \to \mathsf{ivv}); \mathsf{c}_1 \Downarrow \mathsf{t1}}{\Sigma; \textbf{case } \mathsf{v} \textbf{ of } \mathsf{v}_1 \to \mathsf{c}_1; \mathsf{v}_2 \to \mathsf{c}_2 \Downarrow \mathsf{t1}} \text{ C\_EVAL\_CASE\_L}$$

$$\frac{\Sigma; \mathsf{v} \Downarrow_v (\mathsf{R}\ \mathsf{ivv}) \qquad (\Sigma, \mathsf{v}_2 \to \mathsf{ivv}); \mathsf{c}_2 \Downarrow \mathsf{t1}}{\Sigma; \textbf{case } \mathsf{v} \textbf{ of } \mathsf{v}_1 \to \mathsf{c}_1; \mathsf{v}_2 \to \mathsf{c}_2 \Downarrow \mathsf{t1}} \text{ C\_EVAL\_CASE\_R}$$

$$\frac{\Sigma; \mathsf{v} \Downarrow_v [\![\ \mathsf{c}\ ]\!]_{\Sigma'} \qquad \Sigma'; \mathsf{c} \Downarrow \mathsf{t}}{\Sigma; !\mathsf{v} \Downarrow \mathsf{t}} \text{ C\_EVAL\_FORCE}$$

$$\frac{\Sigma; \mathsf{v} \Downarrow_v \textbf{meta } \mathsf{m} \qquad \bullet; \circlearrowleft^{\mathsf{i}}\ \mathsf{m} \mapsto \mathsf{m}'; \mathsf{SS} \quad \mathsf{WrapShifted}(\mathsf{m}', \mathsf{SS}) = \mathsf{m}'' \qquad \Sigma; \mathsf{m}'' \ @\ \mathsf{i} \Downarrow_m \mathsf{t}}{\Sigma; \downarrow^{\mathsf{i}}_{\mathsf{g}} \mathsf{v} \Downarrow \mathsf{t}} \text{ C\_EVAL\_INTERPRET}$$

$$\frac{\Sigma; \mathsf{v} \Downarrow_v \mathsf{ivv} \qquad \mathsf{LiftComp}(\mathsf{ivv}) = \mathsf{m}}{\Sigma; \uparrow \mathsf{v} \Downarrow \textbf{val meta } \mathsf{m}} \text{ C\_EVAL\_LIFT}$$

$$\frac{\Sigma; \mathsf{c}_1 \Downarrow \textbf{val } \mathsf{ivv} \qquad (\Sigma, \mathsf{v} \to \mathsf{ivv}); \mathsf{c}_2 \Downarrow \mathsf{t}}{\Sigma; \textbf{let } \mathsf{v} \leftarrow \mathsf{c}_1 \textbf{ in } \mathsf{c}_2 \Downarrow \mathsf{t}} \text{ C\_EVAL\_LET}$$

$$\frac{\Sigma; \mathsf{v} \Downarrow_v \mathsf{ivv}}{\Sigma; \textbf{return } \mathsf{v} \Downarrow \textbf{val } \mathsf{ivv}} \text{ C\_EVAL\_RETURN} \qquad \frac{}{\Sigma; \lambda \mathsf{v} : \tau^v.\mathsf{c} \Downarrow \textbf{clos } \Sigma\ \mathsf{v}\ \mathsf{c}} \text{ C\_EVAL\_LAM}$$

$$\frac{\Sigma; \mathsf{v} \Downarrow_v \mathsf{ivv} \qquad \Sigma; \mathsf{c} \Downarrow \textbf{clos } \Sigma'\ \mathsf{v'}\ \mathsf{icb} \qquad (\Sigma \cup \Sigma', \mathsf{v'} \to \mathsf{ivv}); \mathsf{icb} \Downarrow \mathsf{t}}{\Sigma; \mathsf{c}\ \mathsf{v} \Downarrow \mathsf{t}} \text{ C\_EVAL\_APP}$$

$$\frac{}{\Sigma; \langle \mathsf{c}_1, \mathsf{c}_2 \rangle \Downarrow (\mathsf{c}_1 \times \mathsf{c}_2)} \text{ C\_EVAL\_PROD}$$

$$\frac{\Sigma; \mathsf{c} \Downarrow (\mathsf{c}_1 \times \mathsf{c}_2) \qquad \Sigma; \mathsf{c}_1 \Downarrow \mathsf{t}}{\Sigma; \iota_L\ \mathsf{c} \Downarrow \mathsf{t}} \text{ C\_EVAL\_PROJ\_L}$$

$$\frac{\Sigma; \mathsf{c} \Downarrow (\mathsf{c}_1 \times \mathsf{c}_2) \qquad \Sigma; \mathsf{c}_2 \Downarrow \mathsf{t}}{\Sigma; \iota_R\ \mathsf{c}_1 \Downarrow \mathsf{t}} \text{ C\_PROJ\_R}$$

Figure 6.30: Computation Evaluation

- Meta evaluation uses the same environment (Σ) as the computation evaluation and extends the mapping from meta variables to CBPV computations.
- Shifting has an environment (Υ) which maps meta variables to new meta variables.

6.3.8 A Well-Staged Evaluation

We want to ensure that the evaluation of the source program is well-staged. Informally, to be well-staged is that program fragments typed at level k are executed before fragments at level $k + 1$.

Keeping only a mono-levelled let construct was key in guiding the design of the calculus. The typing rule for let means that by evaluating a computation at level k, a variable at level k is bound in a computation at level k. Given the let construct is the only way in which to sequence effects, and the result of binding the value has to be used at the current stage, it makes us think very carefully about at what stage evaluation actually happens in. This is important in particular as a $[\![\cdot]\!].$ is interpreted at type $\mathsf{F}\ (\mathsf{U}\ (\mathsf{F}\ (\mathbf{meta}\ \tau^m)))$. In other words, an effectful computation which will produce the code for another effectful computation which produces a piece of meta syntax.

The $\mathbf{let}\ \cdot \leftarrow \cdot\ \mathbf{in}\ \cdot$ syntax makes sure the evaluation of the code generator happens at the correct level. Then the $!\cdot$ syntax is used inside a quotation to dictate where the result of performing the computation should be substituted into the quotation in order to act like a splice. In the meta language we went to a reasonable amount of effort with the compiler environments and during shifting to design a language with no splices in favour of splice environments. We were motivated to do this in order to simplify the compilation to the core language where all the splices need to be bound prior to the quotation anyway.

Therefore, in order to maintain a well-staged execution, the reduction semantics do not need to traverse into representation terms. Something which is of practical importance and makes stating the reduction semantics more straightforward.

6.3.9 Program Theory

There is no run operation in the language to avoid issues with scope extrusion. Instead we define a program theory which is a sequence of top-level splice definitions followed by an expression. Each definition is evaluated in turn and results in a value of type $\mathsf{U}\ (\mathsf{F}\ \mathsf{a})$ which is unwrapped and binds a variable of type a available in the remainder

$$\boxed{\mathbb{S} \vdash_S \mathsf{prog}_s : \tau^s}$$

$$\frac{\mathbb{S}; \bullet \vdash_0 \mathsf{s} : \tau^s \rightsquigarrow \mathsf{m}; \bullet}{\mathbb{S} \vdash_S \mathbf{main}\ \mathsf{s} : \tau^s \rightsquigarrow \mathbf{main}\ (\downarrow_0^0 (\mathbf{meta}\ \mathsf{m}))}\ \text{S\_PROG\_MAIN}$$

$$\frac{\mathbb{S}; (\bullet \vdash_0 \mathsf{s} : \mathsf{Code}_s \tau_1^s \rightsquigarrow \mathsf{m}; \bullet) \qquad (\mathbb{S}, \mathsf{v} : \tau_1^s) \vdash_S \mathsf{prog}_s : \tau_2^s \rightsquigarrow \mathsf{prog}_c'}{\mathbb{S} \vdash_S \mathsf{v} = \mathsf{s}; \mathsf{prog}_s : \tau_2^s \rightsquigarrow \mathsf{v} = (\downarrow_0^0 (\mathbf{meta}\ \mathsf{m})); \mathsf{prog}_c'}\ \text{S\_PROG\_DEF}$$

