

Scalable Metaprogramming in Scala 3



THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the registrar's office.

Thesis

Nicolas Stucki

jury :

| | |
|----------------------|--------------------|
| Prof. Viktor Kunčak | président du jury |
| Prof. Martin Odersky | directeur de thèse |
| Prof. Christoph Koch | rapporteur interne |
| Prof. Walid Taha | rapporteur externe |
| Dr. Ningning Xie | rapporteur externe |

Lausanne, EPFL, 2022

“With great power comes great responsibility”
— Benjamin F. Parker



Acknowledgements

Thesis advisor I would like to thank my advisor *Martin Odersky* for his support and motivation during my academic work. I am grateful for the opportunity to do my Ph.D. studies with him and to work as an engineer in the LAMP laboratory. I am glad that *Martin* guided me towards the design of the new metaprogramming features of Scala, a challenging and gratifying job.

Thesis committee I want to thank my thesis committee *Walid Taha*, *Ningning Xie*, and *Christoph Koch* for reading the thesis and the interesting discussions that followed. I am especially grateful to *Walid Taha* for creating multi-stage programming, the core on which this thesis is based. I also want to thank *Viktor Kunčak* for taking the jury president duties last minute.

Papers and thesis I want to thank *Fengyun Liu*, *Aggelos Biboudis*, *Sébastien Doeraene* and *Paolo Giarrusso* for their contributions to the papers that were used as a basis for the thesis. Furthermore, I am particularly grateful to *Sébastien Doeraene* for kindly helping with revisions of the entire thesis.

Metaprogramming design I especially thank *Fengyun Liu* and *Aggelos Biboudis* for their help while developing, implementing, and formalizing the system. Their contribution to the design and implementation of the system was invaluable. I am also thankful to *Jonathan Brachthäuser* and *Aleksander Boruch-Gruszecki* for help with the formalization of the system.

Scala compiler and language I want to thank *Dmytro Petrashko* for his mentoring on the compiler details. I also want to thank *Guillaume Martres*, *Olivier Blanvillain* and *Aleksander Slawomir Boruch-Gruszecki* for all their input to the metaprogramming system regarding the compiler and the Scala language itself. I am also grateful to *Sébastien Doeraene* and *Paolo Giarrusso* for their help in understanding and finding a solution to problems of type testing.

From Scala 2 to Scala 3 I want to thank *Eugene Burmako* and *Denys Shabalin* for their previous work on the Scala 2 macros system. This work was an indispensable basis for developing the Scala 3 macros system. As well, I want to thank *Anatolii Kmetiuk*, *Fengyun Liu* and *Aggelos Biboudis* for their help porting Scala 2 macros into Scala 3 and for supporting the community in this effort. Their work was vital to be able to ship Scala 3 with the essential core macro libraries of the Scala 2.

Scala professors I want to thank *Martin Odersky* and *Viktor Kunčák* for allowing me the opportunity to be their head assistant for the programming courses; teaching and managing were new and rewarding experiences for me. I also thank *Ondřej Lhoták* for all the insightful discussions we had during his visits to EPFL.

Path to Ph.D. studies I want to thank *Vlad Ureche* and *Aleksandar Prokopec* for being advisors on my projects during my Master's studies. These projects guided me towards Scala and my Ph.D. studies in LAMP.

The office I want to thank my office mates *Manohar Jonnalagedda*, *Heather Miller*, and *Matthieu Bovel* for a fun and relaxed work environment. I want to thank *Natascha Fontana* for all the administrative help she provided. I would also like to thank *Fabien Salvi* for all the technical infrastructure support that he provided.

Good times I am especially thankful for all the fun activities in and around Lausanne with *Aggelos Biboudis*, *Aleksander Boruch-Gruszecki*, *Allan Renucci*, *Anatolii Kmetiuk*, *Claudia Melcarne*, *Darja Jovanovic*, *Denys Shabalyn*, *Dragana Milovancevic*, *Eugene Burmako*, *Felix Mulder*, *Fengyun Liu*, *Georg Schmid*, *Guillaume Martres*, *Heather Miller*, *Jamie Thompson*, *Jonathan Brachthäuser*, *Jorge Vicente Cantero*, *Julien Richard-Foy*, *Lisa Hult*, *Mahathi Jonnalagedda*, *Manohar Jonnalagedda*, *Maria Gazaki*, *Matthieu Bovel*, *Mia Primorac*, *Ólafur Páll Geirsson*, *Olivier Blanvillain*, *Oskar van Rest*, *Sandro Stucki*, *Sébastien Doeraene*, *Théodore Note*, *Valentine Dubus*, *Vojin Jovanovic*, and many more.

COVID-19 My recovery from *COVID-19* has not been easy. I would like to thank everyone for their patience and support during my (long) *long COVID-19* recovery. Without everyone's support, I would not have been able to finish this thesis.

Family I would like to extend my sincere thanks to my parent *Urs* and *Valérie*. Their loving care and support was crucial throughout my studies and allowed me to pursue a Ph.D. degree. I am also thankful to my two grandmothers: *Edith*, who provided encouragement and financial support for my studies, and *Georgette*, whose devotion to teaching inspired me to follow a higher level of education. I am very grateful to my aunt *Evy* and her husband *Moca*, who kindly welcomed me at their home during my first years of studies in Switzerland. Last but not least, I would like to thank my wife *Camila*, who has been present in my life since the start of my Ph.D., making my days happier and believing in me at all times.

Nicolas Stucki
September 2022
Lausanne

Résumé

Un métadéveloppeur devrait être à même de raisonner à propos de la sémantique du code généré. La programmation multi-phase a proposé une solution à la fois élégante et puissante à ce problème. Il s'ensuit une approche de la génération de code fondée sur la sémantique, où celle-ci est entièrement définie par le métaprogramme, et ne peut être accidentellement altérée lorsque l'on génère le code. Cela implique que le code généré est bien typé et hygiénique par construction. L'on peut appliquer cette approche sémantique raisonnée à d'autres abstractions de métaprogrammation. En revanche, différentes abstractions exposent divers degrés d'expressivité. En général, plus les abstractions sont expressives, plus elles sont complexes et moins elles offrent de garanties statiques. Il est difficile, voire impossible, d'identifier une unique abstraction qui soit à la fois simple and parfaitement expressive. Plutôt que devoir choisir l'une ou l'autre de ces abstractions, nous pouvons concevoir un système unique formé de plusieurs abstractions exposant des niveaux différents d'expressivité et de complexité. L'on se doit d'être prudent avec les abstractions de métaprogrammation les plus expressives, compte tenu de ce qu'elles peuvent exposer certains détails du compilateur ou de sa représentation du code, et ainsi porter atteinte à la portabilité. Nous montrons qu'il est possible de concevoir, implémenter et utiliser en production un Système Portable et Évolutif de Métaprogrammation Fondé sur la Sémantique.

Mots-clés : métaprogrammation, macros, programmation multi-phase, inlining, formalisation, types algébriques de données virtuels, Scala

Abstract

A metaprogrammer should be able to reason about the semantics of the generated code. Multi-stage programming introduced an elegant and powerful solution to this problem. It follows a *semantically driven* approach to code generation, where semantics are fully defined by the metaprogram and cannot accidentally change when we generate the code. This implies that the generated code is well typed and hygienic by construction. We can apply this principled semantic approach to other metaprogramming abstractions. However, different metaprogramming abstractions have different levels of expressiveness. Usually, the more expressive abstractions are more complex and give fewer static guarantees. It is hard or impossible to find a single abstraction that is both simple and fully expressive. Instead of choosing a single abstraction, we can design a single system out of several abstractions that *scale* with respect to expressiveness and complexity. We must be careful with the most expressive metaprogramming abstractions, as they may expose parts of the compiler or of its code representations, which hinders *portability*. We demonstrate that it is possible to design, implement and use in production a *Portable Scalable Semantically Driven Metaprogramming System*.

Keywords: metaprogramming, macros, multi-stage programming, inlining, formalization, virtual algebraic data types, Scala

Contents

| | |
|--|------------|
| Acknowledgements | ii |
| Résumé | iv |
| Abstract | v |
| List of Figures | x |
| List of Code Examples | xi |
| List of Theorems, Lemmas and Definitions | xii |
| List of Tables | xiv |
| 1 Introduction | 1 |
| 1.1 Macros Design Principles | 1 |
| 1.2 Scala 2 Macros | 2 |
| 1.3 Scala 3 Macros | 3 |
| 1.4 Contribution | 5 |
| I Inlining for Metaprogramming | 7 |
| 2 Semantics-Preserving Inlining for Metaprogramming | 9 |
| 2.1 Inline Functions | 12 |
| 2.1.1 Inline Values | 13 |
| 2.1.2 Parameters of Inline Functions | 13 |
| 2.1.3 Recursion | 15 |
| 2.1.4 Inline Conditionals | 17 |
| 2.2 Inline Methods | 17 |
| 2.2.1 Members and Bridges | 17 |
| 2.2.2 Overloads | 18 |
| 2.2.3 Abstract Methods and Overrides | 19 |
| 2.3 Transparent Inlining | 24 |
| 2.4 Metaprogramming | 26 |
| 2.4.1 Inline Error | 26 |

| | | |
|-----------|---|-----------|
| 2.4.2 | Inline Pattern Matching | 26 |
| 2.4.3 | Inline Summoning | 27 |
| 2.4.4 | Inlining and Macros | 27 |
| 2.5 | Implementation | 29 |
| 2.6 | Applicability | 29 |
| 2.7 | Related Work | 30 |
| 2.8 | Future Work | 32 |
| 2.9 | Conclusion | 32 |
| | | |
| II | Multi-Stage Programming | 35 |
| | | |
| 3 | Macro and Run-Time Multi-Stage Programming | 37 |
| 3.1 | Macros and Run-Time Multi-Stage Programming | 39 |
| 3.1.1 | Multi-Staging | 39 |
| 3.1.2 | Quoted Values | 40 |
| 3.1.3 | Macros and Multi-Stage Programming | 42 |
| 3.1.4 | Safety | 45 |
| 3.1.5 | Staged Lambdas | 47 |
| 3.1.6 | Staged Constructors | 47 |
| 3.1.7 | Staged Classes | 48 |
| 3.1.8 | Quote Pattern Matching | 48 |
| 3.1.9 | Sub-Expression Transformation | 52 |
| 3.1.10 | Staged Implicit Summoning | 53 |
| 3.2 | Implementation | 54 |
| 3.2.1 | Syntax | 54 |
| 3.2.2 | Run-Time Representation | 55 |
| 3.2.3 | Entry Points | 58 |
| 3.2.4 | Compilation | 59 |
| 3.3 | Reflection | 70 |
| 3.4 | Related Work | 70 |
| 3.5 | Future Work | 72 |
| 3.6 | Conclusion | 74 |
| | | |
| 4 | Multi-Stage Macro Calculus | 77 |
| 4.1 | Multi-Stage Calculus | 78 |
| 4.2 | Core Calculus | 78 |
| 4.3 | Quoted Constants Calculus Extension | 83 |
| 4.4 | Quote Pattern Matching Extension | 85 |
| 4.5 | Global Definitions Extension | 91 |
| 4.6 | Patterns with Type Variables Extension | 97 |
| 4.7 | Parametric Polymorphism Extension | 103 |
| 4.8 | Polymorphic Multi-Stage Macro Calculus | 108 |

Contents

| | | |
|------------|--|------------|
| 4.8.1 | Syntax | 108 |
| 4.8.2 | Environments | 109 |
| 4.8.3 | Typing | 111 |
| 4.8.4 | Operational Semantics | 114 |
| 4.8.5 | Values | 119 |
| 4.9 | Concrete Syntax in Scala | 120 |
| 4.10 | Discussion and Related Work | 121 |
| 4.11 | Future Work | 123 |
| 4.12 | Conclusion | 123 |
| | | |
| III | Typed AST Reflection | 125 |
| | | |
| 5 | Virtual ADTs for Portable Metaprogramming | 127 |
| 5.1 | Scaling APIs with Virtual ADTs | 128 |
| 5.1.1 | Abstract Types: Separating Interface from Implementation | 130 |
| 5.1.2 | TypeTest: Supporting Run-Time Type Tests | 132 |
| 5.1.3 | Extension Methods: Restoring the Interface | 133 |
| 5.1.4 | Abstract Objects: Encoding Companions | 134 |
| 5.1.5 | Unapply Methods: Extractors | 134 |
| 5.1.6 | Singletons: Case Objects | 135 |
| 5.1.7 | Summary | 135 |
| 5.2 | Discussion | 137 |
| 5.2.1 | Changing the Internal Representation | 137 |
| 5.2.2 | Changing the Interface | 137 |
| 5.2.3 | Monomorphism | 138 |
| 5.2.4 | Limitations | 138 |
| 5.3 | Related Work | 139 |
| 5.4 | Future Work | 140 |
| 5.5 | Conclusion | 140 |
| | | |
| 6 | A TASTy Reflection Interface | 143 |
| 6.1 | TASTy Binaries | 143 |
| 6.2 | Overview of the Reflection API | 144 |
| 6.3 | Multi-Stage Programming with Reflection | 146 |
| 6.4 | TASTy Inspector | 149 |
| 6.5 | Decompiler | 150 |
| 6.6 | Macro Annotations | 150 |
| 6.7 | Related Work | 152 |
| 6.8 | Future Work | 153 |
| 6.9 | Conclusion | 153 |

| | |
|--|------------|
| Epilogue | 155 |
| 7 Academic Projects | 157 |
| 8 Core Library, Tools and Community Projects | 159 |
| 9 Conclusion | 161 |
| | |
| Appendix | 163 |
| | |
| A Soundness Proof of the Polymorphic Multi-Stage Macro Calculus | 165 |
| A.1 Proof of Progress | 165 |
| A.2 Proof of Preservation | 169 |
| | |
| Bibliography | 199 |
| | |
| Curriculum Vitae | 207 |

List of Figures

| | | |
|------|--|-----|
| 4.1 | Core Calculus | 79 |
| 4.2 | Quoted Constants Calculus Extension | 84 |
| 4.3 | Structural Quote Patterns Calculus | 86 |
| 4.4 | Pattern Semantics | 87 |
| 4.5 | Global Definitions Extension | 92 |
| 4.6 | Pattern Type Variables Calculus | 97 |
| 4.7 | Pattern Semantics | 98 |
| 4.8 | Parametric Polymorphism Extension | 104 |
| 4.9 | Pattern Semantics | 105 |
| 4.10 | Pattern Unification Semantics | 106 |
| 4.11 | Syntax | 108 |
| 4.12 | Environments | 109 |
| 4.13 | Well-Formed Environment | 110 |
| 4.14 | Well-Formed Type | 110 |
| 4.15 | Well-Formed Constraint | 110 |
| 4.16 | Program Typing | 111 |
| 4.17 | Term Typing | 112 |
| 4.18 | Pattern Typing | 113 |
| 4.19 | Program Operational Semantics | 114 |
| 4.20 | Term Operational Semantics (a) | 115 |
| 4.21 | Term Operational Semantics (b) | 116 |
| 4.22 | Pattern Semantics | 117 |
| 4.23 | Pattern Structural Matching | 117 |
| 4.24 | Pattern Type Unification | 118 |
| 4.25 | Program Values | 119 |
| 4.26 | Term Values | 119 |
| 5.1 | Virtual ADT Interface for Peano Numbers | 136 |
| 5.2 | Virtual ADT Implementation with Case Classes | 136 |
| 5.3 | Virtual ADT Implementation with BigInt | 138 |

List of Code Examples

| | | |
|------|-------------------------------|-----|
| 3.1 | def unrolledPowerCode | 39 |
| 3.2 | given OptionToExpr | 41 |
| 3.3 | def powerCode | 41 |
| 3.4 | def powerMacro | 42 |
| 3.5 | scala.quoted.staging.run | 44 |
| 3.6 | def fusedPowCode | 48 |
| 3.7 | given OptionFromExpr | 49 |
| 3.8 | HOAS pattern | 49 |
| 3.9 | def fuseMapCode | 50 |
| 3.10 | def let | 51 |
| 3.11 | def empty[T] | 51 |
| 3.12 | trait ExprMap | 52 |
| 3.13 | def treeSetFor | 53 |
| 3.14 | def setFor | 53 |
| 3.15 | Quote syntax | 54 |
| 3.16 | Splice syntax | 55 |
| 3.17 | class Expr | 55 |
| 3.18 | object Expr | 56 |
| 3.19 | class Type | 56 |
| 3.20 | object Type | 57 |
| 3.21 | trait Quotes | 57 |
| 3.22 | scala.quoted.staging.Compiler | 59 |
| 3.23 | def emptyList | 62 |
| 6.1 | def switch | 147 |
| 6.2 | def unswitch | 148 |
| 6.3 | object PrintCode | 149 |

List of Theorems, Lemmas and Definitions

| | | |
|----|---|-----|
| 1 | Theorem (Progress for Terms) | 82 |
| 2 | Theorem (Preservation for Terms) | 82 |
| 3 | Theorem (Progress for Programs) | 96 |
| 4 | Theorem (Preservation for Programs) | 96 |
| 5 | Theorem (Progress for Terms) | 96 |
| 6 | Theorem (Preservation for Terms) | 96 |
| | | |
| 1 | Lemma (Canonical Forms) | 82 |
| 2 | Lemma (Extended Progress for Terms) | 82 |
| 3 | Lemma (Substitution) | 83 |
| 4 | Lemma (Multi-Substitution) | 90 |
| 5 | Lemma (Preservation of Pattern Reduction) | 90 |
| 6 | Lemma (Preservation for Match) | 90 |
| 7 | Lemma (Extended Progress for Terms) | 96 |
| 8 | Lemma (Substitution) | 96 |
| 9 | Lemma (Σ -Weakening) | 96 |
| 10 | Lemma (Constraints of Pattern Reduction) | 102 |
| 11 | Lemma (Pattern Constraints Unification) | 102 |
| 12 | Lemma (Constraint Substitution) | 102 |
| 13 | Lemma (Type Substitution) | 102 |
| 14 | Lemma (Pattern Type Substitution) | 102 |
| 15 | Lemma (Well-Formed Type Substitution) | 102 |
| 16 | Lemma (Constraints of Pattern Reduction) | 107 |
| 17 | Lemma (Pattern Constraints Unification) | 107 |
| 18 | Lemma (Unification Locality) | 107 |
| | | |
| 1 | Definition (Restricted Typing Context) | 82 |
| 2 | Definition (Well-Formed Φ) | 90 |
| 3 | Definition (Well-Formed Ω) | 95 |

List of Theorems, Lemmas and Definitions

| | | |
|------|--|-----|
| A.1 | Theorem (Progress for Programs) | 165 |
| A.2 | Theorem (Progress for Terms) | 166 |
| A.3 | Theorem (Preservation for Programs) | 169 |
| A.4 | Theorem (Preservation for Terms) | 171 |
| | | |
| A.1 | Lemma (Canonical Forms) | 166 |
| A.2 | Lemma (Extended Progress for Terms) | 166 |
| A.3 | Lemma (Σ -Weakening) | 171 |
| A.4 | Lemma (Preservation for Match) | 172 |
| A.5 | Lemma (Well-Formed Constraint Shuffle) | 173 |
| A.6 | Lemma (Type Well-Formedness Weakening) | 174 |
| A.7 | Lemma (Preservation of Pattern Reduction) | 174 |
| A.8 | Lemma (Constraint of Pattern Reduction) | 179 |
| A.9 | Lemma (Pattern Constraints Unification) | 184 |
| A.10 | Lemma (Unification Locality) | 187 |
| A.11 | Lemma (Well-Formed Type Weakening) | 187 |
| A.12 | Lemma (Constraint Union) | 188 |
| A.13 | Lemma (Substitution) | 188 |
| A.14 | Lemma (Multi-Substitution) | 190 |
| A.15 | Lemma (Type Substitution Well-formedness) | 190 |
| A.16 | Lemma (Type Substitution) | 192 |
| A.17 | Lemma (Well-Formed Weak Type Substitution) | 194 |
| A.18 | Lemma (Type Multi-Substitution) | 194 |
| A.19 | Lemma (Pattern Type Substitution) | 194 |
| A.20 | Lemma (Constraint Substitution) | 196 |
| | | |
| A.1 | Definition (Well-formed Ω) | 165 |
| A.2 | Definition (Restricted Typing Context) | 166 |
| A.3 | Definition (Well-formedness of Φ) | 174 |



List of Tables

| | | |
|-----|---|----|
| 2.1 | Inline method overriding and implementation | 24 |
|-----|---|----|

1 Introduction

At its core, the Scala library ecosystem relies on metaprogramming. Most of this ecosystem relies on macros to generate new code, and analyze or modify existing code. Macros were introduced in Scala 2 and improved in Scala 3 by the work presented in this dissertation.

1.1 Macros Design Principles

Semantically driven A metaprogrammer should be able to reason about the semantics of the generated code. *Multi-stage programming* introduced an elegant and powerful solution to this problem. It follows a semantically driven approach to code generation, where *semantics are fully defined by the metaprogram and cannot accidentally change* when we generate the code. This implies that the generated code is well typed and hygienic by construction. We can apply this principled semantic approach to other metaprogramming abstractions.

Scalability Different metaprogramming abstractions have different levels of expressiveness. Usually, the more expressive abstractions are more complex and have fewer static guarantees. Instead of choosing a single abstraction, we can design a single system out of several abstractions that scale with respect to expressiveness and complexity. In this system, *simple and common metaprogramming use cases have safe and straightforward implementations, and less common complex uses cases can be implemented at the cost of some complexity.*

Portability The most expressive metaprogramming abstractions expose part of the compiler's internal APIs or code representations. This imposes some practical limitations on the free evolution of the compiler. To *allow macro implementations to be used across compilers and the free evolution of the compiler*, we need to design a proper abstraction over the compiler functionality and code representation.

In this thesis we demonstrate that it is possible to design, implement and use in production a *Portable Scalable Semantically Driven Metaprogramming System*

1.2 Scala 2 Macros

Scala 2 provided a powerful macro system using an AST reflection API. Even though the system was labeled as experimental, it became one of the cornerstones of the language. This design clearly showed the potential of macros in Scala but did introduce some inconvenient features that affected macro implementations and compiler implementations of the macro system.

Syntactic and semantic macros The Scala 2 implementation of macros was quite flexible and powerful, allowing the creation of typed (semantic) and untyped (syntactic) ASTs. The latter provided ad-hoc support for hygiene, protecting against accidental rebinding of variables. This is not enough for the Scala language due to its complex program elaboration during type-checking. Implicit resolution, implicit conversion, overload resolution, and extension methods can affect how the program is elaborated. For example, we could accidentally choose the wrong overload because of the implicits available at call site, capture an unintended implicit argument, or accidentally apply an implicit conversion. These issues arise in the part of the system that is *not semantically driven*.

Low-level first The design of Scala 2 macros is centered on ASTs and then extended with syntactic quasiquotation. This implies that developers of macros must learn all the details of ASTs before using a more straightforward quasiquotation system. They also must be aware of program elaboration and type-checking. This made the macro system *scalable over its expressivity but unscalable over its complexity*.

Unportability Unfortunately, the architectural design of the metaprogramming API bound it to the implementation of the compiler. This came at the cost of reduced portability of macro implementation across compiler versions. Furthermore, this design also made it harder to evolve the Scala 2 compiler as it had to fall in line with the metaprogramming interface. Crucially, it made it *impossible to re-implement the same system in Scala 3*.

1.3 Scala 3 Macros

We use *multi-stage programming at the core of our solution*. It is simple to use and provides strong static guarantees. We use semantics-preserving inline definitions to create multi-stage macros. We design a system that works both for compile-time and run-time code generation and analysis. We extend the multi-stage programming system with a reflection API similar to Scala 2, but we only operate on typed code. All these abstractions follow the *semantically driven* principle. Together they provide a genuinely *scalable* metaprogramming system ranging from the simplicity of inlining and multi-stage programming to the expressivity of AST manipulation.

Scala 3 introduced *TASTy* as a new high-level intermediate representation providing a portability layer between compiler versions. The TASTy format defines an abstract representation of fully elaborated Scala programs. It is the perfect basis to support the internal implementations of all semantically driven metaprogramming. We use the format directly to serialize staged and inline code. We also use this abstraction to define our reflection API using a Virtual ADT encoding to avoid binding the interface with the compiler.

Overview

In this thesis, we consolidate several publications and add new material. We divided the thesis into three parts covering different topics. These topics scale from simple metaprogramming abstractions to more expressive metaprogramming abstractions.

Part I: Inlining for Metaprogramming In Chapter 2 we describe the design of semantics-preserving inlining for metaprogramming. Inlining is an essential component of multi-stage macros and other simpler macro operations.

```
/** Specialize the operation based on static knowledge of the value of `n` */
inline def powerMacro(x: Double, inline n: Int): Double =
  ${ powerCode('x, 'n) }
```

Part II: Multi-Stage Programming We cover the design, implementation, and formalization of Scala 3 multi-stage programming. In Chapter 3 we describe the design and implementation, and in Chapter 4 we provide a formal calculus for multi-stage macros. The calculus is proven sound in Appendix A.

```
/** If `n` is known, generate `{ x * ... * x }` with `n` repetitions of `x` */
def powerCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
  unrolledPowerCode(x, n.valueOrAbort)
```

Introduction

```
/** Generate `{ x * ... * x }` with `n` repetitions of `x` */
def unrolledPowerCode(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
  if n == 0 then '{ 1.0 }
  else if n == 1 then x
  else '{ $x * ${ unrolledPowerCode(x, n - 1) } }
```

Part III: Typed AST Reflection We cover the design and implementation of the reflection API. In Chapter 5 we cover the design of the Virtual ADT encoding used for the reflection API, and in Chapter 6 we describe the reflection API, its uses, and its interactions with multi-stage programming.

```
/** Generate specialized variants of `x^n` for small `n`s.
 *  `{
 *    n match
 *      case 0 => 1.0
 *      case 1 => x
 *      case 2 => x * x
 *      ...
 *      case _ => Math.pow(x, n.toDouble)
 *  }
 */
def switchPower(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
  import quotes.reflect.*
  val caseDefs: Seq[CaseDef] =
    for i <- 0 to numCases yield
      CaseDef(Literal(IntConstant(i)), None, unrolledPowerCode(x, i).asTerm)
  val defaultCaseDef =
    CaseDef(Wildcard(), None, '{ Math.pow($x, $n.toDouble) }.asTerm)
  Match(n.asTerm, caseDefs.toList :+ defaultCaseDef).asExprOf[Double]
```

Epilogue We list related academic projects in Chapter 7. We list libraries and tools that use Scala 3 metaprogramming in Chapter 8. We finally conclude the thesis in Chapter 9.

1.4 Contribution

In the context of this thesis, we designed and implemented the Scala 3 metaprogramming system. To achieve this result, we made the following contributions.

Multi-stage macros using inlining We created a new *multi-stage programming* extension that uses inline methods to define macros.

Design of semantics-preserving inlining for metaprogramming We created a specification for inline methods that support macros and simpler metaprogramming operations. Under the constraints of semantic preservation, we made the system as general as we could.

Unification of multi-stage macros and multi-stage programming We created the first implementation of a multi-stage programming system allowing macros and run-time code generation.

Multi-stage macro calculus We designed a generative and analytical multi-stage macro calculus that captures all features present in Scala 3 macros.

Design of Virtual ADTs We designed a novel ADT encoding in Scala that can completely decouple an interface from its implementation. We also introduced sound type testing for abstract types in Scala to ensure the soundness of the encoding.

Use of TASTy as the metaprogramming lingua franca We show the usefulness of a high-level intermediate binary representation for semantic metaprogramming. It is used for inlining code, serializing staged code, and defining the reflection API.

TASTy inspector We show that this approach can also scale to other kinds of metaprogramming. The TASTy inspector gives a view on a TASTy binary using the same reflection API. We also show a novel approach to code decompilation using this tool.

Scala 3 implementation We fully implemented our metaprogramming system in Scala 3.

Scala ecosystem migration We took part in the first steps of the migrations of core libraries of the ecosystem. This effort provided the initial inertia needed for the rest of Scala 2 projects using macros to migrate to Scala 3.

Inlining for Metaprogramming **Part I**

2 Semantics-Preserving Inlining for Metaprogramming

This chapter contains a published paper authored by Stucki, Biboudis, Doeraene, and Odersky [72].