Figure 6.31: Typing a Source Program

$$\boxed{\mathbb{P} \vdash_C \mathsf{prog}_c : \tau^c}$$

$$\frac{\mathbb{P}; \bullet \vdash_0 \mathsf{c} : \tau^c}{\mathbb{P} \vdash_C \mathbf{main}\ \mathsf{c} : \tau^c}\ \text{C\_PROG\_MAIN}$$

$$\frac{\mathbb{P}; \bullet \vdash_0 \mathsf{c} : \mathsf{F}\ (\mathsf{U}\ (\mathsf{F}\ \tau_1^v)) \qquad (\mathbb{P}, \mathsf{v} : \tau_1^v) \vdash_C \mathsf{prog}_c : \tau^c}{\mathbb{P} \vdash_C \mathsf{v} = \mathsf{c}; \mathsf{prog}_c : \tau^c}\ \text{C\_PROG\_DEF}$$

Figure 6.32: Typing a Core Program

$$\boxed{\mathbb{P}_\Downarrow; \mathsf{prog}_c \Downarrow_p \mathsf{T}}$$

$$\frac{\mathbb{P}_\Downarrow; \bullet; \mathsf{c} \Downarrow \mathsf{T}}{\mathbb{P}_\Downarrow; \mathbf{main}\ \mathsf{c} \Downarrow_p \mathsf{T}}\ \text{C\_EVAL\_PROG\_MAIN}$$

$$\frac{\mathbb{P}_\Downarrow; \bullet; \mathsf{c} \Downarrow \mathbf{val}\ [\![\ \mathsf{c}\ ]\!]_\Sigma \qquad \mathbb{P}_\Downarrow; \Sigma; \mathsf{c} \Downarrow \mathbf{val}\ \mathsf{v} \qquad (\mathbb{P}_\Downarrow, \mathsf{v}_1 \mapsto \mathsf{v}); \mathsf{prog}_c \Downarrow_p \mathsf{T}}{\mathbb{P}_\Downarrow; \mathsf{v}_1 = \mathsf{c}; \mathsf{prog}_c \Downarrow_p \mathsf{T}}\ \text{C\_EVAL\_PROG\_DEF}$$

Figure 6.33: Evaluating a Core Program

of the program. The program theory is propagated implicitly through the judgements which type and evaluate expressions as it is not modified by any case and only consulted by the V_PROG_VAR rule which allows a variable from the program theory to be used at any level.

Splice definitions are also added to the source language and translated into core language splice definitions. There is likewise a source language program theory denoted by $\mathbb{S}$ which is passed implicitly when typing and compiling source language expressions.

6.3.10 Metatheory

In this section we state some of the metatheoretical properties which hold for the calculus.

The primary soundness theorem for the language states that for a given closed term then it evaluates to a value.

Theorem 1. *If* $\bullet \vdash_C \mathit{prog}_c : \tau^c$ *then there exists a* t *such that* $\bullet; \mathit{prog}_c \Downarrow_p t$.

Which is proved by induction by appealing to the main theorem which states soundness for the expression fragment of the language for each definition in the program.

Theorem 2. *If* $\bullet \vdash_n c : \tau^c$ *then there exists a* t *such that* $\bullet; c \Downarrow t$

The most interesting case of the induction is for $\downarrow^{\mathsf{g}}_{\mathsf{i}}$ ie where the typing rules states that if $\downarrow^{\mathsf{g}}_{\mathsf{i}} \ \mathsf{x}$ and $\Gamma \ \vdash_{(\mathsf{n}+\mathsf{i})} \ \mathsf{v} : \mathsf{Meta}_{\mathsf{n}} \ \tau^m$ then the resulting expression is typed at $\Gamma \vdash_{(\mathsf{n}+\mathsf{i})} \downarrow^{\mathsf{i}}_{\mathsf{g}} \mathsf{v} : \mathsf{Interpret\ n\ (C\ g}\ \tau^m)$. In the soundness proof we are required to show that the translation specified in Figure 6.27 respects this type. This is the only point where the levels of the meta syntax and normal syntax interact with each other. Due to how C_EVAL_INTERPRET is defined as the composition of shifting followed by evaluation, the necessary lemma is:

Theorem 3. *If* $\bullet \vdash_k m : \tau^m$ *then there exists a* t *such that* $\bullet; m \mathbin{@} k \Downarrow_m t$

These theorems have been mechanised by implementing a well-typed interpreter in Haskell. A weakness of the approach is that no facts about binders are verified by the mechanisation. It would be an improvement for the future to transfer the mechanisation to Coq or Agda where it would also be feasible to reason about binders.

It would also be desirable to prove some theorems about the compilation from the source language, in particular that the compilation preserves the level structure of the program.

These two directions are left to future work.

6.3.11 Examples

The calculus that we described in Section 6.3 is reasonably complicated and therefore would benefit from several examples describing how typing and evaluation proceeds. In particular, the examples should demonstrate how the calculus could be extended to a realistic language which allowed programs such as the ‹$› combinator from Section 6.1.

Simple Computation p1 is a simple abstraction which serve as a good example of how the basic compilation pipeline works.

$$\begin{aligned}&\mathsf{p1} :: \mathbb{S}\\&\mathsf{p1} = \textbf{main}\ \lambda \mathsf{v}.\mathsf{v}\end{aligned}$$

p1 is a source program and therefore typed and elaborated using rules found in Figure 6.31. The rule S_PROG_MAIN types and elaborates the main expression using rules described in Figure 6.9. The idea behind the elaboration is to first convert the source expression into a meta expression, removing all nested splices and implicit lifting before embedding the meta expression into the computation language using $\downarrow_0^0 \cdot$.

The expression p1 does not contain any splices or implicit cross-stage references so the translation is straightforward into the meta syntax. Supposing that p1 is typed with type $\mathsf{Int}_s \to \mathsf{Int}_s$ then the result of compilation is the core program:

$$\begin{aligned}&\mathsf{cp1} :: \mathbb{P}\\&\mathsf{cp1} = \textbf{main}\ (\downarrow_0^0 (\textbf{meta}\ (\lambda \mathsf{v}_m : \mathsf{Int}_m.\mathsf{v}_m)))\end{aligned}$$

The program is then evaluated by evaluating the definition of the **main** $\cdot$ program. As the definition solely consists of an application of $\downarrow_0^0 \cdot$ to a meta expression, the first step of evaluation is to translate the meta expression into CBPV syntax and then continue evaluation. The rules for interpreting meta syntax are given in Figure 6.27. In our case the M_EVAL_LAM rule interprets the lambda into a computation lambda. A fresh variable is created for the new computation lambda. The type of this variable is $\mathsf{Meta}_0\ \mathsf{Int}_m$, because remember that all variables are bound to uninterpreted meta expressions. The environment is extended with a mapping from the variable v_m to the fresh computation **return** v. The body is not immediately interpreted but the call to $\downarrow_0^0 \cdot$ is applied to the body of the lambda. In this case the body is now a meta expression with a free variable v_m. Later on when this is interpreted by the $\lambda\cdot$ rule the environment is consulted to find out what variable it should be mapped to. For now, it remains uninterpreted.