Programming languages (e.g., C++ [69], F# [78], D [3]) usually offer inlining as a compiler directive for optimization purposes. In some of these, an inline directive is mandatory to trigger inlining, in others it is just a hint for the optimizer. The expectation from a user's perspective is simple: the semantic reasoning for a method call should remain unaffected by the presence of inlining. In other words, inlining is expected to be *semantics-preserving* and consequently this form of inlining can be done late in the compiler pipeline. Inlining is typically implemented in the backend of a compiler, where code representations are simpler to deal with. For example, in the following snippet of code, we would like to avoid one method call, thus the method body itself replaces the call.

```
inline def square(x: Int): Int = x * x

square(y) // inlined as y*y to avoid a call to square at run-time
```

Some programming languages do not provide inlining at the language level and rely on automatic detection of inlining opportunities instead. Java, with its dynamic features such as dynamic dispatch and hot execution paths, pushes the inlining further down the pipeline at the level of the JVM. Scala primarily relies on the JVM for performance, though it can inline while compiling to help the JIT compiler.

However, there is another angle that we can view inlining from. Inlining can be seen as a form of metaprogramming, where inlining is the syntactic construct that turns a program into a program generator [67; 45]. Indeed, the snippet above describes a metaprogram, the metaprogram that is going to generate a method body at the call site. This may seem like a pointless observation, but in this chapter we show that metaprogramming through inlining can be seen as a structured, and *semantics-preserving* methodology for metaprogramming.

Semantics-Preserving Inlining for Metaprogramming

Typically, in the aforementioned programming languages, inlining takes a piece of untyped code and types it at the call site. Therefore the semantics are only defined at the call site and there are no semantics to preserve.

```
// C++
#define square(X) X * X // * does not have any semantics here
square(y) // inlined as y*y and then typed
```

Sometimes such inlining is done using standard call notation (as in C++ `#define`), and at other times it is done using a different language fragment (as in C++ templates). In these cases, inlining is not necessarily semantics preserving and usually does not provide type-safety guarantees at the definition site. In the latter case, C++ allows code that may not generate valid code for some of the type parameters of the template, and is only checked for the actual type arguments. There is no guarantee that an expanded inline call will always type-check.

Non-semantic inlining can be categorized as *syntactic inlining*. To see the difference between the two, consider this example with overloaded methods:

```
def f(x: Any): Int = 1
def f(x: String): Int = 2

inline def g[T](x: T): Int = f(x)

g("abc")
```

When using semantics-preserving inlining, the `inline` keyword can be dropped from the signature without changing the result. That means that the call `f(x)` resolves to the first alternative and the result is 1. With syntactic inlining, we inline the call to `g`, expanding it to `f("abc")`. This then resolves to the second alternative of `f`, yielding 2. So, in a sense, syntactic inlining replaces overloading resolution with compile-time multi-method dispatch.

Syntactic inlining is very powerful and has been used to great effect to produce heavily specialized code. However, it can also be difficult to reason about, leading to errors in expanded code that are hard to track down.

Other compile-time metaprogramming constructs have inlining as an implicit part of what they do. For instance, a macro in Lisp [28] or Scala 2 [12] moves the code of the macro to the call site (this is a form of inlining) and then executes the macro's code at this point. Can we disentangle inlining from the other metaprogramming features? This is the approach followed in Scala 3 [21]. It offers a powerful set of metaprogramming constructs, including staging with quotes `'{...}` and splices ``${...}``. Quotes delay the execution of code while splices compute code that will be inserted in a larger piece of code. Staging is turned from a run-time code-generation feature to a compile-time macro feature by combining it with inlining.

A macro is an inline function with a top-level splice. For example:

```
inline def powerMacro(m: Double, inline n: Int): Double =  
  ${ powerCode('{m}', '{n}') }
```

A call site such as `powerMacro(x, 4)` is expanded by inlining its implementation and following argument evaluation semantics (described in Section 2.1.2):

```
val m = x  
${ powerCode('{m}', '{4}') }
```

The content of this splice is then executed in the context of the call site at compile-time.

When used in conjunction with other metaprogramming constructs, inlining has to be done early, typically during type-checking, because that is when these other constructs apply. Furthermore, it makes sense that inlining by itself should be as “boring” as possible. It should be type-safe and semantics-preserving by default. At the same time, inlined definitions should be usable and composable in interesting ways. For instance, since normal methods can override methods in parent classes or implement abstract methods, it makes sense to allow the same flexibility for inlined methods, as far as is possible.

These considerations lead us to the following principles:

- *Semantics-preserving*: A call to an inline method should have exactly the same semantics as the same method without the inline modifier.
- *Generality*: We want to be able to use and define inline methods as generally as possible, as long as (1) is satisfied.

In an object-oriented language like Scala, the concern for generality poses several interesting questions, which are answered in this chapter:

- Can inline methods implement abstract methods?
- Can inline methods override concrete methods?
- Can inline methods be overridden themselves?
- Can inline methods be abstract themselves?

There is another question here that will influence the answers to these four questions:

- Can inline methods be called at run-time?

Semantics-Preserving Inlining for Metaprogramming

It will turn out that the answer to this question is “it depends”. Some inline methods will need to be callable at run-time, in order to preserve semantics. Others cannot be called at run-time because they use metaprogramming constructs that can only be executed at compile-time.

In this chapter, we explore these questions by presenting the rationale and design of Scala 3’s inlining concept and how it relates to its metaprogramming architecture.

In Section 2.1 we discuss how our design of inline functions is based on the principles above. In Section 2.2 we extend the discussion to the design of inline methods. In Section 2.3 we introduce a simple extension to inline functions that can affect the semantics at call site (not the call itself). In Section 2.4 we show some of the metaprogramming features that can be built on top of semantics-preserving inlining. We conclude by discussing related work in Section 2.7.

2.1 Inline Functions

We introduce the `inline` modifier to denote that a function is an inline function. A function with `inline` can be called as any other function would.

```
inline def logged[T](logger: Logger, x: T): Unit =  
  logger.log(x)
```

Assuming that `Logger` has a proper definition of `log`, the code would type-check. Inlining this code seems simple enough as shown below.

```
logged(new RefinedLogger, 3)  
// expands to:  
// (new RefinedLogger).log(3)
```

But what if the definitions of `log` were the following?

```
class Logger:  
  def log(x: Any): Unit = println(x)  
  
class RefinedLogger extends Logger:  
  override def log(x: Any): Unit = println("Any: " + x)  
  def log(x: Int): Unit = println("Int: " + x)
```

If we look at `logger.log(x)` we can see that the only option is to call `def log(x: Any)` defined in `Logger`. By examining the inline site, one would argue that the method to be invoked should be `def log(x: Int)` defined in `RefinedLogger`. However, this would imply a change in the semantics of the code after inlining where the overloading resolution results

in different method selection depending on whether we use inlining or not. With or without inline, the code should perform the same operation, therefore we need to retain the original overload resolution to avoid breaking the first principle.

The elaboration of extension methods¹, implicit conversions, and implicit resolution must be preserved as these are part of the overload resolution. All of these could change if they are performed with different type information, which could potentially end up calling different methods.

While overloading should not change, it is possible to perform *de-virtualization* without breaking semantics. De-virtualization is an optimization that precomputes the virtual dispatch resolution (override) that would otherwise happen at run-time. In the example, we would directly call the `def log(x: Any)` defined in `RefinedLogger`.

2.1.1 Inline Values

`val` definitions can also be marked as `inline`. An `inline val` inlines the contents of its right-hand side (RHS) as an `inline def` would. Unlike `inline defs`, when inlining an `inline val` we cannot inline any arbitrary RHS as it may recompute the values several times, which would break the evaluation order semantics.

```
inline val x = 4
x // replaced with 4

def z: Int = ...
inline val y = z // error: z is not a known value
y // cannot replace y with z as z may have side effects
```

Therefore we constrain the RHS to be pure and to be able to reduce to a value at compile-time.

This means that the RHS can only contain literal values or references that reduce to a literal constant such as another `inline val` or `def`.

2.1.2 Parameters of Inline Functions

To support *semantics-preserving* inlining in the presence of effects during the evaluation of arguments, the latter must be let-bound at the call site. To illustrate this, consider the square inline function.

```
inline def square(x: Int): Int =
  x * x
```

¹Extension methods allow one to add methods to a type after the type is defined – <https://dotty.epfl.ch/docs/reference/contextual/extension-methods.html>

Semantics-Preserving Inlining for Metaprogramming

This function could be called with an arbitrary parameter which could have side effects. As Scala provides by-value call semantics, the argument expression must be evaluated once before the evaluation of the body of the function. The solution is to let-bind the evaluation of the expression passed as an argument to preserve the evaluation order. In the following example, the method call `n` contains an I/O operation:

```
def n: Int =
  scala.io.StdIn.readInt()

square(n)
// expands to:
//   val x1 = n
//   x1 * x1
```

If we were to inline it as `n * n`, we would mistakenly read two numbers from the standard input. This shows why it is important to have let-bound arguments.

Scala also provides by-name parameters. These parameters need to be evaluated each time they are referred to.

```
inline def twice(thunk: =>Unit): Unit =
  thunk
  thunk

twice { print("Hello!") } // prints: Hello!Hello!
// expands to:
//   def thunk = print("Hello!")
//   thunk
//   thunk
```

Instead of binding them to a `val` we bind them to a `def`. We do not replace each reference `thunk` by `print("Hello!")` to avoid code duplication.

Constant folding After methods and `vals` are inlined, we can perform constant folding optimizations on primitive types. This implies that constants are propagated and primitive operations are performed on them.

```
square(3)
// expands to:
//   val x1 = 3; x1 * x1
// optimized to:
//   3 * 3
// then optimized to:
//   9
```

Additionally, if constant folding evaluates the condition of an `if` to a known value, then we can partially evaluate the `if` and eliminate one of the branches. This allows a limited but simple way to generate simplified code. More complex and domain-specific optimizations demand the use of custom metalanguage code with macros.

Inline parameters In some cases, we do not want to let-bind the arguments and instead we wish to inline them directly where they are used. For this purpose, we allow parameters to be marked as `inline`, but only in `inline` functions. This makes it a metaprogramming feature as it provides semantics that are not expressible in normal functions. These parameters have, by construction, semantics that are similar to by-name and may generate duplicated code. The `inline` parameters allow further specialization of code by duplicating code and allowing each copy to be specialized in a different way. This specialization might come from further inlining or by one of the metaprogramming features.

For example, it is possible to remove closure allocations early on using inline parameters.

```
inline def tabulate3[T](inline f: Int => T): List[T] = List(f(0), f(1), f(2))

tabulate3(x => 2*x)
// expands to:
// List((x => 2*x)(0), (x => 2*x)(1), (x => 2*x)(2))
// which reduces to:
// List(0, 2, 4)
```

Without the `inline` parameter, we would have been forced to let-bind the instantiation of the closure which may have side effects. An optimizer might remove it only if there are no side effects where we can ensure that the whole expression is ignored.

2.1.3 Recursion

Inline functions can call other inline functions and in particular themselves. Calls to an inline function `f` within another inline function `g` are not immediately inlined within the body of `g`. Instead they are only inlined once `g` has itself be inlined in a third, non-`inline` function `h`.

```
inline def f(): Int =
  3

inline def g(): Int =
  f() // f not inlined here

def h(): Int =
  g() // first inlines g then inlines f
```

Semantics-Preserving Inlining for Metaprogramming

Now consider the recursive inline function `power`.

```
inline def power(x: Double, n: Int): Double =
  if n == 0 then 1.0
  else if n == 1 then x
  else if n % 2 == 1 then x * power(x, n - 1)
  else power(x * x, n / 2)

power(expr, 10)
// expands to:
//   val x = expr      // x^1
//   val x1 = x * x    // x^2
//   val x2 = x1 * x1  // x^4
//   val x3 = x2 * x   // x^5
//   x3 * x3          // x^10
```

Note the importance of parameter semantics: if `x` would not be let-bound the computation would be linear instead of the expected logarithmic time. In this example, we assume that `n` will be a constant and that it can be constant folded in the conditions of the `ifs`. In turn, we assumed that after constant folding only one branch will be kept and eventually will stop the recursion. This will not always be the case.

With recursive inlining we introduce potentially non-terminating inline expansions. Consider the previous example, but with an unknown value of `n`.

```
power(expr, m)
// expands in a first step to:
//   val x = expr
//   val n = m
//   if n == 0 then 1.0
//   else if n == 1 then x
//   else if n % 2 == 1 then x * power(x, n - 1)
//   else power(x * x, n / 2)
```

It is apparent that we could take one more unfolding step to the next call of `power` and then recursively do the same again, so we would never end. The expansion will continue until a predefined maximum inline depth limit² is reached and fail compilation.

As we ensure that all calls to inline functions are inlined or a compilation failure occurs, we never need to call these methods at run-time. This implies that the inline function definitions can be removed from the generated code.

²This limit can be increased by the user if necessary

2.1.4 Inline Conditionals

An `inline if` provides a variant of `if` that must be constant-folded in its condition to eliminate one of the branches. If that cannot be done, an error is emitted and no further expansion within the `if` is attempted. Using `inline if` ensures that we always partially evaluate the `if` at compile-time. An `inline if` and an `if` have the exact same semantics at run-time.

This is also useful as an explicit convergence check when using recursive inline functions.

```
inline def power(x: Double, n: Int): Double =
  inline if n == 0 then 1.0
  else inline if n == 1 then x
  else inline if n % 2 == 1 then x * power(x, n - 1)
  else power(x * x, n / 2)

power(expr, m)
// expands in a first step to:
//   val x = expr
//   val n = m
//   inline if n == 0 then 1.0
//   else inline if n == 1 then x
//   else inline if n % 2 == 1 then x * power(x, n - 1)
//   else power(x * x, n / 2)
```

As `n==0` does not have a known value at compile-time, the expansion fails and no further nested expansions are attempted. The same happens for the other nested `inline if`.

2.2 Inline Methods

Inline can also be used for methods in classes or traits. Inline methods will be able to access object fields and interact with virtual dispatch.

2.2.1 Members and Bridges

An inline method may refer in its body to the `this` reference of the current class or to any private member. Let us consider the following inline method defined in a class.

```
class InlineLogger:
  private var count = 0

  inline def log[T](op: () => T): Unit =
    val result = op() // may contain call to log
    count += 1
    println(count + "> " + result)
```

Semantics-Preserving Inlining for Metaprogramming

First, the method evaluates the operation, then it updates the private field count, and then prints it with the result. Note that the operation may contain nested calls to `log` which would use the current count.

```
def inlineLogger: InlineLogger =
  new InlineLogger

inlineLogger.log(() => 5)
// naive expansion:
//   val ths = inlineLogger
//   val result = (() => 5)()
//   ths.count += 1
//   println(ths.count + "> " + result)
```

We need to make sure the prefix of the application (i.e., the receiver) is only evaluated once by let-binding it to `ths`. Then we use `ths` in place of `this` in the inlined code. Unfortunately, the inlined code contains a reference to the private field `count` which is not accessible from the call site (under the JVM model). This does not break *semantics-preservation* but does greatly limit what could be used in the body of an inline method.

To lift this limitation, we instead generate bridges for all members that may not be accessible at the call site. For the `count` we would create a getter and setter that make the bridge possible. This ensures that when the call is inlined all references are still accessible.

```
class InlineLogger:
  private var count = 0

  final def inline$count: Int = // only in generated code
    count

  final def inline$count_=(x: Int): Unit = // only in generated code
    count = x

  inline def log[T](op: () => T): Unit =
    val result = op()
    this.inline$count_=(this.inline$count + 1)
    println(this.inline$count + "> " + result)
```

2.2.2 Overloads

As inline methods must be *semantics-preserving*, the definition and resolution of overloads should not be affected. The overload resolution algorithm does not need any modification, hence it considers all inline and non-inline functions as equivalent. For example, the following variants perform the same overload resolution.

```
def log(msg: String): Unit = ...
def log(x: Any): Unit = ...
log("a")
```

```
inline def log(msg: String): Unit = ...
inline def log(x: Any): Unit = ...
log("a")
```

```
def log(msg: String): Unit = ...
inline def log(x: Any): Unit = ...
log("a")
```

```
inline def log(msg: String): Unit = ...
def log(x: Any): Unit = ...
log("a")
```

2.2.3 Abstract Methods and Overrides

Inline methods implementing interfaces Consider the following example, where we have an inline definition implementing a non-inline abstract method.

```
trait Logger:
  def log[T](op: () => T): Unit

class InlineLogger extends Logger:
  inline def log[T](op: () => T): Unit = println(op())
```

If we have an instance of `InlineLogger` we can inline the code. But now we also allow calls to `Logger.log` which will not be inlined.

```
def logged[T](logger: Logger, x: () => T): Unit =
  logger.log(x)

logged(new InlineLogger, 3)
```

This implies that we have a call to `Logger.log` at run-time, which should be dispatched to `InlineLogger.log`. Therefore if the inline method implements an interface, we cannot ensure it will be completely inlined and we must retain the code at run-time.

Semantics-Preserving Inlining for Metaprogramming

Inline methods overriding normal methods Consider the following example, where we have an inline definition that overrides a non-inline method.

```
class Logger:
  def log[T](op: () => T): Unit = println(op())

class NoLogger extends Logger:
  inline def log[T](op: () => T): Unit = ()
```

If we have an instance of `NoLogger` we can inline the code. But once again we also allow calls to `Logger.log`, which will not be inlined. Unlike with the implementation of the abstract method, it would be tempting to say that, as there exists an implementation of `log`, we could remove `NoLogger.log` from the generated code. However, that would not be semantics-preserving. In order to ensure that calling `Logger.log` on a `NoLogger` does indeed no logging, we must also keep the implementation of `NoLogger.log` at run-time. Then, virtual dispatch will be able to find the correct implementation at run-time.

Overriding inline methods Consider the following example, where we have an inline method that is overridden by another method.

```
class Logger:
  inline def log[T](op: () => T): Unit =
    println(op())

class NoLogger extends Logger:
  /*inline*/ def log[T](op: () => T): Unit =
    ()
```

This time we turned things around and are trying to override an inline method with any method (inline or not). Using the same logged example we have a different way in which semantics-preservation fails.

```
def logged[T](logger: Logger, x: T): Unit =
  logger.log(x) // expanded to the contents Logger.log

logged(new NoLogger)(3)
```

As `log` is inlined from `Logger` before we know which logger we are using, we will always call `Logger.log`. Instead, we would have expected to call `NoLogger.log` which is a semantic breakage.

In general, no inline method can be safely overridden as it bypasses virtual dispatch resolution. Therefore all inline methods are *effectively final*.

Abstract inline methods Consider the following example of an abstract inline method.

```
trait AbstInlineLogger:
  inline def log[T](op: () => T): Unit
```

It would be possible to implement this interface with a non-inline function as it would perfectly preserve the semantics. But this does not offer any expressivity advantage over a normal abstract method. Instead, we will restrict it to only be implementable by inline methods to guarantee that the calls can be inlined. Unlike plain abstract methods, the abstract inline method does not enforce the implementations of inline methods to be retained at run-time.

```
class InlineLogger extends AbstInlineLogger:
  inline def log[T](op: () => T): Unit =
    println(op())

class NoLogger extends AbstInlineLogger:
  inline def log[T](op: () => T): Unit =
    ()
```

It is clear that all implementations of `log` will be inlined if we know statically the receiver of the `log` call which defines the methods. But, can we ever call `AbstInlineLogger.log` directly and still have it inlined?

```
def logged[T](logger: AbstInlineLogger, x: () => T): Unit =
  logger.log(x) // error: cannot inline abstract method
```

Calling it directly will not work as it is impossible to inline. However, by inlining the previous code we can get this abstraction to work.

```
inline def logged[T](logger: AbstInlineLogger, x: () => T): Unit =
  logger.log(x)

logged(new InlineLogger, () => 5)
logged(new NoLogger, () => 6)
```

Now, when `logged` is inlined, the call `logger.log` gets de-virtualized at compile-time and then can be inlined. Crucially, with abstract inline methods, we provide a way to guarantee that all calls to such methods are de-virtualized and inlined at compile-time.

Semantics-Preserving Inlining for Metaprogramming

Inline methods overriding with inline parameters Consider the following example where a method is overridden with an inline method and its parameter is marked inline.

```
class Logger:
  def log[T](x: T): Unit = println(x)

class NoLogger extends Logger:
  inline override def log[T](inline x: T): Unit = ()

val noLogger: NoLogger = new NoLogger
noLogger.log(f()) // expands to: ()

val logger: Logger = noLogger
logger.log(f())
```

Here, `inline` has a deeper effect and provides the possibility to override the call semantics. Whenever we call `Logger.log`, the arguments will be evaluated with the standard by-value semantics. In this case, this implies the evaluation of `f()` which might have side effects. But, when calling `NoLogger.log` the evaluation of the argument is just dropped. As a consequence, the call semantics changed and this pattern should not be allowed.

Abstract inline methods and inline parameters Consider the following example of an abstract inline method with an inline parameter. We implement it with a method that has the same signature.

```
trait AbstInlineLogger:
  inline def log[T](inline x: T): Unit

class InlineLogger extends AbstInlineLogger:
  inline def log[T](inline x: T): Unit = println(x)

class NoLogger extends AbstInlineLogger:
  inline def log[T](inline x: T): Unit = ()

inline def logged[T](logger: AbstInlineLogger, inline x: T): Unit =
  logger.log(x)

val inlineLogger: InlineLogger =
  new InlineLogger
logged(inlineLogger, f()) // expands to: println(f())

val noLogger: NoLogger =
  new NoLogger
logged(noLogger, f()) // expands to: ()
```

With this pattern, the semantics of the inline parameter `x` are preserved across all abstractions, until the de-virtualization of `AbstInlineLogger.log` into `InlineLogger.log` (which preserves the call to `f()`) and `NoLogger.log` (which eliminates it). Therefore, we can allow abstract methods to have inline parameters, as long as all its implementations use corresponding inline parameters as well. This pattern shows another useful reason to have abstract inline methods: regular abstract methods cannot have inline parameters, as we saw earlier, while abstract inline methods can.

Now, consider an alternative implementation of `NoLogger` that does evaluate the argument but does not print it.

```
class NoLogger extends AbstInlineLogger:
  inline def log[T](inline x: T): Unit =
    val y = x
    ()

val noLogger = new NoLogger
logged(noLogger, f())
// expands to:
//   val y = f()
//   ()
```

It also works, and we just emulated by-value parameters. Using a `def` for the parameter would emulate a by-name parameter. This kind of parameter semantic emulation is not always possible, particularly if we have an inline parameter followed by a by-value parameter. In the following example, the `cond` argument will always be evaluated before the `msg` parameter because `cond`.

```
inline def assert(inline msg: String, cond: Boolean): Unit =
  val message = "Assertion failure: " + msg
  if cond then
    throw new AssertionError(message)

assert("expected 1", x == 1)
// expands to:
//   val cond = x == 1
//   val message = "Assertion failure: " + "expected 1"
//   if cond then
//     throw new AssertionError(message)
```

Inline methods summary All inline methods are `final`. Abstract inline methods can only be implemented by inline methods. If an inline method overrides/implements a normal method then it must be retained (i.e., cannot be erased). Retained methods cannot have inline parameters. Table 2.1 shows the overriding rules.

Semantics-Preserving Inlining for Metaprogramming

| | (abstract) method | inline method | abstract inline method |
|------------------------|-------------------|---------------|------------------------|
| - | ◐ | ◑ | ◒ |
| (abstract) method | ◐ | ● | ◑ |
| inline method | × | × | × |
| abstract inline method | × | ◑ | ◒ |

◐ runtime call ◑ inlined ● runtime call and inlined × disallowed
 ◒ inlined after de-virtualization ◑ runtime call and inlined after de-virtualization

Table 2.1: Inline method overriding and implementation. Method definitions listed on the left are implemented or overridden by method definitions listed on the top. The first row describes the behavior of method definitions that do not implement nor override any definitions.

2.3 Transparent Inlining

A simple but powerful metaprogramming extension to inlining is the ability to *refine* the type of an expression after the call is inlined. The inline call is typed and inlined retaining its elaboration as with normal inline functions. But instead of typing the inlined expression as the result type of the inline function, we take the precise type of the inlined expression. This unlocks the ability to change the semantics at the call site around the inlined call, without changing the semantics of the code that was inlined. We use the `transparent` keyword to enable this feature.

```
transparent inline def choose(b: Boolean): Any =
  if b then 3 else "three"

val obj1: Int = choose(true)
val obj2: String = choose(false)
```

This may be used to influence type inference, overload resolution, and implicit resolution at the call site. But as with all inlines, it may not change the elaboration of the inlined code. To illustrate this, consider the following code where we have a definition of a method that is overwritten, overloaded and returns a more precise type.

```

class A:
  def f(a: A): A = ...

class B extends A:
  override def f(a: A): B = ...
  def f(x: B): String = ...

transparent inline def g(inline a1: A, inline a2: A): A =
  a1.f(a2)

val b: B = ???
val y = g(b, b) // expands to: val y: B = b.f(b)

def h(a: A): Unit = println("A")
def h(b: B): Unit = println("B")
h(y) // prints "B" because g is transparent (otherwise would be "A")

```

From Section 2.2, we know that for the call semantics to be preserved we need to make sure that the inlined call to `f` should be to `B.f(A)` at run-time. Before inlining the code in `g`, the call to `f` returned an `A` as we were calling `A.f(A)`. After inlining the code in `g`, this same call gets de-virtualized and we know that we actually call `B.f(A)` and it returns a `B`. Hence the inlined expression is of type `B` and `y` is inferred to be a `B` as well. Then the rest of the code outside the call is subject to this more precise type.

The call to `h(y)` will be statically resolved to a call to the `h(B)` overload as `y` was typed as `B`. On the other hand, if `g` had been a normal function or a non-transparent inline function, the type of `y` would have been `A`. In this case the overload resolution would have chosen `h(A)`. This shows how transparent inline function can affect the semantics around their call site.

To be able to propagate the types as we do we need to inline while typing. Any non-transparent inlining can be performed after typing. Each call to a transparent inline will respect the semantic preservation. In contrast to non-transparent inline function, replacing it with the same function without transparency may change the semantics of the overall program. It is possible to emulate the transparent semantics adding casts that align with the results of the implementation.

```

/*transparent*/ inline def choose(b: Boolean): Any =
  if b then 3 else "three"

val obj1: Int = choose(true).asInstanceOf[Int]
val obj2: String = choose(false).asInstanceOf[String]

```

For this to be sound we would need to prove that these casts have exactly the type of the result based on the types or statically known values of the arguments.

2.4 Metaprogramming

With metaprogramming, we introduce metalanguage features that allow code manipulation. In general, these features only manipulate code in place which maintains the metaprogramming abstractions simple. This is possible because `inline` takes care of placing the metaprogram where it needs to be. In most cases, these metaprogramming features do not have run-time semantics until expanded at the call site. But all of them rely on the knowledge that the code around them or in their parameters preserved their semantics when inlining. These features may be combined with transparent inlining. In this section, we have a non-exhaustive list of metaprogramming features that are supported by inlining.

2.4.1 Inline Error

An error method provides a way to emit custom error messages. The error will be emitted if a call to `error` is inlined and not eliminated as a dead branch.

```
import scala.compiletime.error

inline def div(n: Int, m: Int): Int =
  inline if m == 0 then error("Cannot divide by 0")
  else n / m
```

`error` is not subject to the semantics-preservation principle, since it is illegal in code that is retained at run-time. The same observation applies to most metaprogramming features described in this section.

2.4.2 Inline Pattern Matching

This variant of `match` provides a way to match on the static type of some expression. It ensures that only one branch is kept. In the following example, the scrutinee, `x`, is an inline parameter that we can pattern match on at compile-time.

```
transparent inline def half(inline x: Any): Any =
  inline x match
    case x: Int => x / 2
    case x: Double => x / 2.0d

half(1.0d) // expands to: 1.0d / 2.0d
half(2) // expands to: 2 / 2
val n: Any = 3
half(n) // error: n is not statically known to be an Int or a Double
```

The `inline match` will use the static type of the scrutinee and keep the branch that matches

said type. For this to work, the patterns must be non-overlapping. Unlike the `inline if`, this reduction is not necessarily equivalent to its run-time counterpart when we have more type information.

2.4.3 Inline Summoning

If we need to summon implicit evidence provided by the call site within a method we generally need to pass it as an argument of that method. But we may want to conditionally generate different code based on the existence of such implicit. This is not possible if it is part of the arguments as it would require it before expanding the code.