As the reduction does not continue underneath an abstraction, the body in this case will not be evaluated. This is observed in the C_EVAL_LAM rule which evaluates a lambda to a closure also stores the current environment. Therefore the result of compiling and then evaluating p1 is:

$$\begin{aligned}&\mathsf{res1} :: \mathsf{T}\\&\mathsf{res1} = \mathbf{clos}\ (\bullet, \mathsf{v}_m \to \mathbf{return}\ \mathsf{v})\ \mathsf{v}\ (\downarrow_0^0\ (\mathbf{meta}\ \mathsf{v}_m))\end{aligned}$$

Simple Implicit Lifting p2 is a source language program which includes a cross-level reference of the variable v. Therefore it can be used to highlight two parts of the calculus. Firstly, that during compilation the reference will be replaced with an entry in the splice environment for the surrounding quotation. Secondly, the argument will be lifted to the correct level during evaluation. In this case, the argument is the identity function from the first example.

$$\begin{aligned}&\mathsf{p2} :: \mathbb{S}\\&\mathsf{p2} = \mathbf{main}\ (\lambda \mathsf{v}.[\![\ \mathsf{v}\ ]\!])\ (\lambda \mathsf{v}.\mathsf{v})\end{aligned}$$

During elaboration the S_VAR rule will create a fresh variable v_{m2} and add an entry of $<$ Lift 0 v_{m2} $(\mathsf{Int}_m \to_m \mathsf{Int}_m)$ $(\Uparrow_1 \mathsf{v}_{m1}) >$ to the compile environment (assuming that v is mapped to the fresh variable v_{m1}). The compile environment is then split up by $\mathsf{SplitEnv}(\cdot) = (\cdot, \cdot)$ in the S_QUOTE clause. SPLIT_LIFT_ZERO will extract the definition added by the implicit lift into a splice environment, the compile environment is empty because there was just one implicit lift in the quotation which was only required to be lifted one level. Therefore the result of compiling the source program is as follows:

$$\begin{aligned}\mathsf{vm}\ &= \mathbf{meta}\ (((\lambda \mathsf{v}_{m1} : \mathsf{Int}_m \to_m \mathsf{Int}_m.[\![\ \mathsf{v}_{m2}\ ]\!]_{(\mathsf{v}_{m2} : \mathsf{Int}_m \mapsto (\Uparrow_1 \mathsf{v}_{m1}), \bullet)})\ (\lambda \mathsf{v}_m : \mathsf{Int}_m.\mathsf{v}_m)))\\\mathsf{cp2} &= \mathbf{main}(\downarrow_0^0 \mathsf{vm})\end{aligned}$$

Once compiled, it is time to evaluate. When an application is forced, the function is interpreted but the argument remains unevaluated. Ultimately we end up in the C_EVAL_APP case which first evaluates the function to a closure and then evaluates the body with the functions argument extending the environment. Therefore in this case the body of the lambda is evaluated in the environment:

$$\Sigma = \bullet, \mathsf{v}_{m1} \to \mathbf{return}\ \mathsf{v}, \mathsf{v} \to \mathbf{meta}\ (\lambda \mathsf{v}_m : \mathsf{Int}_m.\mathsf{v}_m)$$

In the cp2 example the body of the lambda expression is an $[\![\ \cdot\ ]\!]$. and therefore interpreted by the M_EVAL_QUOTE rule. The rule does not evaluate the body of the

quotation but does evaluate each of the splices placed in the splice environment. If there wasn't a splice environment for a well-staged interpretation the evaluation would have to traverse inside the quotation. Each entry in the splice environment is translated into a **let** $\cdot \leftarrow \cdot$ **in** $\cdot$. In environment Σ, $(\mathsf{v}_{m2} : (\Uparrow_1 \mathsf{v}_{m1}) \mapsto \cdot, \bullet)$ is interpreted into:

$$\_ = \textbf{let}\ \mathsf{v}_2 \leftarrow (\downarrow_0^0 (\textbf{meta}\ (\Uparrow_1 \mathsf{v}_{m1}))) \textbf{ in } (\textbf{return}\ (\llbracket\ \textbf{return}\ (\textbf{meta}\ \mathsf{v}_{m2})\ \rrbracket))$$

Now the interpreted expression has introduced a new variable v_2 which is a replacement for v_{m2}. Therefore the substitution in the environment is extended once again to map v_{m2} to the correct computation. In this case because the binding was introduced by a splice environment the extension uses $!\cdot$:

$$\Sigma_1 = \Sigma, \mathsf{v}_{m2} \to\ !\mathsf{v}_2$$

When it is necessary to evaluate the contents of the quotation, the environment will replace the metavariable with $!\mathsf{v}_2$ and the typing rules for $!\cdot$ dictate that the argument must evaluate to something of type $\mathsf{U}\ (\mathsf{F}\ \mathsf{a})$. This is achieved during the evaluation of **let** $\mathsf{v}_2 \leftarrow (\downarrow_0^0 (\textbf{meta}\ (\Uparrow_1 \mathsf{v}_{m1})))$ **in** $\cdot$. Let's look into that now.

When $\Uparrow_1\ \mathsf{v}_{m1}$ is interpreted the metavariable v_{m1} is substituted for the value in the environment and then interpreted at one level higher than the current level. Therefore we evaluate $(\lambda \mathsf{v}_m : \mathsf{Int}_m.\mathsf{v}_m)$ at base level 0 with offset 1 due to the lift. As a result, the type of the interpreted expression will be $\mathsf{F}\ (\mathsf{U}\ (\mathsf{F}\ (\mathsf{Meta}_1\ (\mathsf{Int}_m \to_m \mathsf{Int}_m))))$ as could be discerned from the typing rule M_LIFT and the $\mathsf{Interpret}$ function which computes a τ^c from an τ^m.

The transformation is achieved by first shifting the meta expression one level forward. As the expression is closed the splice stack will be empty and so $\mathsf{WrapShifted}(\cdot,\cdot) = \cdot$ just adds a $\llbracket\ \cdot\ \rrbracket.$ constructor around the shifted expression. Then the quotation is evaluated in the normal manner, this time the splice environment is empty. The body of the quotation is **meta** $(\lambda \mathsf{v}_m : \mathsf{Int}_m.\mathsf{v}_m)$.

Therefore the result of evaluating cp2 is finally:

$$\begin{aligned}
&\Sigma_2 = \Sigma_1, \mathsf{v}_2 \to \textbf{meta}\ (\lambda \mathsf{v}_m : \mathsf{Int}_m.\mathsf{v}_m)\\
&\mathsf{res2} :: \mathsf{T}\\
&\mathsf{res2} = (\textbf{val}\ \llbracket\ (\textbf{return}\ (\textbf{meta}\ \mathsf{v}_{m2}))\ \rrbracket_{\Sigma_2})
\end{aligned}$$

This example demonstrates how function values can be lifted by delaying the evaluation of the syntactic representation of function until the last possible moment. Using these ideas it would be possible to implement the ‹$› combinator from Section 6.1.

Iterated Lifting It might have seemed strange in the previous example that the function argument was still unevaluated by the end of computation. Why did we not interpret it into a CBPV function at some point? The answer is that it may not yet have reached its final level. It could still be subject to further lifting inside the quotation. p3 demonstrates this concept in action. The variable v' is used itself in a quotation, therefore the argument to the function will need to be lifted twice before it is at the correct level.

$$\mathsf{e}_3 = (\lambda \mathsf{v}.[\![\ (\lambda \mathsf{v}'.[\![\ \mathsf{v}'\ ]\!])\ \mathsf{v}\ ]\!])\ (\lambda \mathsf{v}.\mathsf{v})$$

It is not instructive to run through this example in a lot of detail because evaluation will not proceed inside the quotation. In order to observe how computation will proceed inside the quotation it will be necessary to use a splice definition in the top-level program theory. The point of the example is to demonstrate a case where delaying evaluation is necessary until the precise moment where the definition will be needed to be used to continue with evaluation.