For this purpose, we introduce a set of delayed summoning (such as `summonInline` and `summonFrom`) that can be used within the body of an `inline` but will only be resolved at call site.

```
import scala.compiletime.summonFrom
inline def setFor[T]: Set[T] =
  summonFrom {
    case ord: Ordering[T] => new TreeSet[T](ord)
    case _                 => new HashSet[T]
  }
```

In a sense, `summonFrom` is a transparent `inline` as the expanded expression will have the type of the body of the chosen branch.

2.4.4 Inlining and Macros

Here is how we define a macro that generates code to compute the power of a number.

```
inline def powerMacro(x: Double, inline n: Int): Double =
  ${ powerCode('{x}, '{n}) }
```

The program is split into the macro definition `powerMacro` and code generators/analyzers `powerCode` and `powerUnrolled`.

```
def powerCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
  n.value match
    case Some(m) => unrolledPowerCode(x, m) // statically known n
    case None   => '{ Math.pow(${x}, ${n}) }

def unrolledPowerCode(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
  if n == 0 then '{ 1.0 }
  else if n == 1 then x
  else '{ ${x} * ${ unrolledPowerCode(x, n - 1) } }
```

Semantics-Preserving Inlining for Metaprogramming

The macro metalanguage provides two core constructs for code manipulation.

- `'{...}` quotes representing code fragments of type `Expr [T]` where `T` is the type of the code within
- `#{...}` splices that insert code fragments into larger code fragments

The code directly in quotes delays the execution of the code, while the code within the splices computes a code fragment now. For example, `'{Math.pow({x}, {n})}` represents a snippet where we will insert the code of `x` and `n`. The operation `Expr.value` allows us to extract the value of `n` if it is known at the call site.

If we use a splice outside of a quote, as in `powerMacro`, we call it a macro. Such a splice will evaluate its contents at compile-time. To make this evaluation efficient we require the code within the top-level splice to be a simple static call to a precompiled function. This way we only interpret a single reflective function call which then executes any user-defined compiled code. In theory it would be possible to let the users call this directly using the metalanguage.

```
#{ powerCode('{x}, '{2}) } // would expand to: x * x * 1
```

But if the users had to use the metalanguage directly, the usability of such a feature would have a high complexity cost. Instead, by using `inline` we can hide the metalanguage behind a normal method call that does not mention the metalanguage. We also avoid expanding the macros before the code is inlined.

```
inline def powerMacro(x: Double, inline n: Int): Double =  
  #{ powerCode('{x}, '{n}) }
```

In this model, the macro expansion logic simply needs to evaluate and replace a piece of code. Now the users of this macro only need to know how to call a method.

```
powerMacro(x, 2)  
// expanded by inline to:  
//   #{ powerCode('{x}, '{2}) }  
// then by macro to:  
//   x * x
```

For macro overrides, we additionally expand the macro inside of the `inline` function as it must exist at run-time. In this case, the macro should also be able to generate a generic fallback version of the code that does not have the call site information.

Given that the implementation of the macro is done directly in the language rather than the metalanguage, the macro can execute arbitrary code at compile-time. This provides extra flexibility and expressivity that is not available when using the metalanguage constructs directly.

2.5 Implementation

Next, we describe the implementation of inlining, as described in this work, as merged in the Scala 3 compiler.

Inlining is performed while typing and inlines fully elaborated typed ASTs. The reason for this design choice is to support the implementation of transparent inlining. One of the very first steps we need to make is to obtain the typed ASTs. This can be done either via the definitions that we are currently typing or from a published TASTy (serialized AST in a binary format) [51] artifact. TASTy contains the fully elaborated typed ASTs of a complete class. From this artifact, we can extract the original AST of the method. Quoted code fragments are also encoded in TASTy.

Once we have the AST, the next step is to perform β -reduction. Most of the complexity comes from making sure that during inlining we make all types as precise as possible without changing the resolution of overloads. When we perform the inlining, we make sure that all references in the code will be accessible at any inline site by generating public accessors if needed.

It is worth noting that overload resolution did not change. Extra checks were added to make sure that the override constraints hold. These constraints are summarized at the end of Section 2.2. Furthermore, all inline method definitions are erased from the code except if marked as *retained*. The RHS of retained methods is evaluated as if inlined to execute any metaprogramming features.

2.6 Applicability

Using inlining as a compiler directive is already widely used and we advocate that the extra restrictions on method overriding are portable to any OO programming language.

The use of inlining as a base for metaprogramming could be used in other compiled languages in general. The quotes and splices were inspired by MetaML and MetaOCaml for run-time code generation. They were transformed into compile-time code generation by simply allowing top-level splices inline methods. Other metaprogramming features like the error and pattern matching would also be useful in many languages.

2.7 Related Work

F# supports inlining of generic functions [78]. However, since generic numeric code—code that uses primitive operators—is treated differently for each numeric type, the mechanism of inlining demands specialized support for type inference. As a result, inline generic functions can have statically resolved type parameters whereas non-inline functions cannot. Scala, in combination with our work, does not infer a different type for the following method:

```
inline def f[T: Numeric](x: T, y: T): T =  
  x + x * y
```

The compiler resolves overloaded methods uniformly and orthogonally to the inlining mechanism (note that `Numeric` is a view bound). F# infers statically resolved type parameters in the inferred type of the corresponding definition of `f` in F#. F# does not support the equivalent of `inline if`, `inline match`, `inline overriding` or the equivalent to `transparent`.

In C++, inlining is a compiler hint that an optimizer may or may not follow. However, inlining is not binding compiler implementors to use inline substitution for any function that is not marked `inline` and vice versa. Similarly to our system, C++17 supports both function and variable inlining. It is worth noting that since C++ supports external linkage, linking behavior needs to be changed to support inlining.

`constexpr` was one of the additions in C++11 and proves crucial in simplifying template metaprogramming. A constant expression defines that an expression can be evaluated at compile-time and is implied to be `inline`. A constant function can return a constant value and may or may not be evaluated at compile-time. In C++20, `constexpr` denotes *immediate functions* (not semantically equivalent to normal functions), which are guaranteed to be inlined and evaluated at compile-time. C++ supports `if constexpr` statements, similar to our `inline if`. In Scala, constant expressions are specified by a very limited set of rules, hence evaluation occurs only inside `inline ifs` and pattern matches. Right-hand sides of `inline` values and arguments for `inline` parameters must be constant expressions in the sense defined by the SLS § 6.24, including platform-specific extensions such as constant folding of pure numeric computations. `constexpr` comes with a very complex set of rules that defines what a `constexpr` function is; essentially a completely specified sub-language. In our work, we decide to abstain from strong compile-time evaluation guarantees to support semantics-preserving inlining. Since all the aforementioned variants of constant expressions in C++ offer a very powerful set of compile-time evaluation in C++, we can compare that aspect too. Firstly, constant expressions are strictly term-level features (as opposed to template metaprograms). In our work, as shown by `transparent` inlining we can refine the type of an expression after the call is inlined.

D [3] supports the usual compiler directive called `inline`. Like C++ it is an advice to the compiler. Similarly to C++, D is also equipped with a powerful template metaprogramming capability.

While C++ uses a functional style for templates, in D a template looks like imperative code, so syntactically D is very close to what a user would write at the term level. Our `inline if`, similarly to D supports conditional compilation based on arguments.

D's *Compile Time Function Execution (CTFE)* is also part of the compile-time metaprogramming but on the interpretation side instead of merely inlining. D functions that are portable and free from side effects can be executed at compile-time. While inlining is a declaration-level directive in our work, in D it is triggered by various “static” contexts such as a `static-if` or a dimension argument in a `static array`. One of the limitations in D is that the function source code must be available while in our work it can also be loaded from compiled code. In the compiler, CTFE comes after the AST manipulation phase (naming, type assignment, etc) has been completed and performs essentially interpretation much like C++. However, as in C++, D cannot introduce new types (or more precise types) in the context.

MetaML [81; 82; 80] and MetaOCaml [40] offer a distinction between the metalanguage and an object language via staging annotations—brackets, `escape` and `run`. The aforementioned syntactic modalities are introduced to denote where the evaluation needs to be deferred and we already cross the boundary of semantics-preserving code. At this point, we can navigate and guide freely the process of generating code from a quoted domain-specific language as shown in past work [70; 42; 63]. The macro system that comes in Scala 3, described in Section 2.4.4 essentially gains inspiration from these technologies and completes what we present in this work. Racket is considered to have one of the most advanced macro systems and racket macros can be viewed as compiler extensions that can expand syntax into existing forms. The work we present in this chapter differs greatly from this direction.

Swift supports cross-module inlining and specialization with two attributes: `inlinable` and `usableFromInline`. The first can be applied to functions and methods, among others, exposing that declaration's implementation as part of the module's public interface. The second introduces a notion of an Application Binary Interface (ABI)-public declaration. Swift's attributes offer an inlining mechanism similar to C++. The most important distinction between Swift and our work is that `inlinable` declarations can only reference ABI-public declarations while we also support access to private methods via bridges. Our work provides automated detection and generation of said bridges at the cost of potentially leaking private implementation details out of the public ABI.

2.8 Future Work

Currently, we provide several out-of-the-box metaprogramming solutions to be used just by inlining while others require full-blown macros. For example, `inline if`, `inline match`, `summonInline` and `error` are all supported in some form by macros but we provide simpler primitives for those operations. As future work, we should identify a core set of metaprogramming features that are often required and provide implementations for them in the standard library.

Changing the implementation of transparent inline methods may break source compatibility, while a normal inline method may break binary compatibility. We need to explore how those can be mitigated and if it is possible to automatically detect all these cases. Swift's approach to the ABI might be considered for this purpose, even though it has an extra syntactic overhead.

We empirically test the correctness of the inlining system by comparing the result of equivalent functions with and without inlining. It would be good to formally prove the soundness of this inlining system. It would be interesting to prove the unsoundness of the system if the overriding restrictions are removed.

2.9 Conclusion

In this work we introduce an inline language primitive to support metaprogramming features. We showed the importance of preserving the semantics while inlining, including the implications of having methods and virtual dispatch. We listed a few metaprogramming features that use inlining and showed how the metalanguage takes advantage of inlining to remain simple.

Multi-Stage Programming **Part II**

3 Macro and Run-Time Multi-Stage Programming

This chapter contains extracts of published papers authored by Stucki, Brachthäuser, and Odersky [73] and authored by Stucki, Biboudis, and Odersky [70]. The content is updated and extended to reflect the latest developments.

Generative programming [17] is used in scenarios such as code configuration of libraries, code optimizations [86], and DSL implementations [18; 83]. There are various kinds of program generation systems, ranging from syntax-based and unhygienic, to fully typed [67; 45]. Modern macro systems, like Racket's, can extend the syntax of the language [24] to create hierarchies of domain-specific languages [6]. In this work, we are not concerned with Racket-like language extensibility, but rather macros that can generate and analyze code of expressions at compile-time.

Principled approaches to metaprogramming, such as MetaML [82; 79; 80] and BER MetaOCaml [13; 40; 39; 38], offer strong foundations for expression code generation. These systems focus on run-time code generation and ensure static safety (well typed and hygienic) and cross-stage safety. They usually provide cross-stage persistence (CSP) of local variables, the ability to refer to values from previous stages, at the cost of not supporting *cross-platform portability* [82]. Cross-platform portability, as defined by Taha and Sheard [82], describes the ability to move code generated on one machine to a (potentially different) machine, to compile and execute it there. This ability is necessary for multi-stage macros in compiled languages with separate compilation. To move code from one machine to the next, cross-platform portability requires code serialization, either in the form of source code or as a serialized intermediate representation. Multi-stage systems that support portability usually need to make some compromises. Some miss analytical capabilities while others allow both generative and analytical macros by resorting to advanced type system machinery. Nowadays, many programming languages provide support for similar mechanisms such as F#, Haskell (Template Haskell [66] and later Typed Template Haskell [32]), Converge [85] and others.

MacroML [26] extended MetaML to provide a multi-stage macro system showing that “*multi-stage programming languages are a good foundation for the semantics-based design of macro systems*” [26]. By design, MacroML takes a conservative approach not to blur the distinction between code and data, explicitly avoids dynamic scoping, and lacks analytical macros. Modular Macros [88] prototyped a compile-time variant of MetaOCaml which takes a similar approach.

Macro and Run-Time Multi-Stage Programming

Monnier and Shao [49] first expressed inlining as staged computation but MacroML offered a user-level perspective by reusing the same mechanisms of quotes and splices, where splices can appear at the top-level (not nested in a quote). Squid [56; 57; 58] provides a multi-stage system for generative and analytical macros. Unlike MacroML, Squid uses statically typed dynamic scoping, tracking free term variables in types, to provide type-safe analytical macros.

While the same line of work inspired many metaprogramming libraries and language features, to our knowledge built-in support for both run-time multi-stage programming and generative macros has not been implemented previously in a unifying manner. We advocate that such a unification has two benefits: firstly, users rely on a single abstraction to express code generation; and secondly, having a single subsystem in the compiler favors maintainability. Our view regarding macros aligns with the benefits of multi-stage programming on domain-specific optimizations [16; 42; 39]: in modern programming languages, inlining (à la C++) with a *sufficiently smart* partial evaluator is not necessarily equivalent with domain-specific optimizations that can be done at compile-time.

Scala 2 already provided an experimental metaprogramming API called *scala.reflect* [12]. *scala.reflect* supports type-aware, run-time and compile-time code generation, providing an expressive and powerful system to the user (both generative and analytical). Despite the success of *scala.reflect*, the API exposed compiler internals and gave rise to portability problems between compiler versions [46]. This API provided syntactic code quasiquotation [65] which lacks the strong static safety provided by multi-stage programming.

We designed the Scala 3 metaprogramming core on multi-stage programming to provide simpler and safer code generation and analysis tools. An additional goal was to make most of the macros that were already available in Scala 2 also available in Scala 3, while making their implementation simpler and safer. To allow more expressivity, when multi-stage programming is too restrictive, we complement it with the reflection API (in Part III). In this chapter we focus on the design and implementation of the Scala 3 macros and run-time multi-stage programming.

Requirements We identify the requirements that a design of a multi-stage macro and run-time system for compiled languages should meet:

- *Cross-platform portability.* It should be possible to use generated code on different machines.
- *Static safety.* Generated code should be hygienic and well typed.
- *Cross-stage safety.* Access to variables should only be allowed at stages where they are available.
- *Generative and analytical.* Programmers should be able to generate as well as analyze and decompose code.

3.1 Macros and Run-Time Multi-Stage Programming

This section describes the design of the multi-stage system implemented in Scala 3. We also show how the requirements are met and how they guide some of the design decisions.

3.1.1 Multi-Staging

Quoted expressions Multi-stage programming in Scala 3 uses quotes `'{ . }` to delay, i.e., stage, execution of code and splices `${ . }` to evaluate and insert code into quotes. Quoted expressions are typed as `Expr [T]` with a covariant type parameter `T`. It is easy to write statically safe code generators with these two concepts. The following example shows a naive implementation of the x^n mathematical operation.

```
import scala.quoted.*
def unrolledPowerCode(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
  if n == 0 then '{ 1.0 }
  else if n == 1 then x
  else '{ $x * ${ unrolledPowerCode(x, n-1) } }
```

Listing 3.1: def unrolledPowerCode

```
'{
  val x = ...
  ${ unrolledPowerCode('{x}, 3) } // evaluates to: x * x * x
}
```

Quotes and splices are duals of each other. For an arbitrary expression `x` of type `T` we have ``${ '{x}' } = x` and for an arbitrary expression `e` of type `Expr [T]` we have `'{ ${e} } = e`.

Abstract types Quotes can handle generic and abstract types using the type class `Type [T]`. A quote that refers to a generic or abstract type `T` requires a given `Type [T]` to be provided in the implicit scope. The following examples show how `T` is annotated with a context bound `(: Type)` to provide an implicit `Type [T]`, or the equivalent using `Type [T]` parameter.

```
import scala.quoted.*
def singletonListExpr[T: Type](x: Expr[T])(using Quotes): Expr[List[T]] =
  '{ List[T]($x) } // generic T used within a quote

def emptyListExpr[T](using Type[T], Quotes): Expr[List[T]] =
  '{ List.empty[T] } // generic T used within a quote
```

If no other instance is found, the default `Type . of [T]` is used. The following example implicitly uses `Type . of [String]` and `Type . of [Option [U]]`.

Macro and Run-Time Multi-Stage Programming

```
val list1: Expr[List[String]] =
  singletonListExpr('{"hello"}) // requires a given `Type[String]`
val list0: Expr[List[Option[T]]] =
  emptyListExpr[Option[U]] // requires a given `Type[Option[U]]`
```

The `Type.of[T]` method is a primitive operation that the compiler will handle specially. It will provide the implicit if the type `T` is statically known, or if `T` contains some other types `Ui` for which we have an implicit `Type[Ui]`. In the example, `Type.of[String]` has a statically known type and `Type.of[Option[U]]` requires an implicit `Type[U]` in scope.

Quote context We also track the current quotation context using a given `Quotes` instance. To create a quote `'{..}` we require a given `Quotes` context, which should be passed as a contextual parameter (using `Quotes`) to the function. Each splice will provide a new `Quotes` context within the scope of the splice. Therefore quotes and splices can be seen as methods with the following signatures, but with special semantics.

```
def '[T](x: T): Quotes => Expr[T] // def '[T](x: T)(using Quotes): Expr[T]
def ${T}(x: Quotes => Expr[T]): T
```

The lambda with a question mark `?=>` is a contextual function; it is a lambda that takes its argument implicitly and provides it implicitly in the implementation the lambda. `Quotes` are used for a variety of purposes that will be mentioned when covering those topics.

3.1.2 Quoted Values

Lifting While it is not possible to use cross-stage persistence of local variables, it is possible to lift them to the next stage. To this end, we provide the `Expr.apply` method, which can take a value and lift it into a quoted representation of the value.

```
val expr1plus1: Expr[Int] = '{ 1 + 1 }
val expr2: Expr[Int] = Expr(1 + 1) // lift 2 into '{ 2 }
```

While it looks type wise similar to `'{ 1 + 1 }`, the semantics of `Expr(1 + 1)` are quite different. `Expr(1 + 1)` will not stage or delay any computation; the argument is evaluated to a value and then lifted into a quote. The quote will contain code that will create a copy of this value in the next stage. `Expr` is polymorphic and user-extensible via the `ToExpr` type class.

```
trait ToExpr[T]:
  def apply(x: T)(using Quotes): Expr[T]
```

3.1 Macros and Run-Time Multi-Stage Programming

We can implement a `ToExpr` using a given definition that will add the definition to the implicits in scope. In the following example we show how to implement a `ToExpr[Option[T]]` for any liftable type `T`.

```
given OptionToExpr[T: Type: ToExpr]: ToExpr[Option[T]] with
  def apply(opt: Option[T])(using Quotes): Expr[Option[T]] =
    opt match
      case Some(x) => '{ Some[T] ( ${Expr(x)} ) }
      case None => '{ None }
```

Listing 3.2: given OptionToExpr

The `ToExpr` for primitive types must be implemented as primitive operations in the system. In our case, we use the reflection API to implement them.

Extracting values from quotes To be able to generate optimized code using the method `unrolledPowerCode` (Listing 3.1), the macro implementation `powerCode` needs to first determine whether the argument passed as parameter `n` is a known constant value. This can be achieved via *unlifting* using the `Expr.unapply` extractor from our library implementation, which will only match if `n` is a quoted constant and extracts its value.

```
def powerCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
  n match
    case Expr(m) => // it is a constant: unlift code n='{m} into number m
      unrolledPowerCode(x, m)
    case _ => // not known: call power at run-time
      '{ power($x, $n) }
```

Listing 3.3: def powerCode

Alternatively, the `n.value` method can be used to get an `Option[Int]` with the value or `n.valueOrElseAbort` to get the value directly.

```
def powerCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
  // emits an error message if `n` is not a constant
  unrolledPowerCode(x, n.valueOrElseAbort)
```

`Expr.unapply` and all variants of `value` are polymorphic and user-extensible via a given `FromExpr` type class.

```
trait FromExpr[T]:
  def unapply(x: Expr[T])(using Quotes): Option[T]
```

We can use given definitions to implement the `FromExpr` as we did for `ToExpr`. The `FromExpr`

Macro and Run-Time Multi-Stage Programming

for primitive types must be implemented as primitive operations in the system. In our case, we use the reflection API to implement them. To implement `FromExpr` for non-primitive types we use quote pattern matching (Listing 3.7 of Section 3.1.8).

3.1.3 Macros and Multi-Stage Programming

The system supports multi-stage macros and run-time multi-stage programming using the same quotation abstractions.

Multi-Stage Macros

Macros We can generalize the splicing abstraction to express macros. A macro consists of a top-level splice that is not nested in any quote. Conceptually, the contents of the splice are evaluated one stage earlier than the program. In other words, the contents are evaluated while compiling the program. The generated code resulting from the macro replaces the splice in the program.

```
def power2(x: Double): Double =  
  ${ unrolledPowerCode('x, 2) } // x * x
```

Inline macros Since using the splices in the middle of a program is not as ergonomic as calling a function; we hide the staging mechanism from end-users of macros. We have a uniform way of calling macros and normal functions. For this, *we restrict the use of top-level splices to only appear in inline methods* (Chapter 2, [72]).

```
// inline macro definition  
inline def powerMacro(x: Double, inline n: Int): Double =  
  ${ powerCode('x, 'n) }  
  
// user code  
def power2(x: Double): Double =  
  powerMacro(x, 2) // x * x
```

Listing 3.4: `def powerMacro`

The evaluation of the macro will only happen when the code is inlined into `power2`. When inlined, the code is equivalent to the previous definition of `power2`. A consequence of using inline methods is that none of the arguments nor the return type of the macro will have to mention the `Expr` types; this hides all aspects of metaprogramming from the end-users.

3.1 Macros and Run-Time Multi-Stage Programming

Avoiding a complete interpreter When evaluating a top-level splice, the compiler needs to interpret the code that is within the splice. Providing an interpreter for the entire language is quite tricky, and it is even more challenging to make that interpreter run efficiently. To avoid needing a complete interpreter, we can impose the following restrictions on splices to simplify the evaluation of the code in top-level splices.

- The top-level splice must contain a single call to a compiled static method.
- Arguments to the function are literal constants, quoted expressions (parameters), calls to `Type.of` for type parameters and a reference to `Quotes`.

In particular, these restrictions disallow the use of splices in top-level splices. Such a splice would require several stages of interpretation which would be unnecessarily inefficient.

Compilation stages The macro implementation (i.e., the method called in the top-level splice) can come from any pre-compiled library. This provides a clear difference between the stages of the compilation process. Consider the following 3 source files defined in distinct libraries.

```
// Macro.scala
def powerCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] = ...
inline def powerMacro(x: Double, inline n: Int): Double =
  ${ powerCode('x, 'n) }
```

```
// Lib.scala (depends on Macro.scala)
def power2(x: Double) =
  ${ powerCode('x, '{2}) } // inlined from a call to: powerMacro(x, 2)
```

```
// App.scala (depends on Lib.scala)
@main def app() = power2(3.14)
```

One way to syntactically visualize this is to put the application in a quote that delays the compilation of the application. Then the application dependencies can be placed in an outer quote that contains the quoted application, and we repeat this recursively for dependencies of dependencies.

```
'{ // macro library (compilation stage 1)
  def powerCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
    ...
  inline def powerMacro(x: Double, inline n: Int): Double =
    ${ powerCode('x, 'n) }
  '{ // library using macros (compilation stage 2)
    def power2(x: Double) =
      ${ powerCode('x, '{2}) } // inlined from a call to: powerMacro(x, 2)
    '{ power2(3.14) /* app (compilation stage 3) */ }
  }
}
```

Macro and Run-Time Multi-Stage Programming

To make the system more versatile, we allow calling macros in the project where it is defined, with some restrictions. For example, to compile `Macro.scala` and `Lib.scala` together in the same library. To this end, we do not follow the simpler syntactic model and rely on semantic information from the source files. When compiling a source, if we detect a call to a macro that is not compiled yet, we delay the compilation of this source to the following compilation stage. In the example, we would delay the compilation of `Lib.scala` because it contains a compile-time call to `powerCode`. Compilation stages are repeated until all sources are compiled, or no progress can be made. If no progress is made, there was a cyclic dependency between the definition and the use of the macro. We also need to detect if at runtime the macro depends on sources that have not been compiled yet. These are detected by executing the macro and checking for JVM linking errors to classes that have not been compiled yet.

Run-Time Multi-Stage Programming

Traditional multi-stage programming supports run-time code generation using the `run` operation to evaluate a quoted expression into its value. We extend the system in the `staging`¹ library, which defines the aforementioned run operation and a way to create and manage compiler instances.

```
package scala.quoted.staging
def run[T](expr: Quotes => Expr[T])(using Compiler): T = ...
```

Listing 3.5: `scala.quoted.staging.run`

The `run` method will provide a fresh instance of the `Quotes` context. The run operation also requires an implicit instance of a `Compiler`, which will be used to compile the code of the expression. Users can choose how to manage the compiler instances depending on their needs.

```
given Compiler = Compiler.make(getClass.getClassLoader)

val power2: Double => Int = run {
  '{ (x: Double) => ${ unrolledPowerCode('x, 2) } }
}
```

In this example, we first evaluate the argument of the `run` to `'{ (x: Double) => x * x }`. Then `run` will compile the content of the quote and evaluate it to produce a lambda implemented as `(x: Double) => x * x`.

¹Implemented in the Scala 3 Dotty project <https://github.com/lampepfl/dotty>. sbt library dependency `"org.scala-lang" %% "scala3-staging" % scalaVersion.value`

3.1.4 Safety

Multi-stage programming is by design statically safe and cross-stage safe.

Static Safety

Hygiene All identifier names are interpreted as symbolic references to the corresponding variable in the context of the quote. Therefore, while evaluating the quote, it is not possible to accidentally rebind a reference to a new variable with the same textual name.

Well-typed If a quote is well typed, then the generated code is well typed. This is a simple consequence of tracking the type of each expression. An `Expr [T]` can only be created from a quote that contains an expression of type `T`. Conversely, an `Expr [T]` can only be spliced in a location that expects a type `T`. As mentioned before, `Expr` is covariant in its type parameter. This means that an `Expr [T]` can contain an expression of a subtype of `T`. When spliced in a location that expects a type `T`, these expressions also have a valid type.

Cross-Stage Safety

Level consistency We define the *staging level* of some code as the number of quotes minus the number of splices surrounding said code. Local variables must be defined and used in the same staging level.

It is never possible to access a local variable from a lower staging level as it does not yet exist.

```
def badPower(x: Double, n: Int): Double =
  ${ unrolledPowerCode('x, n) } // error: value of `n` not known yet
```

In the context of macros and *cross-platform portability*, that is, macros compiled on one machine but potentially executed on another, we cannot support cross-stage persistence of local variables. Therefore, local variables can only be accessed at precisely the same staging level in our system.

```
def badPowerCode(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
  // error: `n` potentially not available in the next execution environment
  '{ power($x, n) }
```

The rules are slightly different for global definitions, such as `unrolledPowerCode`. It is possible to generate code that contains a reference to a *global* definition such as in `'{ power(2, 4) }`. This is a limited form of cross-stage persistence that does not impede cross-platform portability, where we refer to the already compiled code for `power`. Each compilation step will lower the staging level by one while keeping global definitions. In consequence, we can refer to compiled definitions in macros such as `unrolledPowerCode` in `${ unrolledPowerCode('x, 2) }`.

Macro and Run-Time Multi-Stage Programming

We can summarize level consistency in two rules:

- Local variables can be used only at the same staging level as their definition
- Global variables can be used at any staging level

Type consistency As Scala uses type erasure, generic types will be erased at run-time and hence in any following stage. To ensure any quoted expression that refers to a generic type `T` does not lose the information it needs, we require a given `Type [T]` in scope. The `Type [T]` will carry over the non-erased representation of the type into the next phase. Therefore any generic type used at a higher staging level than its definition will require its `Type`.

Scope extrusion Within the contents of a splice, it is possible to have a quote that refers to a local variable defined in the outer quote. If this quote is used within the splice, the variable will be in scope. However, if the quote is somehow *extruded* outside the splice, then variables might not be in scope anymore. Quoted expressions can be extruded using side effects such as mutable state and exceptions. The following example shows how a quote can be extruded using mutable state.

```
var x: Expr[T] = null
' { (y: T) => $ { x = 'y; 1 } }
x // has value '{y}' but y is not in scope
```

A second way a variable can be extruded is through the `run` method. If `run` consumes a quoted variable reference, it will not be in scope anymore. The result will reference a variable that is defined in the next stage.

```
' { (x: Int) => $ { run('x); ... } }
// evaluates to: ' { (x: Int) => $ { x; ... } }
```

To catch both scope extrusion scenarios, our system restricts the use of quotes by only allowing a quote to be spliced if it was not extruded from a splice scope. Unlike level consistency, this is checked at run-time² rather than compile-time to avoid making the static type system too complicated.

Each `Quotes` instance contains a unique scope identifier and refers to its parent scope, forming a stack of identifiers. The parent of the scope of a `Quotes` is the scope of the `Quotes` used to create the enclosing quote. Top-level splices and `run` create new scope stacks. Every `Expr` knows in which scope it was created. When it is spliced, we check that the quote scope is either the same as the splice scope, or a parent scope thereof.

²Using the `-Xcheck-macros` compiler flag

3.1.5 Staged Lambdas

When staging programs in a functional language there are two fundamental abstractions: a staged lambda `Expr [T => U]` and a staging lambda `Expr [T] => Expr [U]`. The first is a function that will exist in the next stage, whereas the second is a function that exists in the current stage. It is often convenient to have a mechanism to go from `Expr [T => U]` to `Expr [T] => Expr [U]` and vice versa.

```
def later[T: Type, U: Type](f: Expr[T] => Expr[U]): Expr[T => U] =
  '{ (x: T) => ${ f('x) } }

def now[T: Type, U: Type](f: Expr[T => U]): Expr[T] => Expr[U] =
  (x: Expr[T]) => '{ ${f($x)} }
```

Both conversions can be performed out of the box with quotes and splices. But if `f` is a known lambda function, `'{ ${f($x)} }` will not β -reduce the lambda in place. This optimization is performed in a later phase of the compiler. Not reducing the application immediately can simplify analysis of generated code. Nevertheless, it is possible to β -reduce the lambda in place using the `Expr.betaReduce` method.

```
def now[T: Type, U: Type](f: Expr[T => U]): Expr[T] => Expr[U] =
  (x: Expr[T]) => Expr.betaReduce('{ ${f($x)} })
```

The `betaReduce` method will β -reduce the outermost application of the expression if possible (regardless of arity). If it is not possible to β -reduce the expression, then it will return the original expression.

3.1.6 Staged Constructors

To create new class instances in a later stage, we can create them using factory methods (usually apply methods of an object), or we can instantiate them with a `new`. For example, we can write `Some(1)` or `new Some(1)`, creating the same value. In Scala 3, using the factory method call notation will fall back to a `new` if no `apply` method is found. We follow the usual staging rules when calling a factory method. Similarly, when we use a `new C`, the constructor of `C` is implicitly called, which also follows the usual staging rules. Therefore for an arbitrary known class `C`, we can use both `'{ C(...) }` or `'{ new C(...) }` as constructors.

3.1.7 Staged Classes

Quoted code can contain any valid expression including local class definitions. This allows the creation of new classes with specialized implementations. For example, we can implement a new version of `Runnable` that will perform some optimized operation.

```
def mkRunnable(x: Int)(using Quotes): Expr[Runnable] = '{
  class MyRunnable extends Runnable:
    def run(): Unit = ... // generate some custom code that uses `x`
  new MyRunnable
}
```

The quoted class is a local class and its type cannot escape the enclosing quote. The class must be used inside the quote or an instance of it can be returned using a known interface (`Runnable` in this case).

3.1.8 Quote Pattern Matching

It is sometimes necessary to analyze the structure of the code or decompose the code into its sub-expressions. A classic example is an embedded DSL, where a macro knows a set of definitions that it can reinterpret while compiling the code (for instance, to perform optimizations). In the following example, we extend our previous implementation of `powCode` to look into `x` to perform further optimizations.

```
def fusedPowCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
  x match
  case '{ power($y, $m) } => // we have (y^m)^n
    fusedPowCode(y, '{ $n * $m }) // generate code for y^(n*m)
  case _ =>
    '{ power($x, $n) }
```

Listing 3.6: def fusedPowCode

Sub-patterns In quoted patterns, the `$` binds the sub-expression to an expression `Expr` that can be used in that case branch. The contents of `${..}` in a quote pattern are regular Scala patterns. For example, we can use the `Expr(_)` pattern within the `${..}` to only match if it is a known value and extract it.

```
def fusedUnrolledPowCode(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
  x match
  case '{ power($y, ${Expr(m)}) } => // we have (y^m)^n
    fusedUnrolledPowCode(y, n * m) // generate code for y * ... * y
  case _ => // ( n*m times )
    unrolledPowerCode(x, n)
```

3.1 Macros and Run-Time Multi-Stage Programming

These value extraction sub-patterns can be polymorphic using an instance of `FromExpr`. In the following example, we show the implementation of `OptionFromExpr` which internally uses the `FromExpr [T]` to extract the value using the `Expr(x)` pattern.

```
given OptionFromExpr[T](using Type[T], FromExpr[T]): FromExpr[Option[T]] with
  def unapply(x: Expr[Option[T]])(using Quotes): Option[Option[T]] =
    x match
      case '{ Some( ${Expr(x)} ) } => Some(Some(x))
      case '{ None } => Some(None)
      case _ => None
```

Listing 3.7: given OptionFromExpr

Closed patterns Patterns may contain two kinds of references: global references such as the call to the power method in `'{ power(. . .) }`, or references to bindings defined in the pattern such as `x` in `case '{ (x: Int) => x }`. When extracting an expression from a quote, we need to ensure that we do not extrude any variable from the scope where it is defined.

```
'{ (x: Int) => x + 1 } match
  case '{ (y: Int) => $z } =>
    // should not match, otherwise: z = '{ x + 1 }
```

In this example, we see that the pattern should not match. Otherwise, any use of the expression `z` would contain an unbound reference to `x`. To avoid any such extrusion, we only match on a ``${. .}`` if its expression is closed under the definitions within the pattern. Therefore, the pattern will not match if the expression is not closed.

HOAS patterns To allow extracting expressions that may contain extruded references we offer a *higher-order abstract syntax* (HOAS) [60] pattern `$f(y)` (or `$f(y1, . . . , yn)`). This pattern will η -expand the sub-expression with respect to `y` and bind it to `f`. The lambda arguments will replace the variables that might have been extruded.

```
'{ ((x: Int) => x + 1).apply(2) } match
  case '{ ((y: Int) => $f(y)).apply($z: Int) } =>
    // f may contain references to `x` (replaced by `${y}`)
    // f = (y: Expr[Int]) => '{ $y + 1 }
    f(z) // generates '{ 2 + 1 }
```

Listing 3.8: HOAS pattern

A HOAS pattern `$x(y1, . . . , yn)` will only match the expression if it does not contain references to variables defined in the pattern that are not in the set `y1, . . . , yn`. In other words, the pattern will match if the expression only contains references to variables defined in the pattern that are in `y1, . . . , yn`. Note that the HOAS patterns `$x()` are semantically equivalent

to closed patterns $\$x$.

This approach was also used in earlier Squid [56] versions. The use of HOAS allows us to keep the involved types simple. The η -expanded sub-expression can be typed with a simple function type. This way, we can avoid scope extrusion without resorting to complex type-level machinery of tracking free variables. HOAS patterns without parameters are considered closed patterns.

Type variables Expressions may contain types that are not statically known. For example, an `Expr [List [Int]]` may contain `list.map(_ .toInt)` where `list` is a `List` of some type. To cover all the possible cases we would need to explicitly match `list` on all possible types (`List [Int]`, `List [Int => Int]`, ...). This is an infinite set of types and therefore pattern cases. Even if we would know all possible types that a specific program could use, we may still end up with an unmanageable number of cases. To overcome this, we introduce type variables in quoted patterns, which will match any type.

In the following example, we show how type variables `t` and `u` match all possible pairs of consecutive calls to `map` on lists. In the quoted patterns, types named with lower cases are identified as type variables. This follows the same notation as type variables used in normal patterns.

```
def fuseMapCode(x: Expr[List[Int]]): Expr[List[Int]] =
  x match
    case '{ ($ls: List[t]).map[u]($f).map[Int]($g) } =>
      '{ $ls.map($g.compose($f)) }
    ...

fuseMapCode('{ List(1.2).map(f).map(g) }) // '{ List(1.2).map(g.compose(f)) }
fuseMapCode('{ List('a').map(h).map(i) }) // '{ List('a').map(i.compose(h)) }
```

Listing 3.9: `def fuseMapCode`

Variables `f` and `g` are inferred to be of type `Expr [t => u]` and `Expr [u => Int]` respectively. Subsequently, we can infer `$g.compose($f)` to be of type `Expr [t => Int]` which is the type of the argument of `$ls.map(...)`.

Type variables are abstract types that will be erased; this implies that to reference them in the second quote we need a given `Type [t]` and `Type [u]`. The quoted pattern will implicitly provide those given types. At run-time, when the pattern matches, the type of `t` and `u` will be known, and the `Type [t]` and `Type [u]` will contain the precise types in the expression.

As `Expr` is covariant, the statically known type of the expression might not be the actual type. Type variables can also be used to recover the precise type of the expression.

3.1 Macros and Run-Time Multi-Stage Programming

```
def let(x: Expr[Any])(using Quotes): Expr[Any] =
  x match
    case '{ $x: t } =>
      '{ val y: t = $x; y }

let('{1}) // will return a `Expr[Any]` that contains an `Expr[Int]`
```

Listing 3.10: def let

While we can define the type variable in the middle of the pattern, their normal form is to define them as a type with a lower case name at the start of the pattern. We use the Scala backquote ``t`` naming convention which interprets the string within the backquote as a literal name identifier. This is typically used when we have names that contain special characters that are not allowed for normal Scala identifiers. But we use it to explicitly state that this is a reference to that name and not the introduction of a new variable.

```
case '{ type t; $x: `t` } =>
```

This is a bit more verbose but has some expressivity advantages such as allowing to define bounds on the variables and be able to refer to them several times in any scope of the pattern.

```
case '{ type t >: List[Int] <: Seq[Int]; $x: `t` } =>
case '{ type t; $x: (`t`, `t`) } =>
```

Type patterns It is possible to only have a type and no expression of that type. To be able to inspect a type, we introduce quoted type pattern case `'[. .] =>`. It works the same way as a quoted pattern but is restricted to contain a type. Type variables can be used in quoted type patterns to extract a type.

```
def empty[T: Type]: Expr[T] =
  Type.of[T] match
    case '[String] => '{ "" }
    case '[List[t]] => '{ List.empty[t] }
    ...
```

Listing 3.11: def empty [T]

`Type.of [T]` is used to summon the given instance of `Type [T]` in scope, it is equivalent to `summon [Type [T]]`.

Type testing and casting It is important to note that instance checks and casts on `Expr`, such as `isInstanceOf [Expr [T]]` and `asInstanceOf [Expr [T]]`, will only check if the instance is of the class `Expr` but will not be able to check the `T` argument. These cases will issue a warning at compile-time, but if they are ignored, they can result in unexpected behavior.

These operations can be supported correctly in the system. For a simple type test it is possible to use the `isExprOf [T]` method of `Expr` to check if it is an instance of that type. Similarly, it is possible to use `asExprOf [T]` to cast an expression to a given type. These operations use a given `Type [T]` to work around type erasure.

3.1.9 Sub-Expression Transformation

The system provides a mechanism to transform all sub-expressions of an expression. This is useful when the sub-expressions we want to transform are deep in the expression. It is also necessary if the expression contains sub-expressions that cannot be matched using quoted patterns (such as local class definitions).

```
trait ExprMap:  
  def transform[T](e: Expr [T])(using Type [T])(using Quotes): Expr [T]  
  def transformChildren[T](e: Expr [T])(using Type [T])(using Quotes): Expr [T] =  
    ...
```

Listing 3.12: trait `ExprMap`

Users can extend the `ExprMap` trait and implement the `transform` method. This interface is flexible and can implement top-down, bottom-up, or other transformations.

```
object OptimizeIdentity extends ExprMap:  
  def transform[T](e: Expr [T])(using Type [T])(using Quotes): Expr [T] =  
    transformChildren(e) match // bottom-up transformation  
      case '{ identity($x) } => x  
      case _ => e
```

The `transformChildren` method is implemented as a primitive that knows how to reach all the direct sub-expressions and calls `transform` on each one. The type passed to `transform` is the expected type of this sub-expression in its expression. For example while transforming `Some(1)` in `'{ val x: Option[Int] = Some(1); ... }` the type will be `Option[Int]` and not `Some[Int]`. This implies that we can safely transform `Some(1)` into `None`.

3.1.10 Staged Implicit Summoning

When summoning implicit arguments using `summon`, we will find the given instances in the current scope. It is possible to use `summon` to get staged implicit arguments by explicitly staging them first. In the following example, we can pass an implicit `Ordering[T]` in a macro as an `Expr[Ordering[T]]` to its implementation. Then we can splice it and give it implicitly in the next stage.

```
inline def treeSetFor[T](using ord: Ordering[T]): Set[T] =
  ${ setExpr[T](using 'ord) }

def setExpr[T:Type](using ord: Expr[Ordering[T]])(using Quotes): Expr[Set[T]] =
  '{ given Ordering[T] = $ord; new TreeSet[T]() }
```

Listing 3.13: def treeSetFor

We pass it as an implicit `Expr[Ordering[T]]` because there might be intermediate methods that can pass it along implicitly.

An alternative is to summon implicit values in the scope where the macro is invoked. Using the `Expr.summon` method we get an optional expression containing the implicit instance. This provides the ability to search for implicit instances conditionally.

```
def summon[T: Type](using Quotes): Option[Expr[T]]
```

```
inline def setFor[T]: Set[T] =
  ${ setForExpr[T] }

def setForExpr[T: Type]() (using Quotes): Expr[Set[T]] =
  Expr.summon[Ordering[T]] match
  case Some(ord) =>
    '{ new TreeSet[T]()($ord) }
  case _ =>
    '{ new HashSet[T] }
```

Listing 3.14: def setFor

3.2 Implementation

This metaprogramming system was implemented for the Dotty Scala 3 compiler [21]. In this section we will cover details about the implementation such as syntax in Section 3.2.1, the run-time encoding in Section 3.2.2, entry points (macros and run-time staging) in Section 3.2.3 and compilation in Section 3.2.4.

3.2.1 Syntax

The quotation syntax using `'` and `$` was chosen to mimic the string interpolation syntax of Scala. Like a string double-quotation, a single-quote block can contain splices. However, unlike strings, splices can contain quotes using the same rules.

| | |
|--|--|
| <pre>s"Hello \$name" '{ hello(\$name) } \${ hello('name) }</pre> | <pre>s"Hello \${name}" '{ hello(\${name}) } \${ hello('name) }</pre> |
|--|--|

Quotes

Quotes come in four flavors: quoted identifiers, quoted blocks, quoted block patterns and quoted type patterns.

```
SimpleExpr ::= ...  
           | ` ` alphaid           // quoted identifier  
           | ` ` { Block } ` `     // quoted block  
Pattern    ::= ...  
           | ` ` { Block } ` `     // quoted block pattern  
           | ` ` [ Type ] ` `     // quoted type pattern
```

Listing 3.15: Quote syntax

Quoted blocks and quoted block patterns contain an expression equivalent to a normal block of code. When entering either of those we track the fact that we are in a quoted block (`inQuoteBlock`) which is used for spliced identifiers. When entering a quoted block pattern we additionally track the fact that we are in a quoted pattern (`inQuotePattern`) which is used to distinguish spliced blocks and splice patterns. Lastly, the quoted type pattern simply contains a type.

Splices

Splices come in three flavors: spliced identifiers, spliced blocks and splice patterns. Scala specifies identifiers containing \$ as valid identifiers but reserves them for compiler and standard library use only. Unfortunately, many libraries have used such identifiers in Scala 2. Therefore to mitigate the cost of migration, we still support them. We work around this by only allowing spliced identifiers³ within quoted blocks or quoted block patterns (`inQuoteBlock`). Splice blocks and splice patterns can contain an arbitrary block or pattern respectively. They are distinguished based on their surrounding quote (`inQuotePattern`), a quote block will contain spliced blocks, and a quote block pattern will contain splice patterns.

```
SimpleExpr ::= ...
            | `$$` alphaid           if inQuoteBlock // spliced identifier
            | `$$` `{` Block `}`     if !inQuotePattern // spliced block
            | `$$` `{` Pattern `}`   if inQuotePattern // splice pattern
```

Listing 3.16: Splice syntax

Quoted Pattern Type Variables

Quoted pattern type variables in quoted patterns and quoted type patterns do not require additional syntax. Any type definition or reference with a name composed of lower cases is assumed to be a pattern type variable definition while typing. A backticked type name with lower cases is interpreted as a reference to the type with that name.

3.2.2 Run-Time Representation

The standard library defines the `Quotes` interface which contains all the logic and the abstract classes `Expr` and `Type`. The compiler implements the `Quotes` interface and provides the implementation of `Expr` and `Type`.

`class Expr` Expressions of type `Expr [T]` are represented by the following abstract class:

```
abstract class Expr[+T] private[scala]
```

Listing 3.17: class Expr

The only implementation of `Expr` is in the compiler along with the implementation of `Quotes`. It is a class that wraps a typed AST and a `Scope` object with no methods of its own. The `Scope` object is used to track the current splice scope and detect scope extrusions.

³In quotes, identifiers starting with \$ must be surrounded by backticks. For example ``$conforms`` from `scala.Predef`.

Macro and Run-Time Multi-Stage Programming

object Expr The companion object of Expr contains a few useful static methods; the apply/unapply methods to use ToExpr/FromExpr with ease; the betaReduce and summon methods. It also contains methods to create expressions out of lists or sequences of expressions: block, ofSeq, ofList, ofTupleFromSeq and ofTuple.

```
object Expr:
  def apply[T](x: T)(using ToExpr[T])(using Quotes): Expr[T] = ...
  def unapply[T](x: Expr[T])(using FromExpr[T])(using Quotes): Option[T] = ...
  def betaReduce[T](e: Expr[T])(using Quotes): Expr[T] = ...
  def summon[T: Type](using Quotes): Option[Expr[T]] = ...
  def block[T](stats: List[Expr[Any]], e: Expr[T])(using Quotes): Expr[T] = ...
  def ofSeq[T: Type](xs: Seq[Expr[T]])(using Quotes): Expr[Seq[T]] = ...
  def ofList[T: Type](xs: Seq[Expr[T]])(using Quotes): Expr[List[T]] = ...
  def ofTupleFromSeq(xs: Seq[Expr[Any]])(using Quotes): Expr[Tuple] = ...
  def ofTuple[T <: Tuple: Tuple.IsMappedBy[Expr]: Type](tup: T)(using Quotes):
    Expr[Tuple.InverseMap[T, Expr]] = ...
```

Listing 3.18: object Expr

class Type Types of type Type[T] are represented by the following abstract class:

```
abstract class Type[T <: AnyKind] private[scala]:
  type Underlying = T
```

Listing 3.19: class Type

The only implementation of Type is in the compiler along with the implementation of Quotes. It is a class that wraps the AST of a type and a Scope object with no methods of its own. The upper bound of T is AnyKind which implies that T may be a higher-kinded type. The Underlying alias is used to select the type from an instance of Type. Users never need to use this alias as they can always use T directly. Underlying is used for internal encoding while compiling the code (see *Type Healing* of Section 3.2.4).

object Type The companion object of Type contains a few useful static methods. The first and most important one is the Type.of given definition. This instance of Type[T] is summoned by default when no other instance is available. The of operation is an intrinsic operation that the compiler will transform into code that will generate the Type[T] at runtime. Secondly, the Type.show[T] operation will show a string representation of the type, which is often useful when debugging. Finally, the object defines valueOfConstant (and valueOfTuple) which can transform singleton types (or tuples of singleton types) into their value.

```
object Type:
  given of[T <: AnyKind](using Quotes): Type[T] = ...
  def show[T <: AnyKind](using Type[T])(using Quotes): String = ...
  def valueOfConstant[T](using Type[T])(using Quotes): Option[T] = ...
  def valueOfTuple[T <: Tuple](using Type[T])(using Quotes): Option[T] = ...
```

Listing 3.20: object Type

Quotes The `Quotes` interface is where most of the primitive operations of the quotation system are defined.

Quotes define all the `Expr[T]` methods as extension methods. `Type[T]` does not have methods and therefore does not appear here. These methods are available as long as `Quotes` is implicitly given in the current scope.

The `Quotes` instance is also the entry point to the reflection API through the `reflect` object. We defer discussion of the `reflect` object to Part III.

Finally, `Quotes` provides the internal logic used in quote un-pickling (`QuoteUnpickler`) in quote pattern matching (`QuoteMatching`). These interfaces are added to the self-type of the trait to make sure they are implemented on this object but not visible to users of `Quotes`.

Internally, the implementation of `Quotes` will also track its current splicing scope `Scope`. This scope will be attached to any expression that is created using this `Quotes` instance.

```
trait Quotes:
  this: runtime.QuoteUnpickler & runtime.QuoteMatching =>

  extension [T](self: Expr[T])
    def show: String
    def matches(that: Expr[Any]): Boolean
    def value(using FromExpr[T]): Option[T]
    def valueOrAbort(using FromExpr[T]): T
  end extension

  extension (self: Expr[Any])
    def isExprOf[X](using Type[X]): Boolean
    def asExprOf[X](using Type[X]): Expr[X]
  end extension

  // abstract object reflect ...
```

Listing 3.21: trait Quotes

Scope The splice context is represented as a stack (immutable list) of `Scope` objects. Each `Scope` contains the position of the splice (used for error reporting) and a reference to the enclosing splice scope `Scope`. A scope is a sub-scope of another if the other is contained in its parents. This check is performed when an expression is spliced into another using the `Scope` provided in the current scope in `Quotes` and the one in the `Expr` or `Type`.

3.2.3 Entry Points

The two entry points for multi-stage programming are macros and the `run` operation.

Macros

Inline macro definitions will inline a top-level splice (a splice not nested in a quote). This splice needs to be evaluated at compile-time. In *Avoiding a complete interpreter* of Section 3.1.3, we stated the following restrictions:

- The top-level splice must contain a single call to a compiled static method.
- Arguments to the function are either literal constants, quoted expressions (parameters), `Type.of` for type parameters and a reference to `Quotes`.

These restrictions make the implementation of the interpreter quite simple. Java Reflection is used to call the single function call in the top-level splice. The execution of that function is entirely done on compiled bytecode. These are Scala static methods and may not always become Java static methods, they might be inside module objects. As modules are encoded as class instances, we need to interpret the prefix of the method to instantiate it before we can invoke the method.

The code of the arguments has not been compiled and therefore needs to be interpreted by the compiler. Interpreting literal constants is as simple as extracting the constant from the AST that represents literals. When interpreting a quoted expression, the contents of the quote is kept as an AST which is wrapped inside the implementation of `Expr`. Calls to `Type.of [T]` also wrap the AST of the type inside the implementation of `Type`. Finally, the reference to `Quotes` is supposed to be the reference to the quotes provided by the splice. This reference is interpreted as a new instance of `Quotes` that contains a fresh initial `Scope` with no parents.

The result of calling the method via Java Reflection will return an `Expr` containing a new AST that was generated by the implementation of that macro. The scope of this `Expr` is checked to make sure it did not extrude from some splice or `run` operation. Then the AST is extracted from the `Expr` and it is inserted as replacement for the AST that contained the top-level splice.

Run-time Multi-Stage Programming

To be able to compile the code, the `scala.quoted.staging` library defines the `Compiler` trait. An instance of `staging.Compiler` is a wrapper over the normal Scala 3 compiler. To be instantiated it requires an instance of the JVM *classloader* of the application.

```
import scala.quoted.staging.*
given Compiler = Compiler.make(getClass.getClassLoader)
```

Listing 3.22: `scala.quoted.staging.Compiler`

The classloader is needed for the compiler to know which dependencies have been loaded and to load the generated code using the same classloader.

```
def mkPower2()(using Quotes): Expr[Double => Double] = ...

run(mkPower2())
```

To run the previous example, the compiler will create code equivalent to the following class and compile it using a new `Scope` without parents.

```
class RunInstance:
  def exec(): Double => Double = ${ mkPower2() }
```

Finally, `run` will interpret `(new RunInstance).exec()` to evaluate the contents of the quote. To do this, the resulting `RunInstance` class is loaded in the JVM using Java Reflection, instantiated and then the `exec` method is invoked.

3.2.4 Compilation

Quotes and splices are primitive forms in the generated typed abstract syntax trees. These need to be type-checked with some extra rules, e.g., staging levels need to be checked and the references to generic types need to be adapted. Finally, quoted expressions that will be generated at run-time need to be encoded (serialized) and decoded (deserialized).

Typing Quoted Expressions

The typing process for quoted expressions and splices with `Expr` is relatively straightforward. At its core, quotes are desugared into calls to `quote`, splices are desugared into calls to `splice`. We track the quotation level when desugaring into these methods.

```
def quote[T](x: T): Quotes ?=> Expr[T]

def splice[T](x: Quotes ?=> Expr[T]): T
```

Macro and Run-Time Multi-Stage Programming

It would be impossible to track the quotation levels if users wrote calls to these methods directly. To know if it is a call to one of those methods we would need to type it first, but to type it we would need to know if it is one of these methods to update the quotation level. Therefore these methods can only be used by the compiler.

At run-time, the splice needs to have a reference to the `Quotes` that created its surrounding quote. To simplify this for later phases, we track the current `Quotes` and encode a reference directly in the splice using `nestedSplice` instead of `splice`.

```
def nestedSplice[T](q: Quotes)(x: q.Nested ?=> Expr[T]): T
```

With this addition, the original `splice` is only used for top-level splices.

The levels are mostly used to identify top-level splices that need to be evaluated while typing. We do not use the quotation level to influence the typing process. Level checking is performed at a later phase. This ensures that a source expression in a quote will have the same elaboration as a source expression outside the quote.

Quote Pattern Matching

Pattern matching is defined in the trait `QuoteMatching`, which is part of the self type of `Quotes`. It is implemented by `Quotes` but not available to users of `Quotes`. To access it, the compiler generates a cast from `Quotes` to `QuoteMatching` and then selects one of its two members: `ExprMatch` or `TypeMatch`. `ExprMatch` defines an `unapply` extractor method that is used to encode quote patterns and `TypeMatch` defines an `unapply` method for quoted type patterns.

```
trait Quotes:
  self: runtime.QuoteMatching & ... =>
  ...

trait QuoteMatching:
  object ExprMatch:
    def unapply[TypeBindings <: Tuple, Tup <: Tuple]
      (scrutinee: Expr[Any])
      (using pattern: Expr[Any]): Option[Tup] = ...
  object TypeMatch:
    ...
```

These extractor methods are only meant to be used in code generated by the compiler. The call to the extractor that is generated has an already elaborated form that cannot be written in source, namely explicit type parameters and explicit contextual parameters.

This extractor returns a tuple type `Tup` which cannot be inferred from the types in the method

signature. This type will be computed when typing the quote pattern and will be explicitly added to the extractor call. To refer to type variables in arbitrary places of `Tup`, we need to define them all before their use, hence we have `TypeBindings`, which will contain all pattern type variable definitions. The extractor also receives a given parameter of type `Expr [Any]` that will contain an expression that represents the pattern. The compiler will explicitly add this pattern expression. We use a given parameter because these are the only parameters we are allowed to add to the extractor call in a pattern position.

This extractor is a bit convoluted, but it encodes away all the quotation-specific features. It compiles the pattern down into a representation that the pattern matcher compiler phase understands.

The quote patterns are encoded into two parts: a tuple pattern that is tasked with extracting the result of the match and a quoted expression representing the pattern. For example, if the pattern has no `$` we will have an `EmptyTuple` as the pattern and `'{1}` to represent the pattern.

```

case '{ 1 } =>
// is elaborated to
case ExprMatch(EmptyTuple)(using '{1}) =>
//           ~~~~~ ~~~~~
//           pattern  expression

```

When extracting expressions, each pattern that is contained in a splice `#{ . . }` will be placed in order in the tuple pattern. In the following case, the `f` and `x` are placed in a tuple pattern `(f, x)`. The type of the tuple is encoded in the `Tup` and not only in the tuple itself. Otherwise, the extractor would return a tuple `Tuple` for which the types need to be tested which is in turn not possible due to type erasure.

```

case '{ ((y: Int) => $f(y)).apply($x) } =>
// is elaborated to
case ExprMatch[. . , (Expr[Int => Int], Expr[Int])](f, x)(using pattern) =>
// pattern = '{ ((y: Int) => pat[Int](y)).apply(pat[Int]()) }

```

The contents of the quote are transformed into a valid quote expression by replacing the splice with a marker expression `pat [T] (. .)`. The type `T` is taken from the type of the splice and the arguments are the HOAS arguments. This implies that a `pat [T] ()` is a closed pattern and `pat [T] (y)` is an HOAS pattern that can refer to `y`.