Using $\uparrow\ \cdot$ e4 is an example of a CBPV program which uses the $\uparrow\ \cdot$ syntax in order to convert a value into meta syntax before it is interpreted by $\downarrow_0^0\ \cdot$. This composition is equivalent to the identity function.

$$\mathsf{e4} :: \mathsf{C}$$
$$\mathsf{e4} = \textbf{let}\ \mathsf{v} \leftarrow (\uparrow\ (\textbf{num}_v\ 0))\ \textbf{in}\ (\downarrow_0^0\ \mathsf{v})$$

First the let expression is evaluated which converts the value number into a meta number.

$$\_ = \bullet; \uparrow\ (\textbf{num}_v\ 0) \Downarrow \textbf{val meta}\ (\textbf{num}_m\ 0)$$

Then the body of the let expression is evaluated in an environment extended with the let-bound variable v.

$$\Sigma = \bullet, \mathsf{v} \to \textbf{meta}\ (\textbf{num}_m\ 0)$$
$$\_ = \Sigma; \downarrow_0^0\ \mathsf{v} \Downarrow \mathsf{t}$$

As $\mathsf{g} = 0$, shifting and $\mathsf{WrapShifted}(\cdot,\cdot) = \cdot$ will not modify the expression so the result (t) is computed by:

$$\_ = \Sigma; \textbf{num}_m\ 0\ @\ 0 \Downarrow_m \textbf{val num}_v\ 0$$

Therefore the composition of $\uparrow\ \cdot$ and $\downarrow_0^0\ \cdot$ is just the identity as expected.

If instead of $\downarrow_0^0 \cdot$ we compute $\downarrow_1^0 \cdot$ then the result will be a representation of a level 1 value at level 0.

$$\begin{aligned}&\mathsf{e5} :: \mathsf{C}\\&\mathsf{e5} = \textbf{let } \mathsf{v} \leftarrow (\uparrow (\textbf{num}_v\ 0)) \textbf{ in } (\downarrow_1^0 \mathsf{v})\end{aligned}$$

Computation of e5 proceeds as before until $\downarrow_\cdot^\cdot \cdot$ is computed. This time shifting and wrapping will modify the $\textbf{num}_m\ 0$ expression. Shifting of $\textbf{num}_m\ 0$ in this case will just result in $\textbf{num}_m\ 0$ but then $\mathsf{WrapShifted}(\cdot,\cdot) = \cdot$ will wrap a quotation around the meta value. The result of evaluating $[\![\ \textbf{num}_m\ 0\ ]\!]_\bullet$ is itself a quotation where the body is unevaluated.

$$\_ = \Sigma; [\![\ \textbf{num}_m\ 0\ ]\!]_\bullet\ @\ 0 \Downarrow_m \textbf{val}\ [\![\ (\textbf{return}\ (\textbf{meta}\ (\textbf{num}_m\ 0)))\ ]\!]_\Sigma$$

This is similar to the effect of the traditional lift operation.

Therefore we have seen how $\uparrow \cdot$ and $\downarrow_\cdot^\cdot \cdot$ can be used together to implement a traditional lift operation in the language. Exploring this direction further would be interesting future work.

6.4 Other Considerations

The main motivation for the core language was to provide a suitable language for compiling a source language which could implicitly lift variables representing functions. It was established in the introduction that being able to lift a wider range of types would improve the API design of staged libraries. The core language specified can now support the lifting of functions but is more complicated than necessary. It would have been possible to target a simpler core language rather than a variant of Levy's CBPV in order to achieve the same effect. We have not yet taken advantage of the ability to embed different evaluation strategies or the ability to add effectful operations to the language. This section will describe some possible future extensions to the language which take advantage of the additional features of CBPV.

The CBV Translation The most immediate difference which could be considered is using a translation into the core language which was similar to the CBV embedding of lambda calculus into CBPV. Using the CBV translation would yield a language closer to a traditional multi-stage language. In particular, there would be no need for the runtime $\downarrow_\cdot^\cdot \cdot$ nor **meta** $\cdot$ value type. The translation would instead interpret lifting using the $\uparrow \cdot$

operator and source expressions would be compiled to CBPV expressions rather than the indirection via **meta** $\cdot$ syntax.

Effects There has been a little study into the combination of effects and multi-stage languages but most formally specified multi-stage languages are not combined with effects. The interaction of levels and effects is interesting because the separation into levels means that also it is necessary to distinguish which effects are necessary for use during program generation and which are necessary for program execution. It would be interesting to explore further how this language could be modified to add effectful combinators. At the least the big-step semantics would have to be modified to evaluate to a stack-based machine or a monadic semantics so that the evaluation order was explicit. At the moment it doesn't matter as there are no effects in the language.

This distinction was touched upon in Section 3.7.1 where we talked about refactoring a program in the monad transformer style into into a staged interpreter which distinguished between some effects which were needed at compilation time and some at runtime.

It may also be interesting to explore how languages which implement effect handlers by extending CBPV (for example (Plotkin and Pretnar, 2009; Forster et al., 2019; Kammar and Plotkin, 2012; Convent et al., 2020)) can be extended to multi-stage languages by adding a level index. The core idea of adding a level index to the typing judgement can be readily applied to many languages without compromising other meta-theoretical properties.

In the practical setting, Lightweight Modular Staging (LMS) (Rompf, 2012) and Template Haskell have always accounted for effects during the translation to the internal representation. LMS is an extension which implements support for runtime code generation to Scala. For LMS accounting for effectul operations is especially important as Scala is an effectful language so the translation makes sure that the effects in the generated program happen in the same order as the source program. This is ensured by a conversion to ANF, which is similar to the style needed in a CBPV language which F values must be bound using the **let** $\cdot \leftarrow \cdot$ **in** $\cdot$ construct.

In short, the decision to start from a variant of CBPV makes a wide array of literature about effect handlers directly accessible so it seems a plausible future direction to use this language as a basis for exploring effectful generation of effectful programs.

Relevance to Stage Polymorphism Our calculus can also be seen as a faithful formalisation of a language with full support for stage-polymorphism. Stage-polymorphism is the name given to a concept where a term in a language can be interpreted in different ways or stages and is usually implemented by a combination of interfaces and operator overloading (Rompf and Odersky, 2010). LMS is the language which is most well-known for implementing stage-polymorphism but it is also possible to embed a stage-polymorphic language in Haskell using type classes.

As our language contains an explicit term representation in the form of **meta** $\cdot$ syntax values, stage polymorphism is implemented by giving different interpretation functions for this explicit representation. In the language at the moment, there is just one interpretation function which can interpret the syntax into any stage but it would be possible to add other interpretations. For example, an interpretation which interprets the syntax as a string or an interpretation which partially evaluates the syntax under some equational theory.

Therefore it is possible that our language could be used as a more faithful formalism to LMS than the one presented by Rompf (2016) which does not really consider effects or stage-polymorphism in any substantial detail.

Other Extensions to the Source Language There are many constructs in the core language which are not used during the compilation from the source syntax. For example, the sum and product constructors are not used at any point during compilation as the source language has neither sums or products. If there was a more complicated source language then these features could be used. Depending on the source language it may also be desirable to extend the meta syntax. At the moment the meta syntax precisely reflects the source syntax which makes the compilation from the source syntax to meta syntax trivial.

What is an Unevaluated Expression? Unevaluated expressions are represented syntactically using the **meta** $\cdot$ embedding. In a normal CBPV calculus unevaluated expressions are stored as frozen CBPV expressions. There is certainly some choice about what representation can be used for unevaluated expressions.

The meta-syntax representation was chosen here because the structure of the term needed to be evaluated in different ways during evaluation. In particular, the syntax is interpreted into different stages depending on which stage the variable ends up being used at. The cost of this was needing to perform interpretation at runtime in order to convert the syntactic representation into a semantic object of the correct stage. There

was little choice but to perform this interpretation as in general it is not known statically which stage the variable would be used at. If there is a restriction to k levels then this interpretation can instead be performed eagerly as the expression is constructed in a manner similar to the WQ data type from Section 6.1.

In a two-level calculus, then the representation of unevaluated terms could be a pair of a current and future stage value. At stage 0, whilst constructing stage 1, the unevaluated expression representation would be a pair of a stage 0 and stage 1 value. Then if the stage 0 variable is used at stage 1, the representation is projected from the pair and copied to the next stage. In the two-level variant, not all values will be liftable, in particular, quotations are not liftable as there is suitable representation for a quotation at level 1 in a two-level language.

Knowing statically that there is a fixed upper bound to the number of levels means that the interpreter itself can be specialised to work with a specific number of levels. This means that rather than performing runtime interpretation of syntax, some computation can be moved into the logic of the evaluator. This seems like an interesting question of partial evaluation in its own right.

Related to this question would be whether it is possible to design a static analysis of the program could be computed and the maximum possible level that each expression could ever be lifted to and specialise the interpreter to a variant which constructs these k-tuples directly in the style sketched above. For example, if it is determined by analysis that a certain fragment of the program only ever used values at two-levels the 2-tuple variant could be used for this fragment rather than having to continually reinterpret the meta syntax.

This kind of whole program analysis would be of practical importance because if it is established that a program only uses one stage then it would be possible to avoid the cost of the generality of n stages. If a user does not use the power of multi-stage features they shouldn't have to pay the cost. Of course, this kind of analysis would be a *whole-program* analysis and so the decisions could only be taken when the program as a whole was known. In particular, consider the identity function once again, the function defined in a library would have to understand how to deal with any k-tuple or meta syntax variant of the runtime argument and specialisations created for each of these call patterns.

Partial Syntactic Embedding In the formalism, the whole source language is left uninterpreted until it is projected into the correct stage. The idea of leaving a syntactic representation of a fragment of the language may be useful in a practical setting. For instance, storing the whole type class derivation proof as a syntactic representation

would allow a form of cross-stage persistence for type-class evidence. This would provide an alternative to CodeC constraints which take the alternative approach of propagating the level a constraint is used to the source program and ensure elaboration immediately provides evidence at the correct level.

Implicit Lifting? Good or bad? In our source language, we allow variables to be used at future stages implicitly. This mirrors what most languages with implicit cross-stage persistence implement but our experience has showed us that implicit lifting of this kind can cause code size issues during code generation. In particular there are two issues with implicit lifting. Firstly, it may be a programming mistake to use the variable at the future stage, you may accidentally use a function at the incorrect stage and the result is implicitly lifted and the function called at runtime rather than compile time. There is a significant difference between $[\![\ \mathsf{f}\ldots\ ]\!]$ and $[\![\ \$(\mathsf{lift}\ \$\ \mathsf{f}\,..)\ ]\!]$, the former will delay the evaluation of f... to the program's runtime while $\$(\mathsf{lift}\ \$\ \mathsf{f}\,..)$ will evaluate the call to f at compile time and then only persist the result of the call to runtime.

In particular during refactoring of large projects into a staged style you often have to ask yourself at what stage a particular definition should be evaluated in. If it is during the compile-time stage then the definition has to be itself staged to take advantage of variables that will be known at compile time. However, with implicit lifting you may miss this opportunity as a now statically known value will be lifted implicitly rather than a type error telling you that the variable is now used across stages. It is a missed opportunity. Implicit coercion is frowned upon in most statically typed languages and perhaps too so would implicit lifting if multi-stage features were more commonly used.

The second issue with implicit lifting, as implemented, is that it can cause a lot of code bloat in the generated code. In our suggested semantics, no evaluation or normalisation happens at all to meta terms, all arguments are completely unevaluated and therefore when lifted to a future stage the whole expression from the prior stage will be copied into place. If you are lifting the same variable multiple times then the same definition can end up repeated in your generated code in multiple places. Practically compilers sometimes have difficulty dealing with a large amount of generated code, especially as typically the generated program will initially be one very large expression.

Could we instead let bound lifted variables which are used more than once? Perhaps, but an issue here is that we have allowed lifting or interpretation in this case to be an effectful operation. Therefore unless the effects of lifting are not observable it is unsound to deduplicate implicitly lifted variables. In Template Haskell this is true as well as the type of lifting is $\mathsf{Quote\ m} \Rightarrow \mathsf{a} \rightarrow \mathsf{m\ (TExp\ a)}$ rather than $\mathsf{a} \rightarrow \mathsf{TExp\ a}$.

Therefore in designing a realistic source language it could be better to remove implicit lifting and force users to explicit lift values from one stage to another.

6.5 Related Work

The most relevant related work is by Kakutani et al. (2019) who have also extended a CBPV calculus with levelled judgements whilst exploring the connection between dual context and Fitch-style modal calculi. They use the language in order to understand the relationship between the two, whereas our focus is from the practical perspective.

6.5.1 Cross-Stage Persistence and MSP

Most languages have taken the approach of implementing cross-stage persistence by lifting as originally suggested by Moggi. For example, both MetaOCaml (Kiselyov, 2014), and Dotty (Stucki et al., 2018) both implement persistance by lifting and therefore have similar restrictions about which values can be lifted to Typed Template Haskell.

The approach by Sampson et al. (2017) is interesting because the cross-stage persistence operates between multiple heterogeneous stages and makes particular use of special language features from future states to persist values.

Some other formalisms tackle cross-stage persistence but also not from a practical perspective and fail to explain how to implement the proposal. In particular Hanada and Igarashi (2014) describes the $\%$ operator which implements CSP by embedding a runtime value into the quotation. In a language with a separate compile-time phase, freezing a runtime value and it's representation on the heap is usually not possible and a serialisation based approach is necessary.

One possible way to implement the $\%$ operator in Haskell would be to use a serialisation mechanism which can serialise the runtime representation of a value. In particular, either using a compact region (Yang et al., 2015) or the proposed serialisation mechanism by Berthold (2010). Currently neither approach is feasible because function values are not compactable and Berthold's approach is not implemented in the compiler.

The advantage of the design for cross-stage persistence in our calculus is that the implementation strategy is clear even if evaluation would be quite inefficient.

6.5.2 Other Adjoint Calculi

There are other adjoint calculi which may also work as a good foundation for multi-stage languages. The language described by Downen and Ariola (2019) looks like an interesting one to consider as it is possible to express a call by need evaluation strategy with the addition of 4 shifting constructs.

Others have investigated designing a language where the CBN and CBV translations into CBPV can be decided upon at runtime (Dunfield, 2015). The conclusion there was that the two translations are not very compositional which led to a different adjoint calculus to be designed. In our setting, we conjecture that as long as the interpretation is consistent by level then it would be possible to sometimes interpret meta syntax into CBV and sometimes CBN.

Chapter 7

Related Work

In this chapter we discuss a selection of related work and how it is relevant to this thesis. Primarily we are interested in other practical implementations of multi-stage languages but we will also briefly discuss some other formalisms and how they compare to our formalism of Typed Template Haskell.

7.1 MetaOCaml

The history of multi-stage programming research is primarily focused on extending ML-like languages with multi-stage features. In particular, the first multi-stage language was MetaML (Taha and Sheard, 1997, 2000) which introduced the idea of using quotations and splices in the typed style. It is also notable for introducing cross-stage persistence by lifting which has largely persisted to modern implementations.

The successor of MetaML is MetaOCaml which is implemented as an extension to the main OCaml compiler. The project was initiated by Walid Taha and the majority of the implementation completed by Calcagno et al. (2003). Later on, Oleg Kiselyov reimplemented MetaOCaml from scratch in what is now known as BER MetaOCaml (Kiselyov, 2014).

BER MetaOCaml is maintained as a series of patches against the latest release of OCaml. In the past this has made it quite difficult to install but recently Opam has been extended to allow the direct installation as in common with other OCaml compilers.