Type variables in quoted patterns are first normalized to have all definitions at the start of the pattern. For each definition of a type variable `t` in the pattern we will add a type variable definition in `TypeBindings`. Each one will have a corresponding `Type [t]` that will get extracted if the pattern matches. These `Type [t]` are also listed in the `Tup` and added in the tuple pattern. It is additionally marked as `using` in the pattern to make it implicitly available in this case branch.

Macro and Run-Time Multi-Stage Programming

```
case '{ type t; ($xs: List[t]).map[t](identity[t]) } =>
// is elaborated to
case ExprMatch[(t), (Type[t], Expr[List[t]])]((using t, xs))(using p) =>
//      ~~~ ~~~~~
//      type bindings      result type      pattern      expression
// p = '{ @patternType type u; pat[List[u]]().map[u](identity[u]) }
```

The contents of the quote are transformed into a valid quote expression by replacing type variables with fresh ones that do not escape the quote scope. These are also annotated to be easily identifiable as pattern variables.

Level Consistency Checking

Level consistency checking is performed after typing the program as a static check. To check level consistency we traverse the tree top-down remembering the context staging level. Each local definition in scope is recorded with its level and each term reference to a definition is checked against the current staging level.

```
// level 0
'{ // level 1
  val x = ... // level 1 with (x -> 1)
  ${ // level 0 (x -> 1)
    val y = ... // level 0 with (x -> 1, y -> 0)
    x // error: defined at level 1 but used in level 0
  }
  // level 1 (x -> 1)
  x // x is ok
}
```

Type Healing

When using a generic type T in a future stage, it is necessary to have a given $\text{Type}[T]$ in scope. The compiler needs to identify those references and link them with the instance of $\text{Type}[T]$. For instance consider the following example:

```
def emptyList[T](using t: Type[T])(using Quotes): Expr[List[T]] =
  '{ List.empty[T] }
```

Listing 3.23: def emptyList

For each reference to a generic type T that is defined at level 0 and used at level 1 or greater, the compiler will summon a $\text{Type}[T]$. This is usually the given type that is provided as parameter,

`t` in this case. We can use the type `t.Underlying` to replace `T` as it is an alias of that type. But `t.Underlying` contains the extra information that it is `t` that will be used in the evaluation of the quote. In a sense, `Underlying` acts like a splice for types.

```
def emptyList[T](using t: Type[T])(using Quotes): Expr[List[T]] =
  '{ List.empty[t.Underlying] }
```

Due to some technical limitations, it is not always possible to replace the type reference with the AST containing `t.Underlying`. To overcome this limitation, we can simply define a list of type aliases at the start of the quote and insert the `t.Underlying` there. This has the added advantage that we do not have to repeatedly insert the `t.Underlying` in the quote.

```
def emptyList[T](using t: Type[T])(using Quotes): Expr[List[T]] =
  '{ type U = t.Underlying; List.empty[U] }
```

These aliases can be used at any level within the quote and this transformation is only performed on quotes that are at level 0.

```
'{ List.empty[T] ... '{ List.empty[T] } ... }
// becomes
'{ type U = t.Underlying; List.empty[U] ... '{ List.empty[U] } ... }
```

If we define a generic type at level 1 or greater, it will not be subject to this transformation. In some future compilation stage, when the definition of the generic type is at level 0, it will be subject to this transformation. This simplifies the transformation logic and avoids leaking the encoding into code that a macro could inspect.

```
'{
  def emptyList[T: Type](using Quotes): Expr[List[T]] = '{ List.empty[T] }
  ...
}
```

A similar transformation is performed on `Type.of[T]`. Any generic type in `T` needs to have an implicitly given `Type[T]` in scope, which will also be used as a path. The example from Listing 3.11 would be transformed as follows:

```
def empty[T](using t: Type[T])(using Quotes): Expr[T] =
  Type.of[T] match ...
// becomes
def empty[T](using t: Type[T])(using Quotes): Expr[T] =
  Type.of[t.Underlying] match ...
// then becomes
def empty[T](using t: Type[T])(using Quotes): Expr[T] =
  t match ...
```

Macro and Run-Time Multi-Stage Programming

The operation `Type.of[t.Underlying]` can be optimized to just `t`. But this is not always the case. If the generic reference is nested in the type, we will need to keep the `Type.of`.

```
def matchOnList[T](using t: Type[T])(using Quotes): Expr[List[T]] =
  Type.of[List[T]] match ...
// becomes
def matchOnList[T](using t: Type[T])(using Quotes): Expr[List[T]] =
  Type.of[List[t.Underlying]] match ...
```

By doing this transformation, we ensure that each abstract type `U` used in `Type.of` has an implicit `Type[U]` in scope. This representation makes it simpler to identify parts of the type that are statically known from those that are known dynamically. Type aliases are also added within the type of the `Type.of` though these are not valid source code. These would look like `Type.of[{type U = t.Underlying; Map[U, U]}]` if written in source code.

Splice Normalization

The contents of a splice may refer to variables defined in the enclosing quote. This complicates the process of serialization of the contents of the quotes. To make serialization simple, we first transform the contents of each level 1 splice. Consider the following example:

```
def power5to(n: Expr[Int]): Expr[Double] = '{
  val x: Int = 5
  ${ powerCode('{x}, n) }
}
```

The variable `x` is defined in the quote and used in the splice. The normal form will extract all references to `x` and replace them with a staged version of `x`. We will replace the reference to `x` of type `T` with a `$y` where `y` is of type `Expr[T]`. Then we wrap the new contents of the splice in a lambda that defines `y` and apply it to the quoted version of `x`. After this transformation we have 2 parts, a lambda without references to the quote, which knows how to compute the contents of the splice, and a sequence of quoted arguments that refer to variables defined in the lambda.

```
def power5to(n: Expr[Int]): Expr[Double] = '{
  val x: Int = 5
  ${ ((y: Expr[Int]) => powerCode('{y}, n)).apply('x) }
}
```

In general, the splice normal form has the shape `$(<lambda>.apply(<args>*))` and the following constraints:

- `<lambda>` a lambda expression that does not refer to variables defined in the outer quote
- `<args>` sequence of quoted expressions or `Type.of` containing references to variables defined in the enclosing quote and no references to local variables defined outside the enclosing quote

Function references normalization A reference to a function `f` that receives parameters is not a valid value in Scala. Such a function reference `f` can be η -expanded as `x => f(x)` to be used as a lambda value. Therefore function references cannot be transformed by the normalization as directly as other expressions as we cannot represent `{f}` with a method reference type. We can use the η -expanded form of `f` in the normalized form. For example, consider the reference to `f` below.

```
'{
  def f(a: Int)(b: Int, c: Int): Int = 2 + a + b + c
  ${ '{ f(3)(4, 5) } }
}
```

To normalize this code, we can η -expand the reference to `f` and place it in a quote containing a proper expression. Therefore the normalized form of the argument `{f}` becomes the quoted lambda `{ (a: Int) => (b: Int, c: Int) => f(a)(b, c) }` and is an expression of type `Expr[Int => (Int, Int) => Int]`. The η -expansion produces one curried lambda per parameter list. The application `f(3)(4, 5)` does not become `$g(3)(4, 5)` but `$g.apply(3).apply(4, 5)`. We add the `apply` because `g` is not a quoted reference to a function but a curried lambda.

```
'{
  def f(a: Int)(b: Int, c: Int): Int = 2 + a + b + c
  ${
    (
      (g: Expr[Int => (Int, Int) => Int]) => '{$g.apply(3).apply(4, 5)}
    ).apply('{ (a: Int) => (b: Int, c: Int) => f(a)(b, c) })
  }
}
```

Then we can apply it and β -reduce the application when generating the code.

```
(g: Expr[Int => Int => Int]) => betaReduce('{$g.apply(3).apply(4)})
```

Macro and Run-Time Multi-Stage Programming

Variable assignment normalization A reference to a mutable variable in the left-hand side of an assignment cannot be transformed directly as it is not in an expression position.

```
'{  
  var x: Int = 5  
  ${ g('{x = 2}') }  
}
```

We can use the same strategy used for function references by η -expanding the assignment operation `x = _` into `y => x = y`.

```
'{  
  var x: Int = 5  
  ${  
    g(  
      (  
        (f: Expr[Int => Unit]) => betaReduce('${f(2)}')  
      ).apply('{ (y: Int) => x = $y }')  
    )  
  }  
}
```

Type normalization Types defined in the quote are subject to a similar transformation. In this example, `T` is defined within the quote at level 1 and used in the splice again at level 1.

```
'{ def f[T] = ${ '{g[T]}' } }
```

The normalization will add a `Type [T]` to the lambda, and we will insert this reference. The difference is that it will add an alias similar to the one used in type healing. In this example, we create a type `U` that aliases the staged type.

```
'{  
  def f[T] = ${  
    (  
      (t: Type[T]) => '{type U = t.Underling; g[U]}'  
    ).apply(Type.of[T])  
  }  
}
```

Serialization

Quoted code needs to be pickled [37] to make it available at run-time in the next compilation phase. We implement this by pickling the AST as a TASTy binary.

TASTy The TASTy format [51] is the typed abstract syntax tree serialization format of Scala 3. It usually pickles the fully elaborated code after type-checking and is kept along the generated Java classfiles. In the compiler, we use it for separate and incremental compilation, documentation generation, and code decompilation. We also use it for language servers in IDEs, and the metaprogramming API for quoted code.

Pickling We use TASTy as a serialization format for the contents of the quotes. To show how serialization is performed, we will use the following example.

```
'{
  val (x, n): (Double, Int) = (5, 2)
  ${ powerCode('{x}', '{n}') } * ${ powerCode('{2}', '{n}') }
}
```

This quote is transformed into the following code when normalizing the splices.

```
'{
  val (x, n): (Double, Int) = (5, 2)
  ${
    ((y: Expr[Double], m: Expr[Int]) => powerCode(y, m)).apply('x, 'n)
  } * ${
    ((m: Expr[Int]) => powerCode('{2}', m)).apply('n)
  }
}
```

Splice normalization is a key part of the serialization process as it only allows references to variables defined in the quote in the arguments of the lambda in the splice. This makes it possible to create a closed representation of the quote without much effort. The first step is to remove all the splices and replace them with holes. A hole is like a splice but it lacks the knowledge of how to compute the contents of the splice. Instead, it knows the index of the hole and the contents of the arguments of the splice. We can see this transformation in the following example where a hole is represented by `<< idx; holeType; args* >>`.

```
  ${ ((y: Expr[Double], m: Expr[Int]) => powerCode(y, m)).apply('x, 'n) }
// becomes
  << 0; Double; x, n >>
```

Macro and Run-Time Multi-Stage Programming

As this was the first hole it has index 0. The hole type is `Double`, which needs to be remembered now that we cannot infer it from the contents of the splice. The arguments of the splice are `x` and `n`; note that they do not require quoting because they were moved out of the splice.

References to healed types are handled in a similar way. Consider the `emptyList` example of Listing 3.23, which shows the type aliases that are inserted into the quote.

```
'{ List.empty[T] }  
// type healed to  
'{ type U = t.Underlying; List.empty[U] }
```

Instead of replacing a splice, we replace the `t.Underlying` type with a type hole. The type hole is represented by `<< idx; bounds >>`.

```
'{ type U = << 0; Nothing..Any >>; List.empty[U] }
```

Here, the bounds of `Nothing..Any` are the bounds of the original `T` type. The types of a `Type.of` are transformed in the same way.

With these transformations, the contents of the quote or `Type.of` are guaranteed to be closed and therefore can be pickled. The AST is pickled into `TASTy`, which is a sequence of bytes. This sequence of bytes needs to be instantiated in the bytecode, but unfortunately it cannot be dumped into the classfile as bytes. To reify it we encode the bytes into a Java `String`. In the following examples we display this encoding in human readable form with the fictitious `tasty"..."` string literal.

```
// pickled AST bytes encoded in a base64 string  
tasty""  
  val (x, n): (Double, Int) = (5, 2)  
  << 0; Double; x, n >> * << 1; Double; n >>  
""  
// or  
tasty""  
  type U = << 0; Nothing..Any; >>  
  List.empty[U]  
""
```

The contents of a quote or `Type.of` are not always pickled. In some cases it is better to generate equivalent (smaller and/or faster) code that will compute the expression. Literal values are compiled into a call to `Expr(<literal>)` using the implementation of `ToExpr` to create the quoted expression. This is currently performed only on literal values, but can be extended to any value for which we have a `ToExpr` defined in the standard library. Similarly, for non-generic types we can use their respective `java.lang.Class` and convert them into a `Type` using a primitive operation `typeConstructorOf` defined in the reflection API.

Unpickling Now that we have seen how a quote is pickled, we can look at how to unpickle it. We will continue with the previous example.

Holes were used to replace the splices in the quote. When we perform this transformation we also need to remember the lambdas from the splices and their hole index. When unpickling a hole, the corresponding splice lambda will be used to compute the contents of the hole. The lambda will receive as parameters quoted versions of the arguments of the hole. For example to compute the contents of `<< 0; Double; x, n >>` we will evaluate the following code

```
((y: Expr[Double], m: Expr[Int]) => powerCode(y, m)).apply('x, 'n)
```

The evaluation is not as trivial as it looks, because the lambda comes from compiled code and the rest is code that must be interpreted. We put the AST of `x` and `n` into `Expr` objects to simulate the quotes and then we use Java Reflection to call the `apply` method.

We may have many holes in a quote and therefore as many lambdas. To avoid the instantiation of many lambdas, we can join them together into a single lambda. Apart from the list of arguments, this lambda will also take the index of the hole that is being evaluated. It will perform a switch match on the index and call the corresponding lambda in each branch. Each branch will also extract the arguments depending on the definition of the lambda. The application of the original lambdas are β -reduced to avoid extra overhead.

```
(idx: Int, args: Seq[Any]) =>
  idx match
    case 0 => // for << 0; Double; x, n >>
      val x = args(0).asInstanceOf[Expr[Double]]
      val n = args(1).asInstanceOf[Expr[Int]]
      powerCode(x, n)
    case 1 => // for << 1; Double; n >>
      val n = args(0).asInstanceOf[Expr[Int]]
      powerCode('{2}, n)
```

This is similar to what we do for splices when we replace the type aliased with holes we keep track of the index of the hole. Instead of lambdas, we will have a list of references to instances of `Type`. From the following example we would extract `t, u, ...`

```
'{ type T1 = t1.Underlying; type Tn = tn.Underlying; ... }
// with holes
'{ type T1 = << 0; ... >>; type Tn = << n-1; ... >>; ... }
```

As the type holes are at the start of the quote, they will have the first `N` indices. This implies that we can place the references in a sequence `Seq(t, u, ...)` where the index in the sequence is the same as the hole index.

Macro and Run-Time Multi-Stage Programming

Lastly, the quote itself is replaced by a call to `QuoteUnpickler.unpickleExpr` which will unpickle the AST, evaluate the holes, i.e., splices, and wrap the resulting AST in an `Expr[Int]`. This method takes the pickled `tasty"..."`, the types and the hole lambda. Similarly, `Type.of` is replaced with a call to `QuoteUnpickler.unpickleType` but only receives the pickled `tasty"..."` and the types. Because `QuoteUnpickler` is part of the self-type of the `Quotes` class, we have to cast the instance but know that this cast will always succeed.

```
quotes.asInstanceOf[runtime.QuoteUnpickler].unpickleExpr[T] (
  pickled = tasty"...",
  types = Seq(...),
  holes = (idx: Int, args: Seq[Any]) => idx match ...
)
```

3.3 Reflection

Multi-stage programming offers a powerful way to generate and analyze programs while ensuring strong static safety guarantees. Providing this safety comes at the cost of expressiveness of the system. For the use cases where extra expressivity is needed we extend the system with a reflection API that allows inspection and creation of typed ASTs. This extension is more expressive because it moves the static guarantees to the run-time (which are still at compile-time for a macro). The design of the reflection API will be covered in Part III.

3.4 Related Work

Our system is heavily inspired by the long line of work by MetaML [82], MetaOCaml [13] and BER MetaOCaml [38]. We rely on the latter for most of our design decisions. We offer the capability of pretty-printing generated code, but our system, contrary to BER MetaOCaml, compiles to native code first. In our case, native code (JVM bytecode) was simpler to implement since we rely on TASTy, the serialization format for typed syntax trees of Scala programs [51]. BER MetaOCaml offers to programmers the capability to process code values in their own way.

Modular Macros [88] offered a compile-time variant of BER MetaOCaml by introducing a new keyword to enable macro expansion. In their work they demonstrate that an existing staged library needs intrusive changes to sprinkle the code with the aforementioned keywords. In our case we just need one definition with a top-level splice and we reuse a staged library unchanged. Modular Macros is a separate project to BER MetaOCaml so the two techniques were not composed.

MacroML [26] pioneered compile-time version of MetaML showing at a theoretical level that the semantics of MetaML subsume macros; MacroML essentially translates macro programs to MetaML programs. Our work presents a confluence of macros and multi-stage programming in the same language (considering the imperative features of Scala, something left out from

MacroML's development). Even though this merge was not demonstrated in the original work by Ganz, Sabry, and Taha [26] we believe that their work provides useful insights for the future foundations of our system.

Template Haskell [66] is a very expressive metaprogramming system that offers support for code generation not only of expressions but also definitions, instances and more. Template Haskell used the type class `lift` to perform cross-stage persistence. We used the same technique for our `Liftable` construct. Code generation in Template Haskell is essentially untyped; the generated code is not guaranteed to be well typed. Typed Template Haskell, on the other hand, also inspired by MetaML and MetaOCaml, offers a more restrictive view in order to pursue a disciplined system for code generation. Typed Template Haskell is still considered to be unsound under side effects [35], providing the same static guarantees as MetaOCaml. To avoid these shortcomings, we permit no side effects in splice operations either. We regard side effects as an important aspect of programming code generators. The decision to disallow effects in splices was taken because it was a simple approach to avoid the unsoundness hole of scope extrusion. At the moment, code generators and delimited control (e.g., like restricting the code generator's effects to the scope of generated binders [34]) was out of the scope of this work but remains a goal of our future work.

F# supports code quotations that offer a quoting mechanism that is not opaque to the user effectively, supporting analysis of F# expression trees at run-time. Programmers can quote expressions and they are offered the choice of getting back either a typed or an untyped expression tree. F# does not support multi-stage programming and currently lacks a code quotation compiler natively⁴. Furthermore, lifting is not supported. Finally, F# does not support splicing of types into quotations.

Scala 2 offers experimental macros (called blackbox in Scala parlance) [12; 11]. The provided macros are quite different from our approach. Those macros expose directly an abstraction of the compiler's ASTs and the current compilation context. Scala Macros require specialized knowledge of the compiler internals. Quasiquotes, additionally, are implemented on top of macros using string interpolators [64] which simplify code generation. However, the user is still exposed to the same complex machinery, inherited from them. Scala also offers macros that can modify existing types in the system (whitebox and annotation macros). They have proven dangerously powerful; they can arbitrarily affect typing in unconventional ways giving rise to problems that can deteriorate IDE support, compiler evolution and code understanding. Instead, we use `transparent inline` macros to be able to influence typing at call site in a controlled manner.

Lightweight Modular Staging (LMS) offers support for Multi-stage Programming in Scala [63]. LMS departs from the use of explicit staging annotations by adopting a *type-based embedding*. On the contrary, a design choice of our system is to offer explicit annotations along the lines of MetaML. We believe that programming with quotes and splices reflects the textual nature of

⁴Splice types into Quotations—<https://github.com/fsharp/fslang-suggestions/issues/584>

this kind of metaprogramming and gives the necessary visual feedback to the user, who needs to reason about code fragments. LMS is a powerful system that preserves the execution order of staged computations and also offers an extensible Graph-based IR. On the flip side, two shortcomings of LMS, namely high compile times and the fact that it is based on a fork of the compiler, were recently discussed as points of improvement [62].

Squid [58] advances the state of the art of staging systems and puts quasiquotes at the center of user-defined optimizations. The user can pattern match over existing code and implement retroactive optimizations modularly. A shortcoming in Squid, implemented as a macro library, is that free variables must be marked explicitly. Furthermore, contexts are represented as contravariant structural types⁵ which complicates the use of the system.

3.5 Future Work

HOAS patterns parameterized by types While in theory these can be supported by combining polymorphic lambdas and contextual lambdas, these are not supported in the current implementation.

```
case '{ def f[T]: U = $f[T]; ... } => f: [T] => Type[T] ?=> Expr[U]
```

Statically checked scope extrusions We attempted to make scope extrusion checks using path-dependent types⁶. This made the types for `Expr` and `Type` dependent on the instance of `Quotes` and used some extra type refinements to encode sub-scoping. In this system extrusions are impossible, but it added extra complexity to the user of the system. The complexity costs were deemed to be great too to be justified and we opted to keep the run-time checks instead.

The alternative and initial idea is to use a co-effect system to ensure that expressions are not extruded through side effects. The first step towards this co-effect system was described by Odersky et al. [54]. This approach has the potential of avoiding any extra complexity on the user side.

Let insertion Let insertion [9; 14; 34; 41], also known as *genlet* in multi-stage programming, is not an indispensable feature for macros yet it might make some code generation abstractions simpler to write. This abstraction gives a way to splice a reference to an expression that is not yet bounded in the surrounding expression. A let binding will be inserted automatically.

⁵`type Code[+Typ, -Ctx]`

⁶<https://github.com/lampepfl/dotty/pull/8940>

Generic instantiation One feature that was available in the Scala 2 experimental macros was the ability to instantiate a generic type. This was performed in a syntactic way and lacked any kind of static guarantees. It should be possible to achieve the same behavior in a statically safe way by providing a type class. A `new T` would be accepted if there is a `New[T]` available in scope. A `Type[T]` might also be required.

```
def f[T: Type : New]: Expr[T] = '{ new T }
```

To be complete, `New` should also allow instantiation using arguments for the constructor. This implies that it would probably be a `New[T] [(T1, ..., Tn)]` where `T1, ..., Tn` are the types of the constructor's parameters. The compiler could automatically generate instances for classes where the constructors that are unambiguous based on the parameter types.

Function matching While it is possible to match on a method definition, its arity must be known. Currently, the best that can be done is to match on all methods of a statically known arity one after the other. It would be beneficial to find an abstraction that allows matching on any method definition.

```
case '{ def f($args*): T = $body(args*) } =>
```

The challenge is to figure out what kind of data type is extracted out and how it would interact with other abstractions such as HOAS patterns. Methods may have type parameters and several term argument lists, which might cause extra complexity.

Pattern patterns Matching and extracting a regular Scala pattern that is within a quoted expression is currently impossible using quote patterns. This ability might be useful for DSLs that want to re-interpret or analyze `match` expressions.

```
case '{ x match { case $pat(x, y) => $body(x, y) } } =>
```

The challenge is that a new kind of `Pattern` construct must be added to the language. This `Pattern` would need to statically know the type of its scrutinee and the extracted type. To be complete, it needs to understand all kinds of patterns and handle them in a uniform way.

With this feature we could also imagine the introduction of quoted pattern literals as an alternative form of quotation that could be spliced in a pattern of a quoted expression.

```
val pat: Pattern[..] = '{case Some($_: Int)}  
'{ x match $pat(y) => y }
```

Class patterns Matching on class, trait or object definitions is not trivial due to the complexity of their signatures. They can extend other classes and define any number of new members. It would be interesting to investigate the feasibility and usefulness of these patterns. This has the potential to be extremely difficult to achieve in general but might be approachable for a useful subset of the language.

```
case '{ class A { ... }; } =>
```

3.6 Conclusion

Metaprogramming has a reputation for being difficult and confusing. However, with multi-stage programming it can become downright pleasant. The strong static guarantees are the key to provide a way to express code generation and analysis with ease and confidence.

We integrate multi-stage programming seamlessly with the language using inline methods. We can support macros and run-time code generation by reducing the core multi-stage system to its essential components. We use type classes to lift and unlift values into and out of quoted expressions. We provide flexible pattern matching on quoted code without introducing extra complexity in the rest of the system.

4 Multi-Stage Macro Calculus

This chapter contains part of a published paper authored by Stucki, Brachthäuser, and Odersky [73]. Three new calculus extensions are added to the original work.

In metaprogramming, code generation and code analysis are complementary. Traditionally, multi-stage metaprogramming extensions for programming languages, like MetaML and BER MetaOCaml, offer strong foundations for code generation but lack equivalent support for code analysis. Similarly, existing macro systems are biased towards code generation.

In this chapter, we present a *polymorphic multi-stage macros calculus* featuring both code generation and code analysis. The calculus directly models separate compilation of macros, internalizing a commonly neglected aspect of macros. The system ensures that the generated code is well typed and hygienic. These foundations are used to implement Scala 3 multi-stage macros.

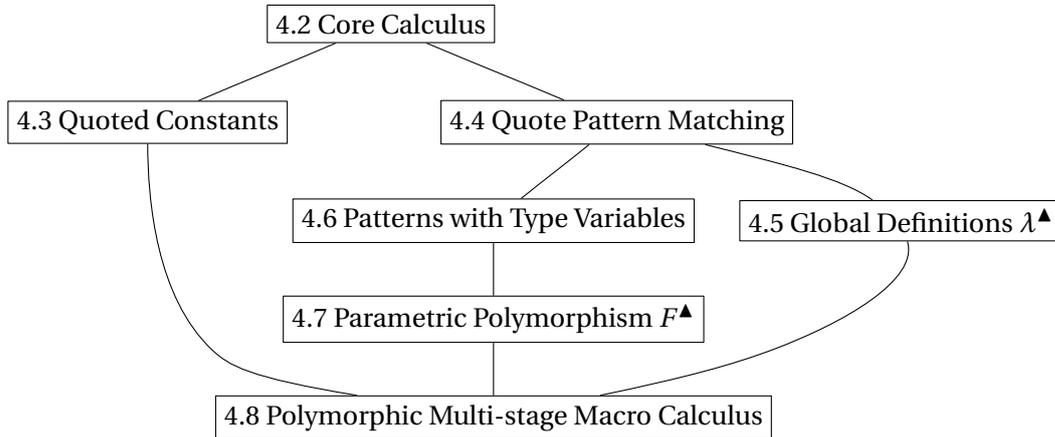
Requirements In Chapter 3, we identified the requirements that a design of a multi-stage macro system for compiled languages should meet:

- *Cross-platform portability*. It should be possible to use generated code on different machines.
- *Static safety*. Generated code should be hygienic and well typed.
- *Cross-stage safety*. Access to variables should only be allowed at stages where they are available.
- *Generative and Analytical*. Programmers should be able to generate as well as analyze and decompose code.

We present a formal calculus that captures the fundamental aspects of the Scala 3 multi-stage macro system. The formalization and implementation advance the state of the art, and satisfy the requirements listed above. We prove soundness of this formalization.

4.1 Multi-Stage Calculus

We present the λ^\blacktriangle multi-stage macros calculus [73] and new extensions. In particular, the *System F* ^{\blacktriangle} polymorphic multi-stage calculus extensions generalize the pattern match of the λ^\blacktriangle multi-stage macros calculus. The presentation is organized into six parts. We first introduce the core calculus in Section 4.2, which extends the simply-typed lambda calculus (STLC) with support for quotes and splices. The two abstractions are at the core of cross-stage safety and static safety. In a second step in Section 4.3, we extend the calculus with operations on quoted constants¹. Then we extend the core calculus with quote analysis by adding quote pattern matching in Section 4.4. We extend the previous calculus to also capture the semantics of compilation of macros using global definitions in Section 4.5. We then extend quote pattern matching to support type variables in Section 4.6. We extend the previous calculus to have *System F* parametric polymorphism in Section 4.7. Finally, we combine all extensions to provide the polymorphic multi-stage macro calculus in Section 4.8 and prove its soundness in Appendix A.



4.2 Core Calculus

This first calculus captures the fundamental semantics of programs that operate on and produce code. To this end, it extends STLC with the ability to delay the computation by quoting the code and the ability to compose delayed computations by splicing.

We use the notation $t_1[t_2/x]$ to denote the standard capture-avoiding substitution of t_2 for x within t_1 . As usual, we follow Barendregt [8] and require that all variable names are globally unique. We also only distinguish terms up to renaming.

Figure 4.1 contains the syntax and semantics of this calculus.

¹A simple extension that should have been in λ^\blacktriangle but was left out due to space limitations in [73].

| | |
|--|--|
| $\text{Term } t ::= \mathbf{c} \mid x \mid \lambda x:T.t \mid t \ t \mid \mathbf{fix} \ t \mid [t] \mid [t]$ | $\text{Typing environment } \Gamma ::= \emptyset \mid \Gamma, x: {}^i T$ |
| $\text{Type } T ::= \mathbf{C} \mid T \rightarrow T \mid [T]$ | $\text{Level } i \in \mathbb{N}_0$ |

| | |
|---|--|
| $\frac{}{\Gamma \vdash^i \mathbf{c} : \mathbf{C}} \quad (\text{T-CONST})$ | $\frac{x: {}^i T \in \Gamma}{\Gamma \vdash^i x : T} \quad (\text{T-VAR})$ |
| $\frac{\Gamma, x: {}^i T_1 \vdash^i t_2 : T_2}{\Gamma \vdash^i \lambda x:T_1.t_2 : T_1 \rightarrow T_2} \quad (\text{T-ABS})$ | $\frac{\Gamma \vdash^i t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash^i t_2 : T_1}{\Gamma \vdash^i t_1 \ t_2 : T_2} \quad (\text{T-APP})$ |
| $\frac{\Gamma \vdash^i t : T \rightarrow T}{\Gamma \vdash^i \mathbf{fix} \ t : T} \quad (\text{T-FIX})$ | $\frac{\Gamma \vdash^{i+1} t : T}{\Gamma \vdash^i [t] : [T]} \quad (\text{T-QUOTE})$ |
| $\frac{\Gamma \vdash^{i-1} t : [T] \quad i \geq 1}{\Gamma \vdash^i [t] : T} \quad (\text{T-SPLICE})$ | |

| | |
|---|---|
| $\frac{t_1 \longrightarrow^i t'_1}{t_1 \ t_2 \longrightarrow^i t'_1 \ t_2} \quad (\text{E-APP-1})$ | $\frac{\vdash^i t_1 \ \mathbf{vl} \quad t_2 \longrightarrow^i t'_2}{t_1 \ t_2 \longrightarrow^i t_1 \ t'_2} \quad (\text{E-APP-2})$ |
| $\frac{\vdash^0 t_2 \ \mathbf{vl}}{(\lambda x:T_1.t_1) \ t_2 \longrightarrow^0 t_1[t_2/x]} \quad (\text{E-BETA})$ | $\frac{t \longrightarrow^i t' \quad i \geq 1}{\lambda x:T.t \longrightarrow^i \lambda x:T.t'} \quad (\text{E-ABS})$ |
| $\frac{t \longrightarrow^{i+1} t'}{[t] \longrightarrow^i [t']} \quad (\text{E-QUOTE})$ | $\frac{t \longrightarrow^i t'}{\mathbf{fix} \ t \longrightarrow^i \mathbf{fix} \ t'} \quad (\text{E-FIX})$ |
| $\mathbf{fix} \ \lambda x:T.t \longrightarrow^0 t[\mathbf{fix} \ \lambda x:T.t/x] \quad (\text{E-FIX-RED})$ | |
| $\frac{t \longrightarrow^{i-1} t' \quad i \geq 1}{[t] \longrightarrow^i [t']} \quad (\text{E-SPLICE})$ | $\frac{\vdash^1 t \ \mathbf{vl}}{[[t]] \longrightarrow^1 t} \quad (\text{E-SPLICE-RED})$ |

| | |
|--|---|
| $\vdash^i \mathbf{c} \ \mathbf{vl} \quad (\text{V-CONST})$ | $\vdash^0 \lambda x:T.t \ \mathbf{vl} \quad (\text{V-ABS-0})$ |
| $\frac{\vdash^{i+1} t \ \mathbf{vl}}{\vdash^i [t] \ \mathbf{vl}} \quad (\text{V-QUOTE})$ | $\frac{i \geq 1}{\vdash^i x \ \mathbf{vl}} \quad (\text{V-VAR})$ |
| $\frac{\vdash^i t \ \mathbf{vl} \quad i \geq 1}{\vdash^i \mathbf{fix} \ t \ \mathbf{vl}} \quad (\text{V-FIX})$ | $\frac{\vdash^i t \ \mathbf{vl} \quad i \geq 1}{\vdash^i \lambda x:T.t \ \mathbf{vl}} \quad (\text{V-ABS})$ |
| $\frac{\vdash^i t_1 \ \mathbf{vl} \quad \vdash^i t_2 \ \mathbf{vl} \quad i \geq 1}{\vdash^i t_1 \ t_2 \ \mathbf{vl}} \quad (\text{V-APP})$ | $\frac{\vdash^{i-1} t \ \mathbf{vl} \quad i \geq 2}{\vdash^i [t] \ \mathbf{vl}} \quad (\text{V-SPLICE})$ |

Figure 4.1: Core Calculus

Syntax

The calculus features the standard forms of simply-typed lambda calculus, that is, constant c , variable x , abstraction $\lambda x:T.t$, and application $t\ t$. In order to express the examples from Chapter 3, we also add support for fixpoint computation $\mathbf{fix}\ t$. The two most important additions to the term syntax are quotation $[t]$ (instead of $\{\tau\}$) and splicing $\llbracket t \rrbracket$ (instead of $\$\{\tau\}$). The syntax of types includes built-in type \mathbf{C} , function type of the form $T \rightarrow T$, and the type of quoted terms $[T]$ (corresponds to $\text{Expr } [T]$). That is, for a term t of type T , the quoted term $[t]$ has type $[T]$.

Example 1. Within the core calculus, we can easily write a function that can generate complex code. The `powerCode` of Listing 3.3 can be encoded in this calculus for a numerical type \mathbf{N} as follows:

$$\mathbf{fix}\ \lambda rec:[\mathbf{N}] \rightarrow \mathbf{N} \rightarrow [\mathbf{N}].$$

$$\lambda x:[\mathbf{N}]. \lambda n:\mathbf{N}. \mathit{ifIsZero}\ n\ [1]\ [\mathit{mult}\ [x]\ \llbracket rec\ x\ (n-1) \rrbracket]$$

Environments

As usual, environments Γ are lists of bindings $x :^i T$. However, they do not only track the type of each binding T , but also at which *staging level* i a variable has been introduced. The *staging level* i is a number in $\mathbb{N}_0 = \{0, 1, 2, \dots\}$. We consider bindings at different levels as disjoint. That is, looking up the binding for x at level j in environment $\Gamma = \Gamma_1, x :^i T, \Gamma_2$ only succeeds if $i = j$. For simplicity, we require well-formedness to establish that environments contain a single binding of each name x , ruling out $\Gamma, x :^i T, x :^j T$ by construction. By definition, the environment guarantees cross-stage safety. Notably, this implies that there is no cross-stage persistence of local variables.

Example 2. Though the calculus itself does not support cross-stage persistence of local variables, it is possible to use values in later stages by lifting and splicing. We generalize this concept in Section 4.3. In the following example, we lift a boolean constant (of type \mathbf{B}) into a quote containing the constant.

$$\lambda x:\mathbf{B}. \mathit{ifIsTrue}\ x\ [\mathbf{true}]\ [\mathbf{false}]$$

Typing

Typing judgments take the form $\Gamma \vdash^i t : T$ and assign the type T to term t . However, they are also parameterized by the staging level i . Conceptually, the level starts at 0, increases each time we encounter a quote (T-QUOTE), and decreases each time we encounter a splice (T-SPLICE). All other typing rules maintain the same level in their premises. Splices cannot be typed at level 0 to avoid negative staging levels. As can be seen in rule T-ABS, bindings are added to the environment with the level at which they are defined. Similarly, variables can only be typed if they were referenced at the level they were defined in (T-VAR).

Example 3. Tracking of levels ensures *cross-stage safety* of variables. Both type derivations in this example fail as expected.

$$\begin{array}{c}
 \frac{}{x:^1 T \in \emptyset, x:^0 T} \text{FAIL} \\
 \hline
 \frac{}{\emptyset, x:^0 T \vdash^1 x : T} \\
 \hline
 \frac{}{\emptyset, x:^0 T \vdash^0 [x] : [T]} \\
 \hline
 \emptyset \vdash^0 \lambda x:T.[x] : T \rightarrow [T]
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{}{x:^0 [T] \in \emptyset, x:^1 [T]} \text{FAIL} \\
 \hline
 \frac{}{\emptyset, x:^1 [T] \vdash^0 x : [T]} \\
 \hline
 \frac{}{\emptyset, x:^1 [T] \vdash^1 [x] : T} \\
 \hline
 \emptyset \vdash^1 \lambda x:[T].[x] : [T] \rightarrow T
 \end{array}$$

Operational Semantics

We present the semantics of our calculus in terms of a small-step operational semantics (Figure 4.1). Like the typing judgments, the evaluation relation is indexed by a staging level i . Also similar to typing, the index starts at 0, increases each time it enters a quote (E-QUOTE), and decreases each time it enters a splice (E-SPLICE). At level 0, the semantics follow the usual STLC semantics and we perform β -reduction (E-BETA) and fix-point computation (E-FIX-RED). The rules E-APP-1, E-APP-2, and E-FIX express the usual congruences. The congruences E-QUOTE and E-SPLICE modify the levels accordingly. Intuitively, the calculus not only performs β -reduction on level 0, but also seeks to reduce all splices at level 1 (E-SPLICE-RED). To achieve this, at levels greater than 0, we need to evaluate under lambdas (E-ABS) in case they contain level 1 splices.

Example 4. The specifics of the operational semantics are illustrated by the following example, which makes use of both reduction rules E-BETA and E-SPLICE-RED. The resulting expression is a value according to our definition since it does not contain any level 1 splice.

$$\begin{array}{ccc}
 [\lambda x:T.(\lambda y:[T].y) [f x]] & \xrightarrow{0} & [\lambda x:T.[f x]] & \xrightarrow{0} & [\lambda x:T.f x] \\
 & \text{E-BETA} & & \text{E-SPLICE-RED} &
 \end{array}$$

Values

While it may appear non-standard, the definition of values $\vdash^i t \mathbf{v!}$ follows directly from the operational semantics. Intuitively, a term is a value if it is a constant (V-CONST), an abstraction (V-ABS-0), or a quote (V-QUOTE) that does not contain any level 1 splices.

Soundness

We show the soundness of the calculus by proving the standard progress and preservation theorems.

Theorem 1 (Progress for Terms).

If $\emptyset \vdash^i t : T$, then t is a value $\vdash^i t \mathbf{v}$ or there exists t' such that $t \longrightarrow^i t'$

Theorem 2 (Preservation for Terms).

If $\Gamma \vdash^i t : T$ and $t \longrightarrow^i t'$, then $\Gamma \vdash^i t' : T$

The full proofs can be found in Appendix A, here we only point out the structure and define key lemmas.

As usual, progress requires us to show that values take canonical forms. The standard canonical forms lemma trivially extends to our non-standard definition of values.

Lemma 1 (Canonical Forms).

- If $\vdash^0 t \mathbf{v}$ and $t : \mathbf{C}$, then $t = \mathbf{c}$ for some \mathbf{c}
- If $\vdash^0 t \mathbf{v}$ and $t : T_1 \rightarrow T_2$, then $t = \lambda x : T_1. t_1$ for some x and t_1
- If $\vdash^0 t \mathbf{v}$ and $t : [T]$, then $t = [t_1]$ for some t_1

Proof of Lemma 1.

By case analysis on the value definition $\vdash^0 t \mathbf{v}$. ■

To prove progress, we also need to prove a more general variant allowing the typing context Γ to only contain bindings on a level greater than 0. To capture this property, we define $\Gamma^{\geq 1}$ as a restricted typing context:

Definition 1 (Restricted Typing Context).

$\Gamma^{\geq 1} ::= \emptyset \mid \Gamma^{\geq 1}, x :^i T \quad \text{for } i \geq 1$

Using the restricted context, we define the generalized version of progress as:

Lemma 2 (Extended Progress for Terms).

If $\Gamma^{\geq 1} \vdash^i t : T$, then t is a value $\vdash^i t \mathbf{v}$ or there exists t' such that $t \longrightarrow^i t'$

Proof of Theorem 1.

The proof of progress trivially follows from Lemma 2, by choosing $\Gamma^{\geq 1} = \emptyset$. ■

For the proof of preservation, we need to adjust the standard substitution lemma to our setting with levels.

Lemma 3 (Substitution).

$\forall i, j \in \mathbb{N}_0$, if $\Gamma \vdash^j t_1 : T_1$ and $\Gamma, x :^j T_1 \vdash^i t_2 : T_2$ then $\Gamma \vdash^i t_2[t_1/x] : T_2$

Proof of Theorem 2.

Induction over the typing derivation, using the substitution lemma (Lemma 3). ■

4.3 Quoted Constants Calculus Extension

This section extends the core calculus of Section 4.2 with the possibility to create quoted constants $[c]$ or extract their values. In Example 2 we saw that it was possible to lift a constant to a quoted constant. But this process becomes inefficient when the domain of the constants is large. To avoid this we introduce a lift operation that takes a constant c into a $[c]$. The inverse operation of taking a $[c]$ into a constant c is also useful for macros.

Figure 4.2 extends Figure 4.1 to define the syntax and semantics of this extended calculus.

Syntax

The syntax of the quoted constants calculus (Figure 4.2) extends the syntax of the core calculus (Figure 4.1). It adds the `lift t` operation that will take a term and lift it into a quote. It also adds the `unlift t with t or t` operation, which takes as argument a quoted term, a lambda containing the operation to evaluate if it is a quoted constant, and a term to evaluate if it is not a quoted constant.

Typing

The quoted constant calculus typing judgments (Figure 4.2) extend the typing judgments of the core calculus (Figure 4.1). Lifting a constant (T-LIFT) is typed similarly to a quote (T-QUOTE) but it does not change the staging level and only works with C types. Unlift is typed (T-UNLIFT) like a pattern match which takes a scrutinee of type $[C]$, a lambda of type $C \rightarrow T$ containing the operation to evaluate if it matches a quoted constant and lastly a term of type T to evaluate if it does not match. None of these operations change the staging level.

Operational Semantics

The operational semantics in Figure 4.2 extend the operational semantics in Figure 4.1. For a `lift t` at staging level 0, we first evaluate the term t until it is a constant c (E-LIFT), then we evaluate by lifting the constant into $[c]$ (E-LIFT-CONST). At all other levels, we also continue to reduce the term t to evaluate level 1 splices. For the `unlift t with t or t` at staging level 0, we first evaluate the scrutinee until it is a constant (E-UNLIFT-SCRUT), then we evaluate the

Multi-Stage Macro Calculus

| |
|--|
| $\text{Term } t ::= \mathbf{c} \mid x \mid \lambda x:T.t \mid t \ t \mid \text{fix } t \mid [t] \mid \llbracket t \rrbracket$ $\mid \text{lift } t \mid \text{unlift } t \text{ with } t \text{ or } t$ |
| $\frac{\Gamma \vdash^i t : \mathbf{C}}{\Gamma \vdash^i \text{lift } t : [\mathbf{C}]} \quad (\text{T-LIFT})$ $\frac{\Gamma \vdash^i t_1 : [\mathbf{C}] \quad \Gamma \vdash^i t_2 : \mathbf{C} \rightarrow T \quad \Gamma \vdash^i t_3 : T}{\Gamma \vdash^i \text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 : T} \quad (\text{T-UNLIFT})$ |
| $\frac{t \longrightarrow^i t'}{\text{lift } t \longrightarrow^i \text{lift } t'} \quad (\text{E-LIFT})$ $\text{lift } \mathbf{c} \longrightarrow^0 [\mathbf{c}] \quad (\text{E-LIFT-CONST})$ $\frac{t_1 \longrightarrow^i t'_1}{\text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \longrightarrow^i \text{unlift } t'_1 \text{ with } t_2 \text{ or } t_3} \quad (\text{E-UNLIFT-SCRUT})$ $\text{unlift } [\mathbf{c}] \text{ with } t_2 \text{ or } t_3 \longrightarrow^0 t_2 \ \mathbf{c} \quad (\text{E-UNLIFT-SUCC})$ $\frac{\vdash^0 t_1 \ \mathbf{vl} \quad t_1 \neq [\mathbf{c}]}{\text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \longrightarrow^0 t_3} \quad (\text{E-UNLIFT-FAIL})$ $\frac{\vdash^i t_1 \ \mathbf{vl} \quad t_2 \longrightarrow^i t'_2 \quad i \geq 1}{\text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \longrightarrow^i \text{unlift } t_1 \text{ with } t'_2 \text{ or } t_3} \quad (\text{E-UNLIFT-WITH})$ $\frac{\vdash^i t_1 \ \mathbf{vl} \quad \vdash^i t_2 \ \mathbf{vl} \quad t_3 \longrightarrow^i t'_3 \quad i \geq 1}{\text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \longrightarrow^i \text{unlift } t_1 \text{ with } t_2 \text{ or } t'_3} \quad (\text{E-UNLIFT-OR})$ |
| $\frac{\vdash^i t \ \mathbf{vl} \quad i \geq 1}{\vdash^i \text{lift } t \ \mathbf{vl}} \quad (\text{V-LIFT})$ $\frac{\vdash^i t_1 \ \mathbf{vl} \quad \vdash^i t_2 \ \mathbf{vl} \quad \vdash^i t_3 \ \mathbf{vl} \quad i \geq 1}{\vdash^i \text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \ \mathbf{vl}} \quad (\text{V-UNLIFT})$ |

Figure 4.2: Quoted Constants Calculus Extension

with branch (E-UNLIFT-SUCC) or **or** branch (E-UNLIFT-FAIL) depending on the contents of the quote. At all other levels, we also continue to reduce the scrutinee (E-UNLIFT-SCRUT), the **with** branches (E-UNLIFT-WITH) and **or** branches (E-UNLIFT-OR) to evaluate level 1 splices.

Example 5.

$$\begin{aligned} \text{lift } \mathbf{c} &\longrightarrow^0 [\mathbf{c}] \\ \text{unlift } [\mathbf{c}] \text{ with } (\lambda x:\mathbf{C}.x) \text{ or } t &\longrightarrow^0 \mathbf{c} \\ \text{unlift } (\text{lift } \mathbf{c}) \text{ with } (\lambda x:\mathbf{C}.x) \text{ or } t &\longrightarrow^0 \mathbf{c} \end{aligned}$$

Values

The value definition in Figure 4.2 extends the value definition in Figure 4.1. At level 0, the lift and unlift operations are evaluated away. At any other level, we need to ensure transitively that sub-terms do not have any splices at level 1.

Soundness

Soundness for this extension follows the same proof structure as the core calculus. The proofs can be extended by adding the missing cases without changing the theorems and lemmas.

4.4 Quote Pattern Matching Extension

We now extend the core calculus of Section 4.2 with support for analytical macros. To this end, we add a pattern matching construct that can deconstruct a piece of code into its components. Sub-expressions of the code can be selectively extracted using a *bind pattern*. Importantly, patterns can only match on a subset of the language, specifically STLC+Fix.

Figures 4.3 and 4.4 extend Figure 4.1 to define the syntax and semantics of this calculus.

Syntax

We extend the core calculus with two new syntactic constructs: a pattern matching operation $t_s \text{ match } [t_p] \text{ then } t_t \text{ else } t_e$, and a bind pattern $\llbracket x \rrbracket_T^{\overline{x_k:T_k^k}}$. The former matches a scrutinee t_s against a pattern t_p . If the match succeeds the **then** branch t_t is evaluated, otherwise the **else** branch t_e is evaluated. A pattern t_p may contain any of the following language constructs: constants, references, lambdas, applications, and fix point operators. In addition, t_p may contain a bind pattern $\llbracket x \rrbracket_T^{\overline{x_k:T_k^k}}$, which will extract a sub-expression of type T from the quote and bind it to x . The extracted sub-expression is locally closed under $\overline{x_k:T_k^k}$, that is, it can contain x_k as free variables. We say the sub-expression is *locally closed*, since in addition to free variables $\overline{x_k:T_k^k}$ bound in the pattern, it can contain free variables defined outside of the pattern. Bind is commonly used without any $\overline{x_k:T_k^k}$ as $\llbracket x \rrbracket_T$ to match a closed sub-expression. For a formula F_k that mentions k , we write $\overline{F_k^k}$ to denote $F_{k_1} F_{k_2} \dots F_{k_n}$ where n is the implicit size of the repetition that depends on k .

Multi-Stage Macro Calculus

| |
|---|
| $\text{Term } t ::= c \mid x \mid \lambda x:T.t \mid t \ t \mid \text{fix } t \mid [t] \mid t $ $\mid t \ \text{match } [t] \ \text{then } t \ \text{else } t \mid \llbracket x \rrbracket_T^{\overline{xk:T_k^k}}$ |
| $\frac{\Gamma \vdash^i t_s : [T_p] \quad \emptyset \vdash^{i+1} t_p : T_p \dashv \Gamma_t \quad \Gamma; \Gamma_t \vdash^i t_t : T \quad \Gamma \vdash^i t_e : T}{\Gamma \vdash^i t_s \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t_e : T} \quad (\text{T-MATCH})$ |
| $\frac{\vdash^1 t_s \ \mathbf{vl} \quad t_s \equiv t_p \Rightarrow \sigma}{[t_s] \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t_e \longrightarrow^0 \sigma(t_t)} \quad (\text{E-MATCH-SUCC})$ $\frac{\vdash^1 t_s \ \mathbf{vl} \quad t_s \equiv t_p \not\Rightarrow \sigma}{[t_s] \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t_e \longrightarrow^0 t_e} \quad (\text{E-MATCH-FAIL})$ $\frac{t_s \longrightarrow^i t'_s}{t_s \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t_e \longrightarrow^i t'_s \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t_e} \quad (\text{E-MATCH-SCRUT})$ $\frac{\vdash^0 t_s \ \mathbf{vl} \quad t_t \longrightarrow^i t'_t \quad i \geq 1}{t_s \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t_e \longrightarrow^i t_s \ \text{match } [t_p] \ \text{then } t'_t \ \text{else } t_e} \quad (\text{E-MATCH-THEN})$ $\frac{\vdash^0 t_s \ \mathbf{vl} \quad \vdash^0 t_t \ \mathbf{vl} \quad t_e \longrightarrow^i t'_e \quad i \geq 1}{t_s \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t_e \longrightarrow^i t_s \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t'_e} \quad (\text{E-MATCH-ELSE})$ |
| $\frac{\vdash^i t_s \ \mathbf{vl} \quad \vdash^i t_t \ \mathbf{vl} \quad \vdash^i t_e \ \mathbf{vl} \quad i \geq 1}{\vdash^i t_s \ \text{match } [t_p] \ \text{then } t_t \ \text{else } t_e \ \mathbf{vl}} \quad (\text{V-MATCH})$ |

Figure 4.3: Structural Quote Patterns Calculus

Typing

Typing a pattern match (T-MATCH) requires that the scrutinee of type $[T_p]$ be matched against a pattern of type T_p . Patterns themselves are typed under a different typing judgment $\Gamma_p \vdash^i t : T \dashv \Gamma_t$. Here Γ_p represents an input and contains the bindings defined within the pattern. In contrast, Γ_t represents an output and contains bindings introduced by the pattern, which are then made available in the **then** branch. The pattern is typed at level $i + 1$ as if it was in a quote (as reflected by the syntax).

| | | |
|---|---------------|---------|
| $\Gamma_p \vdash^i \mathbf{c} : \mathbf{C} \dashv \emptyset$ | (T-PAT-CONST) | |
| $\frac{x : ^i T \in \Gamma_p}{\Gamma_p \vdash^i x : T \dashv \emptyset}$ | (T-PAT-VAR) | |
| $\frac{\Gamma_p, x : ^i T_1 \vdash^i t : T_2 \dashv \Gamma_t}{\Gamma_p \vdash^i \lambda x : T_1. t : T_1 \rightarrow T_2 \dashv \Gamma_t}$ | (T-PAT-ABS) | |
| $\frac{\Gamma_p \vdash^i t_1 : T_1 \rightarrow T_2 \dashv \Gamma_{t_1} \quad \Gamma_p \vdash^i t_2 : T_1 \dashv \Gamma_{t_2}}{\Gamma_p \vdash^i t_1 t_2 : T_2 \dashv \Gamma_{t_1}; \Gamma_{t_2}}$ | (T-PAT-APP) | |
| $\frac{\Gamma_p \vdash^i t : T \rightarrow T \dashv \Gamma_t}{\Gamma_p \vdash^i \mathbf{fix} t : T \dashv \Gamma_t}$ | (T-PAT-FIX) | |
| $\frac{\overline{x_k : ^i T_k \in \Gamma_p}^k}{\Gamma_p \vdash^i \llbracket x \rrbracket_T^{\overline{x_k : T_k}^k} : T \dashv \emptyset, x : ^{i-1} \overline{[T_k]} \rightarrow^k [T]}$ | (T-PAT-BIND) | |
| $\frac{\emptyset \vdash t_s \sqsupset t_p \Rightarrow \sigma}{t_s \equiv t_p \Rightarrow \sigma}$ | | (E-PAT) |
| <i>Pattern bindings</i> $\Phi ::= \emptyset \mid \Phi, x \mapsto x$ | | |
| $\Phi \vdash \mathbf{c} \sqsupset \mathbf{c} \Rightarrow []$ | (E-PAT-CONST) | |
| $\frac{\Phi \vdash t_{s_1} \sqsupset t_{p_1} \Rightarrow \sigma_1 \quad \Phi \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma_2}{\Phi \vdash t_{s_1} t_{s_2} \sqsupset t_{p_1} t_{p_2} \Rightarrow \sigma_1 \circ \sigma_2}$ | (E-PAT-APP) | |
| $\frac{\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma}{\Phi \vdash \mathbf{fix} t_s \sqsupset \mathbf{fix} t_p \Rightarrow \sigma}$ | (E-PAT-FIX) | |
| $\Phi \vdash \Phi(x_p) \sqsupset x_p \Rightarrow []$ | (E-PAT-VAR) | |
| $\frac{\Phi, x_p \mapsto x_s \vdash t_s \sqsupset t_p \Rightarrow \sigma}{\Phi \vdash \lambda x_s : T. t_s \sqsupset \lambda x_p : T. t_p \Rightarrow \sigma}$ | (E-PAT-ABS) | |
| $\frac{FV(t_s) \cap \text{range}(\Phi) \subseteq \overline{\Phi(x_k)}^k \quad t'_s = \overline{\lambda x'_k : [T_k]}^k . [t_s] [\overline{[x'_k]} / \Phi(x_k)]^k}{\Phi \vdash t_s \sqsupset \llbracket x \rrbracket_T^{\overline{x_k : T_k}^k} \Rightarrow [t'_s / x]}$ | (E-PAT-BIND) | |

Figure 4.4: Pattern Semantics

The rules for pattern typing mostly coincide with their term typing counterparts. They only differ in their treatment of environments. First, the Γ_p environment tracks any binding added by a lambda pattern (T-PAT-ABS). It is used to look up references in (T-PAT-VAR) and to determine the types of free variables in a bind pattern (T-PAT-BIND). Second, the Γ_t environment collects x bindings added by $\llbracket x \rrbracket_T^{\overline{x_k:T_k^k}}$, which will be made available in the **then** branch.

The bind pattern $\llbracket x \rrbracket_T^{\overline{x_k:T_k^k}}$ matches against an arbitrary expression locally closed under $\overline{x_k:T_k^k}$. We represent this closed term as a carried function taking arguments of the corresponding types $\overline{T_k^k}$ (T-PAT-BIND). In the output environment, we bind x to a value of type $\overline{[T_k]} \rightarrow [T]$. As a special case of rule T-PAT-BIND, we match on a locally closed sub-expression where the $\overline{x_k:T_k^k}$ is empty, and therefore the type of x is simply $[T]$. Note that the i in this typing judgment is only present to inform at which level x must be added in Γ_t .

Operational Semantics

Once more, the operational semantics in Figures 4.3 and 4.4 extend the operational semantics in Figure 4.1. First of all, to handle quoted pattern matching, we extend the reduction relation $t \longrightarrow^i t'$ with additional rules. At staging level 0, we first evaluate the scrutinee until it is a value (E-MATCH-SCRUT). At all other levels, we also continue to reduce the **then** (E-MATCH-THEN) and **else** branches (E-MATCH-ELSE).