BER MetaOCaml has support for runtime code generation. In order to support "compile-time" code generation the generated code can be printed and saved into a file which will be separately compiled. The language implements the standard cross-stage

persistence by lifting.

The BER MetaOCaml implementation uses the combinator approach used by Template Haskell in order to construct the code representation. OCaml as a language has far less support for implicit arguments and still a greater emphasis on type inference for expressions. Therefore constructing an untyped representation using combinators has not affected soundness. It seems possible as the language continues to develop that soundness issues will become evident.

A way that MetaOCaml has been unsound is by the unsound treatment of evidence introduced by GADTs. This hole was plugged drastically by completely rejecting pattern matching on a GADT constructor inside quotations. A more natural approach would be to consider the levels of the evidence as we do in our core calculus.

As well as the combinator style implementation, an implementation based on the intermediate form was proposed by Roubinchtein (2015). The idea of the implementation is to use the serialised intermediate representation in order to represent quotations. Unlike us, the motivation was not for soundness but for speed. Their implementation follows a similar trajectory to ours, splice points are used in order to delay substitution into an opaque representation. It's not clear how well the implementation performed relative to the combinator based approach because the evaluation doesn't benchmark aspects such as compilation speed or the speed of computing a program. Their implementation is simplified because the intermediate representation doesn't contain type information.

7.2 Dotty

In more recent times the new Scala 3 compiler, Dotty, has also added support for multi-stage features (Stucki et al., 2018). There is support for both runtime and simple compile-time code generation.

Dotty is quite interesting to compare to Typed Template Haskell as the Scala source language is complicated in a similar way to Haskell. There is also a complicated elaboration phase which resolves implicit and type arguments. The internal representation of quotations is a typed syntax tree (TASTY (Odersky et al., 2016)) in the same way as our proposed implementation.

Through experimentation, it has been observed that cross-stage references to implicit evidence are forbidden in Dotty. Others have reported soundness issues to do with

using GADTs which introduce constraints into the environment[1]. Although we did not formally consider GADTs in our core calculus, the idea of adding a level to a constraint extends naturally to constraints locally introduced by a GADT.

On the other hand, they also recognise the same issues to do with cross-stage references to types and solve the issue in a similar way to the LiftT proposal. In Dotty, types can be explicitly quoted by `'[]` , quoting a type `T` yields a value of type `Type[T]`. These type representations can be spliced into positions which require a type in an expression.

As well as being able to explicitly quote and splice types, the compiler will attempt to correct any cross-stage reference to a type by using an implicit argument. For example if `T` is bound at level 0 and used in a quotation at level 1 then the type will be rewritten to `${ summon[Type[T]] }` which is similar to our proposal for LiftT.

Lifting values is supported in the standard cross-stage persistence by serialisation. It is notable that Dotty does not implicitly lift variables used across stages and that the user must explicitly insert a lift themselves. This decision lines up with my intuition that implicit lifting is sometimes undesirable.

The support in Dotty for compile-time evaluation is limited compared to Typed Template Haskell as a full Scala interpreter is not available during compilation. Therefore the primary use of Dotty metaprogramming is for runtime code generation which can still be performant on the JVM due to the advanced JIT compilers. Benchmarks of multi-stage programs using Dotty or LMS are usually running a large number of iterations so the JIT can warm-up and the cost of the initial code generation becomes relatively smaller.

7.3 Multi-Stage Programming and Haskell

7.3.1 MetaHaskell

MetaHaskell (Mainland, 2012) was a proposal for extending Template Haskell to support heterogenous metaprogramming. Our concern has solely been homogeneous metaprogramming so MetaHaskell is not directly relevant. There is sometimes some confusion that MetaHaskell is related to Typed Template Haskell because Geoffrey Mainland also implemented Typed Template Haskell and due to the similarity with the name MetaOCaml.

[1] `https://github.com/lampepfl/dotty/issues/9353`

7.3.2 Using Typed Template Haskell

There have only been a few examples of papers and libraries which have used Typed Template Haskell in order to write staged programs. We distinguish here between "trivial" uses of Typed Template Haskell where a typed quote is used to provide a nicer API for a value constructed using the untyped combinators and a properly staged library where the majority of code is constructed in the staged style with quotes and splices.

Most recently there has been a renaissance of staging using Typed Template Haskell partly due to my increased interest in the subject and the staging community becoming more aware of Typed Template Haskell existing. In particular the work of Pickering et al. (2020) has already been discussed as a novel staging example using Typed Template Haskell. Willis et al. (2020) implements a staged parser combinator library with very competitive performance compared to parser generators. Yallop et al. (2018) uses advanced language features such as data families to implement a generic interface for partially static data. This paper is a great example of how an advanced type system can help write complicated code generators.

7.3.3 Generating Core Expressions

Winant et al. (2017) proposed an alternative well-typed code-generation technique for generating core programs. Their approach was to embed a representation of core terms into a dependently typed language (Agda) and then write a program which resulted in a core expression which could be inserted into Haskell program. The rich type system of Agda allowed a very precise specification about the type of the generated program. Their main example is of generating lenses, which they claim impossible in Typed Template Haskell, where the resulting type of the lens depends on a value level description of the structure of the data type. This style of heterogeneous meta-programming between two high-level languages with advanced type systems would be interesting to explore further.

7.4 Cloud Haskell

It is useful to compare the treatment of typed quotations to a different metaprogramming facility implemented in Haskell: Cloud Haskell (Epstein et al., 2011). Both approach the issues of persistence and polymorphism in a slightly different manner but are implemented independently.

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

```
static 1 :: StaticPtr Int
```

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

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

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

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

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

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

The internal representation used for static forms doesn't contain type information so type variables don't need to be persisted in the same manner as for the new suggested representation for Template Haskell quotations. When a StaticPtr is dereferenced it is the responsibility of the user to ensure that is happens at the correct type.

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

Static forms are dereferenced at runtime, therefore the optimiser can't specialise a program based on a specific static form. Compared to programs constructed using compile-time metaprogramming this can lead to worse performance.

Serialising Programs The Cloud Haskell interface is intended to serialise closed expressions but is not compositional; there is no way in which to combine two static pointers together. For this reason, the common way to interact with static pointers is to define a simple DSL for building larger programs which contain static pointers. The most elegant example is in the STATIC-CLOSURE[2] library. Static forms are used to construct the leaves by implementing cross-stage persistence for top-level functions and closed expressions.

```
data Closure t a where
  CPure :: !(Closure t (t → a)) → t → a → Closure t a
  CStaticPtr :: !(StaticPtr a) → Closure t a
  CAp :: !(Closure t (a → b)) → !(Closure t a) → Closure t b
```

A Closure t a represents an expression of type a that we can serialise to a value of type t. It is either a StaticPtr, a pure value a which we can serialise into value of type t and also deserialised by the client or an application of two closures. Closures can be serialised, deserialised and dereferenced.

```
putClosure :: Binary t ⇒ Closure t a → Put
getClosure :: Binary t ⇒ Get (Closure t a)
unclosure :: Closure t a → a
```

Instead of using quotation and splicing to construct larger terms expressions are constructed using this DSL.

Elaboration Evidence Functions which use type class constraints and implicit parameters are forbidden from appearing in static forms. This is commonly solved by

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

defining wrapper data types which remove the constraint.

```
data Dict c = c ⇒ Dict

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

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

The Dict data type is used to store the constraint. When it is pattern matched on the constraint is available in the definition. showInt no longer contains a type class constraint and so can appear as the argument to the static keyword.

7.5 Other Modal Type Systems

Recently modal type systems have also been proposed for functional reactive programming (FRP) languages which guarantee certain properties such as not leaking space between time intervals by keeping reference to previously computed streams. Of particular note is Rattus (Bahr, 2020) which is implemented by embedding the language into Haskell and uses a combination of a core and constraint solver plugins in order to augment the type system with extra typing rules for the modality operations.

This perhaps points to the possibility of using a combination of plugins for experimenting in future with modal type systems in Haskell. There are a lot of similarities between the typing rules to Typed Template Haskell, Rattus and Cloud Haskell but none of the code is shared between their implementations. It would be interesting to work out how to exploit these similarities in future.

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

In recent times, Fitch-style Modal calculi (Clouston, 2018; Gratzer et al., 2019) have become a popular way of specifying a modal type system due to their good computational properties. It would be interesting future work to attempt to modify our

core calculus to a Fitch-style system which was not level indexed. In particular the calculus for Simple RaTT (Bahr et al., 2019) looks like a good starting point.

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

7.6 Other Multi-Stage Formalisms

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

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

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

Chapter 8

Conclusion

At the start of this thesis we set out with the goal of investigating ways of designing high-level libraries with guarantees about optimisation. Attention was swiftly turned to multi-stage programming techniques as a way to write compilers in the form of libraries, in a style familiar to normal programmers. It was also fortunate that Haskell already implemented the multi-stage constructs in the form of Typed Template Haskell. However, it was quickly evident that there were serious issues with the implementation which required careful consideration. The most serious of these being the soundness issues which have been the focus on my work for the past three years. Once the soundness issues are settled it will perhaps be time to write some multi-stage programs.

This thesis has attempted to tackle foundational issues in the practical design of multi-stage languages. We have considered both the theoretical challenges of designing a language which interacts correctly with other common language features but also tackled the challenge of implementing these ideas in a production-grade compiler. The implementation is itself not production ready yet but we are confident that the ideas expressed in this thesis will guide the implementation to be ready in the near future.

A glimpse of what is possible to achieve using Typed Template Haskell was presented in Section 5.5 and so we are hopeful that the work on exploiting multi-stage programming will continue into the future.

Bibliography

Adams, M. D., Farmer, A., and Magalhães, J. P. (2014). Optimizing SYB is easy! In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, page 71–82, New York, NY, USA. Association for Computing Machinery.

Augustsson, L. (2018). `geniplate-mirror-0.7.6` library.

Bahr, P. (2020). Modal FRP for all: Functional reactive programming without space leaks in Haskell. Submitted to POPL 2021.

Bahr, P., Graulund, C. U., and Møgelberg, R. E. (2019). Simply RaTT: A Fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.*, 3(ICFP).

Berthold, J. (2010). Orthogonal serialisation for Haskell. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages*, IFL'10, page 38–53, Berlin, Heidelberg. Springer-Verlag.

Bierman, G. M. and de Paiva, V. C. (2000). On an intuitionistic modal logic. *Studia Logica*, 65(3):383–416.

Bottu, G.-J., Karachalias, G., Schrijvers, T., Oliveira, B. C. d. S., and Wadler, P. (2017). Quantified class constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, page 148–161, New York, NY, USA. Association for Computing Machinery.

Brady, E. and Hammond, K. (2006). A verified staged interpreter is a verified compiler. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, page 111–120, New York, NY, USA. Association for Computing Machinery.

Brady, E., McBride, C., and McKinna, J. (2003). Inductive families need not store their indices. In *International Workshop on Types for Proofs and Programs*, pages 115–129. Springer.

Breitner, J. (2018). A promise checked is a promise kept: Inspection testing. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, page 14–25, New York, NY, USA. Association for Computing Machinery.

Calcagno, C., Taha, W., Huang, L., and Leroy, X. (2003). Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE03, pages 57–76. Association for Computing Machinery.

BIBLIOGRAPHY

Cardelli, L. (1988). Phase distinctions in type theory. Unpublished Manuscript.

Chakravarty, M. M. T., Keller, G., and Peyton Jones, S. (2005). Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253.

Clouston, R. (2018). Fitch-style modal lambda calculi. In *International Conference on Foundations of Software Science and Computation Structures*, pages 258–275. Springer.

Consel, C. and Danvy, O. (1991). For a better support of static data flow. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 496–519, Berlin, Heidelberg. Springer-Verlag.

Convent, L., Lindley, S., McBride, C., and McLaughlin, C. (2020). Doo bee doo bee doo. *Journal of Functional Programming*, 30:e9.

Davies, R. and Pfenning, F. (2001). A modal analysis of staged computation. *J. ACM*, 48(3):555–604.

den Heijer, B. (2011). Optimal loop breaker choice for inlining. Master's thesis.

Downen, P. and Ariola, Z. (2019). Compiling with classical connectives. In submission.

Dunfield, J. (2015). Elaborating evaluation-order polymorphism. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 256–268, New York, NY, USA. Association for Computing Machinery.

Eisenberg, R. A. (2016). *Dependent types in Haskell: Theory and practice*. University of Pennsylvania.

Eisenberg, R. A. and Peyton Jones, S. (2017). Levity polymorphism. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 525–539, New York, NY, USA. Association for Computing Machinery.

Eisenberg, R. A., Weirich, S., and Ahmed, H. G. (2016). Visible type application. In *European Symposium on Programming*, pages 229–254. Springer.

Epstein, J., Black, A. P., and Peyton Jones, S. (2011). Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA. ACM.

Farmer, A. (2015). *HERMIT: Mechanized Reasoning during Compilation in the Glasgow Haskell Compiler*. PhD thesis, University of Kansas, USA.

Farmer, A., Gill, A., Komp, E., and Sculthorpe, N. (2012). The hermit in the machine: A plugin for the interactive transformation of GHC core language programs. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, page 1–12, New York, NY, USA. Association for Computing Machinery.

Fogarty, S., Pasalic, E., Siek, J., and Taha, W. (2007). Concoqtion: Indexed types now! In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '07, page 112–121, New York, NY, USA. Association for Computing Machinery.

Forster, Y., Kammar, O., Lindley, S., and Pretnar, M. (2019). On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Journal of Functional Programming*, 29:e15.

Gratzer, D., Kavvos, G., Nuyts, A., and Birkedal, L. (2020). Multimodal dependent type theory. In submission.

Gratzer, D., Sterling, J., and Birkedal, L. (2019). Implementing a modal dependent type theory. *Proc. ACM Program. Lang.*, 3(ICFP).

Gundry, A. (2013). *Type inference, Haskell and dependent types*. PhD thesis, University of Strathclyde.

Gundry, A. (2015). A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 11–22, New York, NY, USA. ACM.

Hanada, Y. and Igarashi, A. (2014). On cross-stage persistence in multi-stage programming. In *International Symposium on Functional and Logic Programming*, pages 103–118. Springer.

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., et al. (1992). Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164.