To model the semantics of nested patterns, we introduce an additional reduction relation $\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma$. It states that the sub-term t_s matches the sub-pattern t_p with a substitution map Φ , which provides a mapping from a variable defined in the scrutinee to one defined in the pattern. In addition to matching, the relation also transforms the **then** part of the match t_t into $\sigma(t_t)$ where all bindings defined in the pattern are substituted.

E-PAT is the only evaluation rule for $t_s \equiv t_p \Rightarrow \sigma$ and is only there to introduce the initial empty Φ into $\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma$.

To reduce a match operation, if the pattern matched, we evaluate it into t'_t (E-MATCH-SUCC) where t'_t does not contain any of the bindings defined in the pattern. We say that a pattern (or sub-pattern) did not match if $t_s \equiv t_p \not\Rightarrow \sigma$, that is we cannot derive a match. Therefore, if the pattern did not match we evaluate to the **else** branch t_e (E-MATCH-FAIL).

Exactly as in pattern typing, sub-pattern matching $\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma$ can match the syntactic form of STLC+Fix, and bind sub-terms. Rules that do not introduce bindings in the **then** branch will not modify the σ , they will only propagate the results from sub-evaluation. The rules E-PAT-CONST, E-PAT-FIX, and E-PAT-APP are straightforward.

Interestingly, when matching a lambda, the substitution Φ will track the relationship between the binding in the pattern and in the scrutinee (E-PAT-ABS). When matching a reference to a binding defined in the pattern, we use Φ to know the name of the equivalent binding in the scrutinee (E-PAT-VAR). We only match if those references are equivalent under the Φ mapping.

Finally, we have the bind pattern (E-PAT-BIND), which may match any sub-term as long as it has the correct type *and* the correct free variables. First, consider reduction of the simplified $\llbracket x \rrbracket_T$ pattern using E-PAT-BIND.

$$\frac{FV(t_s) \cap \text{range}(\Phi) \subseteq \emptyset \quad t'_s = \lceil t_s \rceil}{\Phi \vdash t_s \sqsupset \llbracket x \rrbracket_T \Rightarrow \lceil t'_s / x \rceil}$$

In this case, the premise requires us to show that the intersection of the free variables of t_s and the range of Φ is empty. In other words, it means that t_s does not contain a reference to a binding that was defined in the scrutinee. Since it is only locally closed, it may still have references to bindings defined *outside* of the pattern match, as those will be valid when inserted in the **then** branch (*cf.*, rule T-MATCH). If the match succeeds, rule E-PAT-BIND reduces to $t_t \lceil \lceil t_s \rceil / x \rceil$. Now, consider the case where $\overline{x_k : T_k}^k$ is not empty. This implies that we will match a t_s that might contain references to bindings defined in the pattern. In this case, we cannot simply substitute t_s for x , we first need to re-bind all free variables. In particular, for each free variable $\Phi(x_k)$ defined in the pattern, we η -expand t'_s by creating a lambda that receives a staged argument of type $x'_k : \lceil T_k \rceil$. In the body of that lambda we substitute $\Phi(x_k)$ with the $\lceil x'_k \rceil$. This process results in a curried lambda of the type $\overline{\lceil T_k \rceil}^k \rightarrow \lceil T \rceil$.

Example 6. In the following example, we show how to perform β -reduction at level 1 using quote matching. The code below is an encoding of HOAS pattern example of Listing 3.8 for a numerical type **N**.

$$\lambda x : \lceil \mathbf{N} \rceil . x \text{ match } \lceil (\lambda y : \mathbf{N} . \llbracket f \rrbracket_{\mathbf{N}}^{y : \mathbf{N}}) \llbracket z \rrbracket_{\mathbf{N}} \rceil \text{ then } f \ z \text{ else } x$$

Values

The value definition in Figure 4.3 extends the value definition in Figure 4.1. At level 0, the pattern match operations are evaluated away. At any other level, we need to ensure transitively that sub-terms do not have any splices at level 1. As the pattern is not evaluated by itself it is considered a value of its own.

Example 7. The ability to match individually quoted constants allows us to unlift a quoted value into a value known in the current stage. In the example, we unlift a boolean constant (**B**) returning the value applied to *succ* or *fail* if it is not a constant.

$$\lambda x : \mathbf{B} . \lambda \text{succ} : \mathbf{B} \rightarrow T . \lambda \text{fail} : T .$$

$$x \text{ match } \lceil \text{true} \rceil \text{ then } \text{succ } \text{true} \text{ else}$$

$$x \text{ match } \lceil \text{false} \rceil \text{ then } \text{succ } \text{false} \text{ else } \text{fail}$$

Substitution

To handle the new syntactic form of pattern matching, substitution is extended to homomorphically apply substitution to its sub-terms. As patterns can only refer to bindings defined in the pattern itself, as ensured by Γ_p , substitution does not need to go into the pattern.

Soundness

The statements of the progress and preservation theorems carry over unchanged from Section 4.2. Extending the proof of progress with a case for pattern matching is trivial; depending on the pattern reduction, we simply invoke E-MATCH-SUCC or E-MATCH-FAIL. The proof of preservation requires a few additional definitions and lemmas. First of all, we extend the substitution lemma to parallel substitutions:

Lemma 4 (Multi-Substitution).

$\forall i, j \in \mathbb{N}_0$, if $\overline{\Gamma \vdash^j t_k : T_k}^k$ and $\overline{\Gamma, x :^j T_k \vdash^i t : T}^k$ then $\Gamma \vdash^i \overline{t[t_k/x_k]}^k : T$

Next, we state well-formedness of the substitution Φ with respect to environments Γ_p and Γ_δ .

Definition 2 (Well-Formed Φ).

We say Φ is well formed with respect to Γ_p and Γ_δ , written $\Gamma_p \mid \Gamma_\delta \vdash \Phi \mathbf{wf}$, if and only if Φ is a bijection between $dom(\Gamma_p)$ and $dom(\Gamma_\delta)$, such that $dom(\Gamma_p) \cap dom(\Gamma_\delta) = \emptyset$ and

$\forall x_p :^1 T \in \Gamma_p. \Phi(x_p) :^1 T \in \Gamma_\delta$.

Using Definition 2, we can state type preservation of the pattern reduction. That is, reducing a well typed match $\Phi \vdash t_s \boxdot p \Rightarrow \sigma$ results in a well typed term t' .

Lemma 5 (Preservation of Pattern Reduction).

If Eqs. (1) to (5) hold, then $\Gamma \vdash^0 \sigma(t) : T$

$$\Gamma; \Gamma_\delta \vdash^1 t_s : T_1 \tag{1}$$

$$\Gamma_p \vdash^1 t_p : T_1 \dashv \Gamma_t \tag{2}$$

$$\Gamma; \Gamma_t \vdash^0 t : T \tag{3}$$

$$\Phi \vdash t_s \boxdot t_p \Rightarrow \sigma \tag{4}$$

$$\Gamma_p \mid \Gamma_\delta \vdash \Phi \mathbf{wf} \tag{5}$$

Here, premises (1) to (3) correspond to the premises of rule T-MATCH. Finally, the match case in the proof of preservation follows directly from Lemma 5:

Lemma 6 (Preservation for Match).

If $\Gamma \vdash^0 [t_s] \mathbf{match} [t_p] \mathbf{then} t_t \mathbf{else} t_e : T$ and $\vdash^1 t_s \mathbf{vl}$ and $t_s \equiv t_p \Rightarrow \sigma$, then $\Gamma \vdash^0 \sigma(t_t) : T$

4.5 Global Definitions Extension

The calculus from the previous section allowed us to write programs that generate and analyze code using quotes and splices.

In realistic compiled languages, code is first compiled on some machine and then used on a potentially different machine. If a macro is executed when compiling, the code it generates will also be compiled and then used on another machine. This implies that we need cross-platform portability to compile the generated code. If a macro definition is itself compiled, we need to compile a program containing quoted code. In practice, this requires a form of serialization but for this to be sound it also requires cross-platform portability.

To capture the semantics of compiling programs, we extend the previous calculus with global library function definitions. These definitions will be compiled before they are used. The calculus also adds a restricted notion of cross-stage persistence that is compatible with cross-platform portability.

Figure 4.5 extends Figures 4.1, 4.3 and 4.4 to define the syntax and semantics of the λ^\blacktriangle multi-stage macro calculus.

Syntax

The calculus extends the syntax of Figure 4.3 adding a syntactic form for programs p . Programs are lists of library bindings `def $x = [t]$ in p` ending in a single term `eval t` that will be evaluated after all library bindings have been compiled. Here, `def $x = [t]$ in p` represents the definition of a library function that is made available as x in the remaining program. The implementation t , which is given as *code*, is compiled and then added to a simplified store, which we use to model the set of compiled functions loaded in the program. In this program, a macro is a splice within t . The syntactic form `eval t` represents a program that will be evaluated without being compiled. In practice, `eval t` would be an interpreted call to the `main` method.

Environment

Figure 4.5 introduces store-like run-time libraries Ω that map library function names x to their implementation. It also adds a new typing environment Σ , which types libraries Ω and allows us to track references to compiled programs. Values will only ever be added to the library Ω but never updated. Importantly, the bindings in Σ (and Ω) are not annotated with staging levels and are thus staging-level agnostic.

Multi-Stage Macro Calculus

| |
|---|
| <p style="text-align: center;"> <i>Program</i> $p ::= \text{eval } t \mid \text{def } x = [t] \text{ in } p$ <i>Library typing</i> $\Sigma ::= \emptyset \mid \Sigma, x : T$ <i>Global store</i> $\Omega ::= \emptyset \mid \Omega, x := t$ </p> |
| $\frac{\Sigma \mid \emptyset \vdash^1 t : T_1 \quad \Sigma, x : T_1 \vdash p : T_2}{\Sigma \vdash \text{def } x = [t] \text{ in } p : T_2} \quad (\text{T-DEF})$ |
| $\frac{\Sigma \mid \emptyset \vdash^0 t : T}{\Sigma \vdash \text{eval } t : T} \quad (\text{T-EVAL})$ |
| $\frac{x : T \in \Sigma}{\Sigma \mid \Gamma \vdash^i x : T} \quad (\text{T-LINK})$ |
| $\frac{x : T \in \Sigma}{\Sigma \mid \Gamma_p \vdash^i x : T \dashv \emptyset} \quad (\text{T-PAT-LINK})$ |
| $\frac{t \xrightarrow{\Omega}^0 t'}{\text{eval } t \mid \Omega \longrightarrow \text{eval } t' \mid \Omega} \quad (\text{E-EVAL})$ |
| $\frac{t \xrightarrow{\Omega}^1 t'}{\text{def } x = [t] \text{ in } p \mid \Omega \longrightarrow \text{def } x = [t'] \text{ in } p \mid \Omega} \quad (\text{E-MACRO})$ |
| $\frac{\vdash^1 t \mathbf{vl}}{\text{def } x = [t] \text{ in } p \mid \Omega \longrightarrow p \mid \Omega, x := t} \quad (\text{E-COMPILE})$ |
| $\frac{x \in \text{dom}(\Omega)}{x \xrightarrow{\Omega}^0 \Omega(x)} \quad (\text{E-LINK})$ |
| $\Phi \vdash x \boxtimes x \Rightarrow [] \quad (\text{E-PAT-LINK})$ |
| $\frac{\vdash^0 t \mathbf{vl}}{\vdash \text{eval } t \mathbf{vl}} \quad (\text{V-EVAL})$ |

Figure 4.5: Global Definitions Extension

Typing

The typing judgments in Figure 4.5 extend the ones presented in Figure 4.1. We modify the judgment form $\Gamma \vdash^i t : T$ to also track the library typing Σ as $\Sigma | \Gamma \vdash^i t : T$. All existing rules simply pass Σ unmodified to their premises. To type programs, we add a new typing judgment $\Sigma \vdash p : T$ where Σ tracks the library bindings. When typing a library definition $\mathbf{def} \ x = [t] \ \mathbf{in} \ p$ (T-DEF), we type the term t at level 1. This way the term t can contain splices, which in turn allows us to model macros. The rest of the program is typed by adding x to the library environment Σ . Typing $\mathbf{eval} \ t$ (T-EVAL) simply types t at level 0 (like in Section 4.2) but adds the Σ , which will not change in the remainder of the derivation. To be able to access library functions, we add rules T-LINK and T-PAT-LINK, which look up the signature of a free variable x in Σ . We assume that $\text{dom}(\Sigma)$ and $\text{dom}(\Gamma)$ are disjoint and hence there cannot be any ambiguity with (T-VAR). Note that, unlike rule T-VAR, variables in Σ are stage-polymorphic, therefore library functions display a form of cross-stage persistence.

Example 8. Library functions in Σ can be used at any level after they are compiled. This is illustrated by the typing derivation below, where f is used at staging levels 0 and 1. Assuming that $f : [\mathbf{C}] \rightarrow \mathbf{C} \in \Sigma$, we can see that all premises are satisfiable with $T_1 = T_3 = [\mathbf{C}]$ and $T_2 = \mathbf{C}$.

$$\begin{array}{c}
 \frac{f : T_3 \rightarrow T_2 \in \Sigma \quad \Sigma | \emptyset \vdash^2 \mathbf{c} : \mathbf{C} \quad T_3 = [\mathbf{C}]}{\Sigma | \emptyset \vdash^1 f : T_3 \rightarrow T_2 \quad \Sigma | \emptyset \vdash^1 [\mathbf{c}] : T_3} \quad T_1 = [T_2] \\
 \frac{\Sigma | \emptyset \vdash^1 f [\mathbf{c}] : T_2}{\Sigma | \emptyset \vdash^0 f [\mathbf{c}] : T_1} \\
 \frac{f : T_1 \rightarrow \mathbf{C} \in \Sigma \quad \Sigma | \emptyset \vdash^0 f : T_1 \rightarrow \mathbf{C}}{\Sigma | \emptyset \vdash^0 f [f [\mathbf{c}]] : \mathbf{C}} \\
 \hline
 \Sigma \vdash \mathbf{eval} \ f [f [\mathbf{c}]] : \mathbf{C}
 \end{array}$$

Example 9. In the following example, we show how the combined calculus can be used to describe an optimization of our “DSL for mathematical operations”. We match a numeric expression and check whether it is a call to our global *power* DSL function defined in a library. The code below is an encoding of the last version of `fusedPowCode` of Listing 3.6 for a numerical type `N` with a `*` multiplication operation.

```
def fusedPowCode = [
  fix λrec:[N]→[N]→[N].λx:[N].λn:[N].
    x match [power [y]_N [m]_N] then rec y [* [n] [m]] else [power [x] [n]]
] in def power4 = [
  λn:N.[fusedPowCode [power n 2] 2]
] in p
```

Operational Semantics

Like in the case of typing, the operational semantics in Figure 4.5 extends the one presented in Figure 4.3 by modifying the relation $t \longrightarrow^i t'$ to be indexed with a run-time library Ω as $t \longrightarrow_{\Omega}^i t'$. Also, like in typing, all existing rules simply pass on Ω to their premises. To specify the evaluation of programs, we introduce a new relation $p \mid \Omega \longrightarrow p' \mid \Omega'$, where a program p with a library Ω evaluates to a program p' with a potentially extended library Ω' . For a library definition `def x = [t] in p`, where t is a value (*i.e.*, where t does not contain macros), the library store is updated with a binding $x := t$ and the definition is removed (E-COMPILE). Importantly, in this process we are taking a t at staging level 1 and compiling it, making it available as a run-time dependency in the rest of the program; the compiled library function t can be used on arbitrary levels, including level 0. If the library function t in a definition `def x = [t] in p` still contains splices at staging level 1 (macros), we first need to evaluate them (E-MACRO). Using the reduction of the core calculus from Section 4.2, the contents of the macros will be evaluated at that point. This will produce a quote value that is then canceled with the splice. To evaluate the final expression `eval t`, we simply reduce the term t using the run-time dependencies in Ω (E-EVAL). At this point, Ω is fixed and will not change, and it will just propagate down to allow E-LINK to use Ω . A reference x to a library function typed with T-LINK at level 0 will lead to x being replaced by the compiled code from Ω (E-MACRO). We say that we “link the reference with the compiled function”. At any other level $i \geq 1$, we simply keep the reference to x , since it is considered a value. When matching a library function reference (E-PAT-LINK) we match if the scrutinee is a reference to the same library function. This pattern does not change or use Φ in any way as it only needs to handle local variables.

Values

The definition of values of Figure 4.3 is kept unchanged. We merely add a value definition for programs $\vdash p \mathbf{v}$. The only program value is $\mathbf{eval} \ t$ where t is required to be a value (V-EVAL).

Example 10. The following example illustrates evaluation in the calculus. We define a library function *powerCode* (implemented as in Example 1) as a macro taking a quoted base x and an exponent n of numeric type (\mathbf{N}); and a library function *power2* using the macro with 2 as the exponent.

$$\begin{array}{l} \mathbf{def} \ \mathit{powerCode} = [\mathbf{fix} \ \lambda \mathit{rec} : [\mathbf{N}] \rightarrow \mathbf{N} \rightarrow [\mathbf{N}]. \dots] \ \mathbf{in} \\ \mathbf{def} \ \mathit{power2} = [\lambda x : \mathbf{N}. [\mathit{powerCode} \ [x] \ 2]] \ \mathbf{in} \\ \mathbf{eval} \ \mathit{power2} \ 3 \end{array} \quad \Bigg| \quad \emptyset$$

First, we *compile* the macro definition *powerCode* since it does not contain any splices at level 1, storing it in Ω .

$$\begin{array}{l} \mathbf{def} \ \mathit{power2} = [\lambda x : \mathbf{N}. [\mathit{powerCode} \ [x] \ 2]] \ \mathbf{in} \\ \mathbf{eval} \ \mathit{power2} \ 3 \end{array} \quad \Bigg| \quad \emptyset, \ \mathit{powerCode} := \mathbf{fix} \ \lambda \mathit{rec} : [\mathbf{N}] \rightarrow \mathbf{N} \rightarrow [\mathbf{N}]. \dots$$

Next, we perform *macro expansion* and evaluate the code in the splice of *power2*.

$$\begin{array}{l} \mathbf{def} \ \mathit{power2} = [\lambda x : \mathbf{N}. [x * x * 1]] \ \mathbf{in} \\ \mathbf{eval} \ \mathit{power2} \ 3 \end{array} \quad \Bigg| \quad \emptyset, \ \mathit{powerCode} := \mathbf{fix} \ \lambda \mathit{rec} : [\mathbf{N}] \rightarrow \mathbf{N} \rightarrow [\mathbf{N}]. \dots$$

Performing splice canceling results in:

$$\begin{array}{l} \mathbf{def} \ \mathit{power2} = [\lambda x : \mathbf{N}. x * x * 1] \ \mathbf{in} \\ \mathbf{eval} \ \mathit{power2} \ 3 \end{array} \quad \Bigg| \quad \emptyset, \ \mathit{powerCode} := \mathbf{fix} \ \lambda \mathit{rec} : [\mathbf{N}] \rightarrow \mathbf{N} \rightarrow [\mathbf{N}]. \dots$$

As before, we compile *power2*, which does not contain splices at level 0 anymore.

$$\mathbf{eval} \ \mathit{power2} \ 3 \quad \Bigg| \quad \emptyset, \ \mathit{powerCode} := \mathbf{fix} \ \lambda \mathit{rec} : [\mathbf{N}] \rightarrow \mathbf{N} \rightarrow [\mathbf{N}]. \dots, \ \mathit{power2} = \lambda x : \mathbf{N}. x * x * 1$$

Finally, we evaluate the *main* program:

$$\mathbf{eval} \ 9 \quad \Bigg| \quad \emptyset, \ \mathit{powerCode} := \mathbf{fix} \ \lambda \mathit{rec} : [\mathbf{N}] \rightarrow \mathbf{N} \rightarrow [\mathbf{N}]. \dots, \ \mathit{power2} = \lambda x : \mathbf{N}. x * x * 1$$

Soundness

To state the progress and preservation theorems for the λ^Δ calculus, we introduce an auxiliary well-formedness relation $\Sigma \vdash \Omega \mathbf{wf}$, which means that Ω is well typed under environment Σ .

Definition 3 (Well-Formed Ω).

$\Sigma \vdash \Omega \mathbf{wf}$ if and only if $\mathit{dom}(\Sigma) = \mathit{dom}(\Omega) \wedge \forall i \in \mathbb{N}_0, x \in \mathit{dom}(\Omega). \Sigma \upharpoonright \emptyset \vdash^i \Omega(x) : \Sigma(x)$

Using this definition, we state the soundness of the calculus in terms of progress and preservation for programs. Again, the full proofs can be found in Appendix A.

Theorem 3 (Progress for Programs).

If $\Sigma \vdash p : T$, then p is a value $\vdash p \mathbf{v!}$ or, for any Ω such that $\Sigma \vdash \Omega \mathbf{wf}$, there exists p' and Ω' such that $p \mid \Omega \longrightarrow p' \mid \Omega'$

Theorem 4 (Preservation for Programs).

If $\Sigma \vdash p : T$, $\Sigma \vdash \Omega \mathbf{wf}$ and $p \mid \Omega \longrightarrow p' \mid \Omega'$, then there exists a Σ' such that $\Sigma' \supseteq \Sigma$, $\Sigma' \vdash p' : T$ and $\Sigma' \vdash \Omega' \mathbf{wf}$

Since we added rule T-LINK, we need to revisit the lemmas from the previous section. Lemma 1 (of Canonical Forms) still holds because there was no change in value definitions.

We also need to restate the various theorems and lemmas for terms to account for libraries and library typing.

Theorem 5 (Progress for Terms).

If $\Sigma \mid \emptyset \vdash^i t : T$ and $\Sigma \vdash \Omega \mathbf{wf}$, then t is a value $\vdash^i t \mathbf{v!}$ or there exists t' such that $t \longrightarrow_{\Omega}^i t'$

Lemma 7 (Extended Progress for Terms).

If $\Sigma \mid \Gamma^{\geq 1} \vdash^i t : T$ and $\Sigma \vdash \Omega \mathbf{wf}$, then t is a value $\vdash^i t \mathbf{v!}$ or there exists t' such that $t \longrightarrow_{\Omega}^i t'$

Theorem 6 (Preservation for Terms).

If $\Sigma \mid \Gamma \vdash^i t : T$, $t \longrightarrow_{\Omega}^i t'$ and $\Sigma \vdash \Omega \mathbf{wf}$, then $\Sigma \mid \Gamma \vdash^i t' : T$

Lemma 8 (Substitution).

$\forall i, j \in \mathbb{N}_0$, if $\Sigma \mid \Gamma \vdash^j t_1 : T_1$ and $\Sigma \mid \Gamma, x : T_1 \vdash^i t_2 : T_2$ then $\Sigma \mid \Gamma \vdash^i t_2[t_1/x] : T_2$

Most of the proofs carry through unchanged. To show preservation we additionally need the following lemma.

Lemma 9 (Σ -Weakening).

If $\Sigma \vdash p : T$ and $\Sigma' \supseteq \Sigma$, then $\Sigma' \vdash p : T$

4.6 Patterns with Type Variables Extension

It is not always possible or practical to match on a pattern where all types are statically known. For example, if we want a single pattern that can match any application, we do not wish to repeat the pattern for each argument type.

To this end, we extend the calculus of Section 4.4 with support for existential type variables in patterns. These type variables are local to the match construct and are immediately evaluated away on a successful match. The extension introduces a notion of type constraints and unification in the operational semantics.

Figures 4.6 and 4.7 extend Figures 4.1, 4.3 and 4.4 to define the syntax and semantics of this extended calculus.

| |
|---|
| $ \begin{array}{l} \text{Term } t ::= \mathbf{c} \mid x \mid \lambda x:T.t \mid t \ t \mid \mathbf{fix} \ t \mid [t] \mid [t] \\ \quad \quad \quad \mid t \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t \ \mathbf{else} \ t \mid \llbracket x \rrbracket_T^{\bar{x}_k:T^k} \\ \text{Type } T ::= \mathbf{C} \mid T \rightarrow T \mid [T] \mid X \\ \text{Typing environment } \Gamma ::= \emptyset \mid \Gamma, x :^i T \mid \Gamma, X \\ \text{Constraints } C ::= \emptyset \mid C, T = T \end{array} $ |
| $ \frac{\Gamma \vdash^i t_s : [T_p] \quad \Gamma; \bar{X} \mid \emptyset \vdash^{i+1} t_p : T_p \dashv \Gamma_t \quad \Gamma; \bar{X}; \Gamma_t \vdash^i t_t : T \quad \Gamma \vdash^i t_e : T}{\Gamma \vdash^i t_s \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t_t \ \mathbf{else} \ t_e : T} \quad (\text{T-MATCH}) $ |
| $ \frac{t_s \longrightarrow^i t'_s}{t_s \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t_t \ \mathbf{else} \ t_e \longrightarrow^i t'_s \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t_t \ \mathbf{else} \ t_e} \quad (\text{E-MATCH-SCRUT}) $ $ \frac{\vdash^1 t_s \ \mathbf{vl} \quad \bar{X} \vdash t_s \equiv t_p \Rightarrow \sigma}{[t_s] \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t_t \ \mathbf{else} \ t_e \longrightarrow^0 \sigma(t_t)} \quad (\text{E-MATCH-SUCC}) $ $ \frac{\vdash^1 t_s \ \mathbf{vl} \quad \bar{X} \vdash t_s \equiv t_p \not\Rightarrow \sigma}{[t_s] \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t_t \ \mathbf{else} \ t_e \longrightarrow^0 t_e} \quad (\text{E-MATCH-FAIL}) $ $ \frac{\vdash^0 t_s \ \mathbf{vl} \quad t_t \longrightarrow^i t'_t \quad i \geq 1}{t_s \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t_t \ \mathbf{else} \ t_e \longrightarrow^i t_s \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t'_t \ \mathbf{else} \ t_e} \quad (\text{E-MATCH-THEN}) $ $ \frac{\vdash^0 t_s \ \mathbf{vl} \quad \vdash^0 t_t \ \mathbf{vl} \quad t_e \longrightarrow^i t'_e \quad i \geq 1}{t_s \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t_t \ \mathbf{else} \ t_e \longrightarrow^i t_s \ \mathbf{match} \ \bar{X} \ [t_p] \ \mathbf{then} \ t_t \ \mathbf{else} \ t'_e} \quad (\text{E-MATCH-ELSE}) $ |

Figure 4.6: Pattern Type Variables Calculus

| | | |
|---|---------------|-------------|
| $\frac{\emptyset \vdash t_s \sqsupset t_p \Rightarrow \sigma_1 \mid C \quad \text{unify}(\overline{X}_i^i \mid C) \Rightarrow \sigma_2}{\overline{X}_i^i \vdash t_s \equiv t_p \Rightarrow \sigma_2 \circ \sigma_1}$ | (E-PAT) | |
| $\Phi \vdash \mathbf{c} \sqsupset \mathbf{c} \Rightarrow [] \mid \emptyset$ | (E-PAT-CONST) | |
| $\frac{\Phi \vdash t_{s_1} \sqsupset t_{p_1} \Rightarrow \sigma_1 \mid C_1 \quad \Phi \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma_2 \mid C_2}{\Phi \vdash t_{s_1} t_{s_2} \sqsupset t_{p_1} t_{p_2} \Rightarrow \sigma_1 \circ \sigma_2 \mid C_1; C_2}$ | (E-PAT-APP) | |
| $\frac{\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C}{\Phi \vdash \mathbf{fix} t_s \sqsupset \mathbf{fix} t_p \Rightarrow \sigma \mid C}$ | (E-PAT-FIX) | |
| $\Phi \vdash \Phi(x_p) \sqsupset x_p \Rightarrow [] \mid \emptyset$ | (E-PAT-VAR) | |
| $\frac{\Phi, x_p \mapsto x_s \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C}{\Phi \vdash \lambda x_s : T_1. t_s \sqsupset \lambda x_p : T_2. t_p \Rightarrow \sigma \mid C, T_1 = T_2}$ | (E-PAT-ABS) | |
| $\frac{FV(t_s) \cap \text{range}(\Phi) \subseteq \overline{\Phi(x_k)}^k \quad t'_s = \overline{\lambda x'_k : [T_k]^k} \cdot [t_s][[\overline{x'_k}] / \overline{\Phi(x_k)}]^k}{\Phi \vdash t_s \sqsupset [\llbracket x \rrbracket_T^{x_k : T_k}] \Rightarrow [t'_s / x] \mid \{type(t_s) = T\}}$ | (E-PAT-BIND) | |
| $\text{unify}(\emptyset \mid \emptyset) \Rightarrow []$ | | (U-EMPTY) |
| $\frac{\text{unify}(\overline{X}_i^i \mid C) \Rightarrow \sigma}{\text{unify}(\overline{X}_i^i \mid C, T = T) \Rightarrow \sigma}$ | | (U-EQ) |
| $\frac{\text{unify}(\overline{X}_i^i; \overline{X}_j^j \mid C[T/X]) \Rightarrow \sigma}{\text{unify}(\overline{X}_i^i, X, \overline{X}_j^j \mid C, T = X) \Rightarrow [T/X] \circ \sigma}$ | | (U-PAT-VAR) |
| $\frac{\text{unify}(\overline{X}_i^i \mid C, T_{s_1} = T_{p_1}, T_{s_2} = T_{p_2}) \Rightarrow \sigma}{\text{unify}(\overline{X}_i^i \mid C, T_{s_1} \rightarrow T_{s_2} = T_{p_1} \rightarrow T_{p_2}) \Rightarrow \sigma}$ | | (U-ABS) |
| $\frac{\text{unify}(\overline{X}_i^i \mid C, T_s = T_p) \Rightarrow \sigma}{\text{unify}(\overline{X}_i^i \mid C, [T_s] = [T_p]) \Rightarrow \sigma}$ | | (U-QUOTE) |

Figure 4.7: Pattern Semantics

Syntax

The new calculus extends the quote pattern match calculus with type variables. Type variable X is added as a possible type T and the matching operation is extended to define a set \overline{X} of type variables. These type variables are accessible in the pattern and the **then** branch of the pattern match. If the pattern matches, the type variables in the **then** branch are substituted by concrete types found in the pattern. The syntax \overline{X} is used as a shorthand for \overline{X}_k^k when there can be no ambiguity in the indexing.