Jay, B. and Peyton Jones, S. (2008). Scrap your type applications. In *International Conference on Mathematics of Program Construction*, pages 2–27. Springer.

Jones, M. P. (1992). *Qualified types: theory and practice*. PhD thesis, University of Oxford.

Jones, M. P. (1995a). Dictionary-free overloading by partial evaluation. *Lisp Symb. Comput.*, 8(3):229–248.

Jones, M. P. (1995b). Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, page 97–136, Berlin, Heidelberg. Springer-Verlag.

Jones, N. D., Gomard, C. K., and Sestoft, P. (1993). *Partial evaluation and automatic program generation*.

Kakutani, Y., Murase, Y., and Nishiwaki, Y. (2019). Dual-context modal logic as left adjoint of fitch-style modal logic. *Journal of Information Processing*, 27:77–86.

Kammar, O. and Plotkin, G. D. (2012). Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 349–360, New York, NY, USA. Association for Computing Machinery.

Kawata, A. and Igarashi, A. (2019). A dependently typed multi-stage calculus. In *Asian Symposium on Programming Languages and Systems*, pages 53–72. Springer.

Kiselyov, O. (2014). The design and implementation of BER MetaOCaml. In Codish, M. and Sumii, E., editors, *Functional and Logic Programming*, pages 86–102, Cham. Springer International Publishing.

Kiselyov, O. (2017). Generating code with polymorphic let: A ballad of value restriction, copying and sharing. *Electronic Proceedings in Theoretical Computer Science*, 241:1–22.

Kiselyov, O. (2018). Reconciling abstraction with high performance: A metaocaml approach. *Foundations and Trends in Programming Languages*, 5(1):1–101.

Kiss, C., Pickering, M., and Wu, N. (2018). Generic deriving of generic traversals. *Proc. ACM Program. Lang.*, 2(ICFP).

Kmett, E. (2018). `lens-4.16` library.

Kokaji, Y. and Kameyama, Y. (2011). Polymorphic multi-stage language with control effects. In *Asian Symposium on Programming Languages and Systems*, pages 105–120. Springer.

Kusee, W. (2017). Compiling Agda to Haskell with fewer coercions. Master's thesis.

Levy, P. B. (2004). *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2)*. Kluwer Academic Publishers, USA.

Lewis, J. R., Launchbury, J., Meijer, E., and Shields, M. B. (2000). Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 108–118, New York, NY, USA. ACM.

Magalhães, J. P. (2012). Optimisation of generic programs through inlining. In *Symposium on Implementation and Application of Functional Languages*, pages 104–121. Springer.

Mainland, G. (2012). Explicitly heterogeneous metaprogramming with MetaHaskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*, pages 311–322, Copenhagen, Denmark.

Mainland, G. and Peyton Jones, S. (2010). Major proposed revision of Template Haskell.

Marlow, S. and Peyton Jones, S. (2012). The Glasgow Haskell Compiler. In Brown, A. and Wilson, G., editors, *The Architecture of Open Source Applications*, chapter 5.

Martínez, G., Ahman, D., Dumitrescu, V., Giannarakis, N., Hawblitzel, C., Hriţcu, C., Narasimhamurthy, M., Paraskevopoulou, Z., Pit-Claudel, C., Protzenko, J., et al. (2019). Meta-F* Proof Automation with SMT, Tactics, and Metaprograms. In *European Symposium on Programming*, pages 30–59. Springer, Cham.

McBride, C. and Paterson, R. (2008). Applicative programming with effects. *Journal of functional programming*, 18(1):1–13.

Nanevski, A. (2002). Meta-programming with names and necessity. page 206–217.

Nuyts, A. and Devriese, D. (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*, LICS '18, page 779–788, New York, NY, USA. Association for Computing Machinery.

Odersky, M., Burmako, E., and Petrashko, D. (2016). A TASTY alternative.

Pašalic, E. (2004). *The role of type equality in meta-programming*. PhD thesis, OGI School of Science & Engineering at OHSU.

Pašalic, E., Taha, W., and Sheard, T. (2002). Tagless staged interpreters for typed languages. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 218–229, New York, NY, USA. Association for Computing Machinery.

Peyton Jones, S., Jones, M., and Meijer, E. (1997). Type classes: an exploration of the design space. In *Haskell Workshop*.

Peyton Jones, S. and Marlow, S. (2002). Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434.

Peyton Jones, S. and Shields, M. (2002). Lexically scoped type variables. Microsoft Research.

Peyton Jones, S., Tolmach, A., and Hoare, T. (2001). Playing by the rules: rewriting as a practical optimisation technique in ghc. In *2001 Haskell Workshop*. ACM SIGPLAN.

Peyton Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. (2007). Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82.

Peyton Jones, S., Weirich, S., Eisenberg, R. A., and Vytiniotis, D. (2016). A reflection on types. In *A List of Successes That Can Change the World*, pages 292–317. Springer.

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

Pickering, M. (2019). Overloaded quotations. GHC proposal 246.

Pickering, M. (2020). Make Q (TExp a) into a newtype. GHC proposal 195.

Pickering, M., Löh, A., and Wu, N. (2020). Staged sums of products. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, Haskell 2020, pages 122–135, New York, NY, USA. Association for Computing Machinery.

Pickering, M., Wu, N., and Kiss, C. (2019a). Multi-stage programs in context. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, Haskell 2019, page 71–84, New York, NY, USA. Association for Computing Machinery.

Pickering, M., Wu, N., and Németh, B. (2019b). Working with source plugins. In *Proceedings of the 2019 ACM SIGPLAN Symposium on Haskell*, Haskell '19, New York, NY, USA. Association for Computing Machinery.

Plotkin, G. and Pretnar, M. (2009). Handlers of algebraic effects. In Castagna, G., editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg. Springer Berlin Heidelberg.

Rompf, T. (2012). *Lightweight Modular Staging and Embedded Compilers Abstraction without Regret for High-Level High-Performance Programming*. PhD thesis, Lausanne.

Rompf, T. (2016). The essence of multi-stage evaluation in LMS. In *A List of Successes That Can Change the World*, pages 318–335. Springer.

Rompf, T. and Odersky, M. (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*, GPCE '10, pages 127–136, New York, NY, USA. ACM.

Roubinchtein, E. (2015). IR-MetaOCaml: (re)implementing MetaOCaml. Master's thesis, University of British Columbia.

Sampson, A., McKinley, K. S., and Mytkowicz, T. (2017). Static stages for heterogeneous programming. *Proc. ACM Program. Lang.*, 1(OOPSLA).

Santos, A. L. M. (1995). *Compilation by transformation in non-strict functional languages*. PhD thesis, University of Glasgow.

Serrano, A., Hage, J., Peyton Jones, S., and Vytiniotis, D. (2020). A quick look at impredicativity. *Proc. ACM Program. Lang.*, 4(ICFP).

Serrano, A., Hage, J., Vytiniotis, D., and Peyton Jones, S. (2018). Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 783–796, New York, NY, USA. ACM.

Sheard, T. and Peyton Jones, S. (2002). Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA. ACM.

Stucki, N., Biboudis, A., and Odersky, M. (2018). A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, page 14–27, New York, NY, USA. Association for Computing Machinery.

Sulzmann, M., Chakravarty, M. M. T., Jones, S. P., and Donnelly, K. (2007). System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, page 53–66, New York, NY, USA. Association for Computing Machinery.

Taha, W. (2004). *A Gentle Introduction to Multi-stage Programming*, pages 30–50. Springer Berlin Heidelberg, Berlin, Heidelberg.

Taha, W. and Nielsen, M. F. (2003). Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 26–37, New York, NY, USA. Association for Computing Machinery.

Taha, W. and Sheard, T. (1997). Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '97, pages 203–217, New York, NY, USA. ACM.

Taha, W. and Sheard, T. (2000). MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242.

Theriault, A. (2019). Levity polymorphic lift. GHC proposal 209.

Vytiniotis, D., Peyton Jones, S., Schrijvers, T., and Sulzmann, M. (2011). OutsideIn(X) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA. ACM.

Weirich, S., Voizard, A., de Amorim, P. H. A., and Eisenberg, R. A. (2017). A specification for dependent types in Haskell. *Proc. ACM Program. Lang.*, 1(ICFP).

Willis, J., Wu, N., and Pickering, M. (2020). Staged selective parser combinators. *Proc. ACM Program. Lang.*, 4(ICFP).

Winant, T., Cockx, J., and Devriese, D. (2017). Expressive and strongly type-safe code generation. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, pages 199–210, New York, NY, USA. ACM.

Yallop, J. (2017). Staged generic programming. *Proc. ACM Program. Lang.*, 1(ICFP).

Yallop, J., von Glehn, T., and Kammar, O. (2018). Partially-static data as free extension of algebras. *Proc. ACM Program. Lang.*, 2(ICFP).

Yang, E. Z., Campagna, G., Ağacan, O. S., El-Hassany, A., Kulkarni, A., and Newton, R. R. (2015). Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 362–374, New York, NY, USA. Association for Computing Machinery.