Environment

The environment Γ is extended to keep track of type variables X . We also use \overline{X} to define an environment that only contains type variables. The addition of type variables implies that we need to make sure that all environments are well formed with respect to type variables. In short, an environment is well formed $\vdash \Gamma$ **wf** if the definition of the type variable appears before the uses in the environment. Similarly, a type is well formed $\Gamma \vdash T$ **wf** if $\vdash \Gamma$ **wf** and $ftv(T) \in \Gamma$. The explicit well-formedness rules can be found in Figures 4.13 to 4.15.

Typing

The typing of T-MATCH is extended to define any number of existential type variables \overline{X} . These type variables are accessible in the pattern and in the **then** branch of the pattern match. Bindings in Γ_t may contain references to these type variables.

Typing rules for patterns are extended with an extra Γ environment at the beginning $\Gamma | \Gamma_p \vdash^i t : T \dashv \Gamma_t$. Γ is the environment of the match term and is only used for well-formedness of the types in the patterns. The typing of **match** (T-MATCH) adds the \overline{X} to the pattern typing environment $\Gamma; \overline{X}$ and the typing environment $\Gamma; \overline{X}; \Gamma_t$ of the **then** branch. This does not change in any of the pattern typing rules.

For convenience, we add the implicit requirement that all environments and types in typing judgments are well-formed. For term typing $\Gamma \vdash^i t : T$, well-formedness of types is defined as $\Gamma \vdash T$ **wf** and well-formedness of environments is defined as $\vdash \Gamma$ **wf**. For pattern typing $\Gamma | \Gamma_p \vdash^i t : T \dashv \Gamma_t$, well-formedness of types is defined as $\Gamma; \Gamma_p \vdash T$ **wf** and well-formedness of environments is defined as $\vdash \Gamma; \Gamma_p$ **wf** and $\vdash \Gamma; \Gamma_t$ **wf**. In particular, the type T in T-PAT-BIND is well formed because it appears in $\vdash \Gamma; \Gamma_t$ **wf**.

Example 11. The following pattern shows how to match on any application regardless of its type using a type variable for the type of the argument. The derivation also shows how the bound type variables interact with the bind pattern and the **then** branch of the match. Here, $\Gamma_t = \emptyset$, $x_1 :^0 [X \rightarrow T]$, $x_2 :^0 [X]$ is defined by the pattern derivation and used in the **then** branch.

$$\frac{\begin{array}{c} \dots \quad \Gamma, X \mid \emptyset \vdash^{-1} \llbracket x_2 \rrbracket_X : X \dashv \emptyset, x_2 :^0 [X] \\ \hline \Gamma, X \mid \emptyset \vdash^{-1} \llbracket x_1 \rrbracket_{X \rightarrow T} \llbracket x_2 \rrbracket_X : T \dashv \emptyset, x_1 :^0 [X \rightarrow T], x_2 :^0 [X] \end{array}}{\Gamma \vdash^0 t \mathbf{match} X \llbracket \llbracket x_1 \rrbracket_{X \rightarrow T} \llbracket x_2 \rrbracket_X \rrbracket \mathbf{then} \llbracket [x_1] [x_2] \rrbracket \mathbf{else} t : [T]} \quad \frac{\begin{array}{c} x_2 :^0 [X] \in \Gamma, X, \dots, x_2 :^0 [X] \\ \hline \Gamma, X, \dots, x_2 :^0 [X] \vdash^0 x_2 : [X] \end{array}}{\dots} \quad \frac{\dots}{\Gamma, X; \Gamma_t \vdash^0 \llbracket [x_1] [x_2] \rrbracket : [T]} \quad \frac{\dots}{\Gamma \vdash^0 t : [T]}$$

Operational Semantics

For terms, the operational semantics of Figure 4.6 are virtually unchanged. We only need to modify E-MATCH-SCRUT, E-MATCH-SUCC, E-MATCH-FAIL, E-MATCH-SCRUT, E-MATCH-THEN and E-MATCH-ELSE to include the \bar{X} in $t_s \mathbf{match} \bar{X} [t_p] \mathbf{then} t_t \mathbf{else} t_e$. The pattern operational semantics will use the \bar{X} while performing the match and therefore must be passed in $\bar{X} \vdash t_s \equiv t_p \Rightarrow \sigma$ from E-MATCH-SUCC, E-MATCH-FAIL into E-PAT.

In E-PAT (Figure 4.7), a pattern that contains a type variable will only match if there exists a type for which the pattern matches the scrutinee. To figure out this type, the pattern matching will also return a set of type constraints C in $\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma_1 \mid C$ (not to be confused with \mathbf{C} for constants). Then the constraints are unified to provide a substitution for all \bar{X} of the pattern in $\mathit{unify}(\bar{X} \mid C) \Rightarrow \sigma_2$. The composition $\sigma_2 \circ \sigma_1$ will substitute all x introduced by the bind patterns and all \bar{X} defined in the match.

The collection of constraints is mostly straightforward with the exception of the bind pattern. E-PAT-CONST and E-PAT-VAR do not add constraints, E-PAT-FIX propagates the constraints, E-PAT-APP combines the constraints and E-PAT-ABS adds the constraint that the two argument types must be the same. On the other hand E-PAT-BIND adds a single constraint $\mathit{type}(t_s) = T$ where $\mathit{type}(t_s)$ is the type of t_s . This implies that for any quoted sub-term we need to be able to recover its type at run-time. This is implicitly assumed in the calculus.

The unification $\mathit{unify}(\bar{X} \mid C) \Rightarrow \sigma$ will try to unify the constraints C with the type variables \bar{X} . The result of the unification is a substitution σ that will be used to substitute all \bar{X} with concrete types. To solve the unification, we need to reach U-EMPTY which states that if we have no type variables and no constraints we can use the empty substitution. U-EQ introduces the notion that if two types are equal they can be removed from the constraint. U-ABS and U-QUOTE simply flatten the constraints. The most interesting rule is the U-PAT-VAR which takes the constraint $T = X$ and will set T to be the solution of X . A key property to make this work is that $\mathit{ftv}(T) \cap \bar{X} = \emptyset$ which means that neither $C[T/X]$ nor $\mathit{range}([T/X])$ will introduce a reference to any of the \bar{X} . This is a property that holds by construction of the constraints as

X can only appear in the pattern and hence on the right-hand side of the constraints. This property is captured by well-formedness of the constraint $\Gamma \mid \bar{X} \vdash C$ **wf**.

Example 12. The following match has an X that needs to be resolved at run-time.

$$[(\lambda x_1:\mathbf{C}.x_1) \mathbf{c}] \text{ match } X \ [(\lambda x_2:X.x_2) \llbracket x \rrbracket_X] \text{ then } [(\lambda x_3:X.x_3) [x]] \text{ else } t$$

The structure of the tree will obviously match and it will collect the $\emptyset, \mathbf{C}=X, \text{type}(\mathbf{c})=X$ which is equivalent to the constraint $\emptyset, \mathbf{C}=X, \mathbf{C}=X$ from the E-PAT-ABS and E-PAT-BIND rules, which are combined using the E-PAT-APP rule. From this we can infer that the type substitution is $[\mathbf{C}/X]$.

$$\frac{\frac{\text{unify}(\emptyset \mid \emptyset) \Rightarrow []}{\text{unify}(\emptyset \mid \emptyset, \mathbf{C}=\mathbf{C}) \Rightarrow [\mathbf{C}/X]}}{\text{unify}(X \mid \emptyset, \mathbf{C}=X, \mathbf{C}=X) \Rightarrow [\mathbf{C}/X]}$$

Therefore the match will succeed and use the combination of the substitution for the bound term x and the substitution for the variable X .

$$\begin{aligned} [(\lambda x_1:\mathbf{C}.x_1) \mathbf{c}] \text{ match } X \ [(\lambda x_2:X.x_2) \llbracket x \rrbracket_X] \text{ then } [(\lambda x_3:X.x_3) [x]] \text{ else } t &\longrightarrow^0 \\ &[\mathbf{C}/X] \circ [\mathbf{c}/x] ([(\lambda x_3:X.x_3) [x]]) \longrightarrow^0 \\ &[(\lambda x_3:\mathbf{C}.x_3) [\mathbf{c}]] \end{aligned}$$

Substitution

To handle the new syntactic form of pattern matching, substitution is extended to homomorphically apply substitution to its sub-terms and sub-types. As patterns can only refer to bindings defined in the pattern itself, as ensured by Γ_p , term substitution does not need to go into the pattern. On the other hand, the pattern might refer to a type defined in Γ , which needs substitution in the pattern.

Soundness

The statements of the progress and preservation theorems carry over unchanged from Section 4.4. The proofs of progress and presentation only need to explicitly mention the \bar{X} .

With the change in E-PAT, Lemma 6 does not follow directly from Lemma 5 anymore. To prove it, we first need Lemma 10 to show that the constraint is well-formed. Then, we need Lemma 11 to prove that the substitution resulting from the unification will replace all the type variables of the match in the **then** branch.

Lemma 10 (Constraints of Pattern Reduction).

If Eqs. (1) to (4) then $\Gamma; \Gamma_\delta \mid \bar{X}; \Gamma_p \vdash C$ **wf**.

$$\Gamma; \Gamma_\delta \vdash^1 t_s : T_s \quad (1)$$

$$\Gamma; \bar{X} \mid \Gamma_p \vdash^1 t_p : T_p \dashv \Gamma_t \quad (2)$$

$$\Phi \vdash t_s \boxplus t_p \Rightarrow \sigma \mid C \quad (3)$$

$$\Gamma_p \mid \Gamma_\delta \vdash \Phi \text{ **wf**} \quad (4)$$

Lemma 11 (Pattern Constraints Unification).

If $\text{unify}(\bar{X} \mid C) \Rightarrow \sigma$ and $\Gamma \mid \bar{X} \vdash C$ **wf** and $\Gamma \vdash T$ **wf** and $\Gamma; \bar{X} \vdash^0 t : T$, then $\Gamma \vdash^0 \sigma(t) : T$

To prove Lemma 6 we can apply Lemma 5 to get the first premise of Lemma 11. And then, using Lemma 10, we get the second premise needed to use Lemma 11.

To prove Lemma 11 we also need to prove Lemma 12 on type substitution in constraints. This is used in the case U-PAT-VAR where we have $C[T/X]$.

Lemma 12 (Constraint Substitution).

If $\Gamma \mid \bar{X}, X \vdash C$ **wf** and $\Gamma \vdash T$ **wf**, then $\Gamma \mid \bar{X} \vdash C[T/X]$ **wf**

To apply the type substitution of Lemma 11 we need the type substitution Lemma 13.

Lemma 13 (Type Substitution).

$\forall i \in \mathbb{N}_0$, if $\Gamma_1 \vdash T_1$ **wf** and $\Gamma_1, X; \Gamma_2 \vdash^i t : T$ then $\Gamma_1; (\Gamma_2[T_1/X]) \vdash^i t[T_1/X] : T[T_1/X]$

As type substitution is also performed on patterns, the T-MATCH case of Lemma 13 will require the auxiliary Lemma 14.

Lemma 14 (Pattern Type Substitution).

$\forall i \in \mathbb{N}$, if $\Gamma_1 \vdash T_1$ **wf** and $\Gamma_1, X; \Gamma_2 \mid \Gamma_p \vdash^i t : T \dashv \Gamma_t$
then $\Gamma_1; (\Gamma_2[T_1/X]) \mid \Gamma_p[T_1/X] \vdash^i t[T_1/X] : T[T_1/X] \dashv \Gamma_t[T_1/X]$

To prove Lemmas 12 to 14 we also need to prove well-formedness of types after substitution.

Lemma 15 (Well-Formed Type Substitution).

If $\Gamma \vdash T_1$ **wf** and $\Gamma, X \vdash T_2$ **wf**, then $\Gamma \vdash T_2[T_1/X]$ **wf**

4.7 Parametric Polymorphism Extension

Adding *System F*-like parametric polymorphism to the core calculus of Section 4.2 is straightforward. However, combining it with the quote pattern matching type variables and bind pattern has some interesting consequences.

We extend the calculus of Section 4.6 with support for parametric polymorphism. To this end, we add type lambdas and type applications to the terms and the patterns. We also generalize the bind pattern to allow type-polymorphic bindings and enhance type unification.

Figures 4.8 to 4.10 extends Figures 4.1, 4.3, 4.4 and 4.6 to define the syntax and semantics of this extended calculus.

Syntax

We extend the calculus with two new syntax constructs: the type lambda $\Lambda X. t$ and the type application $t T$. In addition, we extend the syntax of the bind pattern to allow the capture of type variables \overline{X}_j^j defined in the pattern $\llbracket x \rrbracket_T^{\overline{X}_j^j \overline{x_k:T_k^k}}$. The syntax of types is extended to not only have type variables X , but also type abstractions $\forall X. T$.

Environment

Figure 4.6 from the calculus of Section 4.6 already introduced the necessary environment definitions.

Typing

Typing of (T-TABS) and (T-TAPP) are the usual typing rules from *System F*, with the extra tracking of the current staging level.

The rules (T-PAT-TABS) and (T-PAT-TAPP) mostly coincide with their typing counterparts. However, the Γ_p environment tracks any type binding added by a type lambda pattern (T-PAT-TABS). It is used for well-formedness of the pattern and to determine the free type variables in a bind pattern (T-PAT-BIND).

The rule (T-PAT-BIND) for the bind pattern $\llbracket x \rrbracket_T^{\overline{X}_j^j \overline{x_k:T_k^k}}$ now matches against an arbitrary expression locally closed under \overline{X}_j^j and $\overline{x_k:T_k^k}$. We represent this closed term as k curried lambdas taking arguments of the corresponding types \overline{T}_k^k nested in j curried type lambdas. Hence, in the output environment, we bind x to a value of type $\forall \overline{X}_j^j. \overline{[T_k]}^k \rightarrow [T]$.

Multi-Stage Macro Calculus

| |
|---|
| $ \begin{aligned} \text{Term } t & ::= \mathbf{c} \mid x \mid \lambda x:T.t \mid t t \mid \text{fix } t \mid \lceil t \rceil \mid \lfloor t \rfloor \\ & \quad \mid t \text{ match } \overline{X} \lceil t \rceil \text{ then } t \text{ else } t \mid \llbracket x \rrbracket_T^{\overline{X}_j^j \overline{x}_k:T_k^k} \\ & \quad \mid \Lambda X.t \mid t T \\ \text{Type } T & ::= \mathbf{C} \mid T \rightarrow T \mid \lceil T \rceil \mid X \mid \forall X.T \\ \text{Pattern bindings } \Phi & ::= \emptyset \mid \Phi, x \mapsto x \mid \Phi, X \mapsto X \end{aligned} $ |
| $ \frac{\Gamma, X \vdash^i t : T}{\Gamma \vdash^i \Lambda X.t : \forall X.T} \quad (\text{T-TABS}) \qquad \frac{\Gamma \vdash^i t : \forall X.T_2}{\Gamma \vdash^i t T_1 : T_2[T_1/X]} \quad (\text{T-TAPP}) $ |
| $ \frac{\Gamma \mid \Gamma_p, X \vdash^i t : T \dashv \Gamma_t}{\Gamma \mid \Gamma_p \vdash^i \Lambda X.t : \forall X.T \dashv \Gamma_t} \quad (\text{T-PAT-TABS}) \qquad \frac{\Gamma \mid \Gamma_p \vdash^i t : \forall X.T_2 \dashv \Gamma_t}{\Gamma \mid \Gamma_p \vdash^i t T_1 : T_2[T_1/X] \dashv \Gamma_t} \quad (\text{T-PAT-TAPP}) $ |
| $ \frac{\overline{X}_j \in \Gamma_p^j \quad \overline{x}_k : T_k \in \Gamma_p^k}{\Gamma \mid \Gamma_p \vdash^i \llbracket x \rrbracket_T^{\overline{X}_j^j \overline{x}_k:T_k^k} : T \dashv \emptyset, x : i^{-1} \forall \overline{X}_j^j \lceil T_k \rceil \rightarrow^k \lceil T \rceil} \quad (\text{T-PAT-BIND}) $ |
| $ \frac{t \rightarrow^i t'}{t T \rightarrow^i t' T} \quad (\text{E-TAPP}) \qquad \frac{t \rightarrow^i t' \quad i \geq 1}{\Lambda X.t \rightarrow^i \Lambda X.t'} \quad (\text{E-TABS}) $ |
| $ (\Lambda X.t) T \rightarrow^0 t[T/X] \quad (\text{E-TBETA}) $ |
| $ \frac{\emptyset \vdash t_s \sqsupset t_p \Rightarrow \sigma_1 \mid C \mid \overline{X}_l^l \quad \overline{X}_l^l \vdash \text{unify}(\overline{X}_l^l \mid C) \Rightarrow \sigma_2}{\overline{X}_l^l \vdash t_s \equiv t_p \Rightarrow \sigma_2 \circ \sigma_1} \quad (\text{E-PAT}) $ |
| $ \vdash^0 \Lambda X.t \mathbf{v} \mathbf{l} \quad (\text{V-TABS-0}) $ |
| $ \frac{\vdash^i t \mathbf{v} \mathbf{l} \quad i \geq 1}{\vdash^i \Lambda X.t \mathbf{v} \mathbf{l}} \quad (\text{V-TABS}) \qquad \frac{\vdash^i t \mathbf{v} \mathbf{l} \quad i \geq 1}{\vdash^i t T \mathbf{v} \mathbf{l}} \quad (\text{V-TAPP}) $ |

Figure 4.8: Parametric Polymorphism Extension

Operational Semantics

We extend the small-step semantics of the calculus. At level 0, the semantics follow the usual *System F* semantics, and we perform type β -reduction (E-TBETA). The rule E-TAPP expresses the usual congruence. Similarly to E-ABS, rule E-TABS needs to reduce at level 1 or above to reduce nested splices.

| | |
|---|---------------|
| $\Phi \vdash \mathbf{c} \sqsupset \mathbf{c} \Rightarrow [] \mid \emptyset \mid \emptyset$ | (E-PAT-CONST) |
| $\frac{\Phi \vdash t_{s_1} \sqsupset t_{p_1} \Rightarrow \sigma_1 \mid C_1 \mid \overline{X_{l_1}^{l_1}} \quad \Phi \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma_2 \mid C_2 \mid \overline{X_{l_2}^{l_2}}}{\Phi \vdash t_{s_1} t_{s_2} \sqsupset t_{p_1} t_{p_2} \Rightarrow \sigma_1 \circ \sigma_2 \mid C_1; C_2 \mid \overline{X_{l_1}^{l_1}}; \overline{X_{l_2}^{l_2}}}$ | (E-PAT-APP) |
| $\frac{\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X_l^l}}{\Phi \vdash \mathbf{fix} t_s \sqsupset \mathbf{fix} t_p \Rightarrow \sigma \mid C \mid \overline{X_l^l}}$ | (E-PAT-FIX) |
| $\Phi \vdash \Phi(x_p) \sqsupset x_p \Rightarrow [] \mid \emptyset \mid \emptyset$ | (E-PAT-VAR) |
| $\frac{\Phi, x_p \mapsto x_s \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X_l^l}}{\Phi \vdash \lambda x_s : T_1. t_s \sqsupset \lambda x_p : T_2. t_p \Rightarrow \sigma \mid C, T_1 = T_2 \mid \overline{X_l^l}}$ | (E-PAT-ABS) |
| $\frac{\Phi, X_p \mapsto X_s \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X_l^l}}{\Phi \vdash \Lambda X_s. t_s \sqsupset \Lambda X_p. t_p \Rightarrow \sigma \mid C \mid X_s / X_p \mid \overline{X_l^l}, X_s}$ | (E-PAT-TABS) |
| $\frac{\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X_l^l}}{\Phi \vdash t_s T_s \sqsupset t_p T_p \Rightarrow \sigma \mid C, T_s = T_p \mid \overline{X_l^l}}$ | (E-PAT-TAPP) |
| $\frac{FV(t_s) \cap \text{range}(\Phi) \subseteq \overline{\Phi(X_j)^j}; \overline{\Phi(x_k)^k} \quad t'_s = \overline{\Lambda X_j^j. \left(\lambda x'_k : [T_k]^k. [t_s] \mid [x'_k] / \Phi(x_k) \right) [X'_j / \Phi(X_j)]^j}}{\Phi \vdash t_s \sqsupset \llbracket x \rrbracket_T^{X_j^j x_k : T_k^k} \Rightarrow [t'_s / x] \mid \{type(t_s) = T\} \mid \emptyset}$ | (E-PAT-BIND) |

Figure 4.9: Pattern Semantics

The premise of the rule E-PAT-BIND is generalized from $FV(t_s) \cap \text{range}(\Phi) \subseteq \overline{\Phi(x_k)^k}$ to $FV(t_s) \cap \text{range}(\Phi) \subseteq \overline{\Phi(X_j)^j}; \overline{\Phi(x_k)^k}$. This implies that we will match a t_s that might contain references to types defined in the pattern. In this case, we cannot simply substitute t_s for x . We first need to bind all free variables and free type variables. In particular, for each free variable $\Phi(x_k)$ defined in the pattern, we η -expand t'_s by creating a lambda that receives a staged argument of type $x'_k : [T_k]^k$. Then, for each free type variable $\Phi(X_j)$ defined in the pattern, we type- η -expand by creating a type lambda that receives a replacement for that type. In the body of that lambda we substitute $\Phi(x_k)$ with the $[x'_k]$ and X_j with X'_j . This process results in a curried lambda of type $\overline{\forall X_j^j. [T'_k]^k \rightarrow [T]}$, where $T'_k = T_k[X'_j / \Phi(X_j)]^j$.

Just like (E-PAT-ABS), the rule (E-PAT-TABS) tracks the relationship between the binding in the pattern and the binding in the scrutinee in Φ as type mappings. The type variable mappings of Φ are only used for the bind pattern. Other relationships between types are made through the constraints. Rule E-PAT-TAPP is similar to E-PAT-APP but adds a type constraint.

| | |
|---|-------------|
| $\overline{X}_l^l \vdash \text{unify}(\emptyset \mid \emptyset) \Rightarrow []$ | (U-EMPTY) |
| $\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C, T=T) \Rightarrow \sigma}$ | (U-EQ) |
| $\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i; \overline{X}_j^j \mid C[T/X]) \Rightarrow \sigma \quad ftv(T) \cap \overline{X}_l^l = \emptyset}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i, X, \overline{X}_j^j \mid C, T=X) \Rightarrow [T/X] \circ \sigma}$ | (U-PAT-VAR) |
| $\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C, T_{s_1}=T_{p_1}, T_{s_2}=T_{p_2}) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C, T_{s_1} \rightarrow T_{s_2}=T_{p_1} \rightarrow T_{p_2}) \Rightarrow \sigma}$ | (U-ABS) |
| $\frac{\overline{X}_l^l, X_1 \vdash \text{unify}(\overline{X}_i^i \mid C, T_1=(T_2[X_1/X_2])) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C, \forall X_1. T_1=\forall X_2. T_2) \Rightarrow \sigma}$ | (U-TABS) |
| $\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C, T_s=T_p) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C, [T_s]=[T_p]) \Rightarrow \sigma}$ | (U-QUOTE) |

Figure 4.10: Pattern Unification Semantics

E-PAT-TABS may introduce new free type variables in the constraints. These should not escape through the substitution created by the type unification. To ensure this, we track these variables \overline{X}_l^l along with constraints in $\Phi \vdash t_s \sqsupseteq t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l$. Then we pass them to the unification $\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C) \Rightarrow \sigma$ in E-PAT. U-PAT-VAR has the added premise $ftv(T) \cap \overline{X}_l^l = \emptyset$ to ensure that the unification only succeeds if none of those variables are present in the substitution. We also need a U-TABS rule to unify $\forall X.T$ with each other. To make the constraints comparable, we substitute one of the variables with the other. We keep the variable that is defined in the scrutinee. As this is also a local variable that is added to the constraint, we add it to the \overline{X}_l^l .

Example 13. There are two interesting interactions between type variables introduced by a match and those introduced in type abstraction patterns.

$$\dots \text{match } X_1 \text{ } [\Lambda X_2. \llbracket x \rrbracket_{X_1}] \text{ then } \dots \text{ else } \dots \quad (1)$$

$$\dots \text{match } X_1 \text{ } [\Lambda X_2. \llbracket x \rrbracket_{X_1}^{X_2}] \text{ then } \dots \text{ else } \dots \quad (2)$$

In the case of Eq. (1) we have a closed bind pattern, and U-PAT-VAR ensures that the pattern will only match if the term x does not contain references to X_2 . In the case of Eq. (2) we can refer to X_2 in x , and the unification may result in a type containing a reference to X_2 . The rule T-PAT-BIND will replace this X_2 with the well-scoped X_2' .

Values

The term values for type lambdas and type applications follow the same rules as normal lambdas and applications.

Soundness

The statements of the progress and preservation theorems carry over unchanged from Section 4.6. We only need to add the usual new cases in the proof for type abstraction and type application.

To prove Lemma 6 with the additional \overline{X}_l^l tracking local pattern variables, we need to modify slightly Lemmas 5, 10 and 11. We replace Lemma 10 with Lemma 16 to add the extra \overline{X}_l^l with the local pattern type variables to make the constraint well formed. We also need to add cases to cover T-PAT-TAPP and T-PAT-TABS, which follow similar proofs to T-PAT-APP and T-PAT-ABS. The case for T-PAT-BIND is extended to account for the captured type variables and type lambdas.

Lemma 16 (Constraints of Pattern Reduction).

If Eqs. (1) to (4) then $\Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C$ **wf**.

$$\Gamma; \Gamma_\delta \vdash^1 t_s : T_s \quad (1)$$

$$\Gamma; \overline{X}_i^i \mid \Gamma_p \vdash^1 t_p : T_p \dashv \Gamma_t \quad (2)$$

$$\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l \quad (3)$$

$$\Gamma_p \mid \Gamma_\delta \vdash \Phi \text{ **wf**} \quad (4)$$

We also replace Lemma 11 with Lemma 17 to add the extra \overline{X}_l^l . This environment is used in U-PAT-VAR to ensure that the unification only succeeds if none of those variables are present in the substitution. To prove this case we use Lemma 18. We also need to add a proof case for the new U-TABS rule.

Lemma 17 (Pattern Constraints Unification).

If $\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C) \Rightarrow \sigma$ and $\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C$ **wf** and $\Gamma \vdash T$ **wf** and $\Gamma; \overline{X}_i^i \vdash^0 t : T$, then $\Gamma \vdash^0 \sigma(t) : T$

Lemma 18 (Unification Locality).

If $\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C) \Rightarrow \sigma$ and $X \in \overline{X}_l^l$, then $X \notin \text{image}(\sigma)$

To restate the Lemma 5 we only need to add the \overline{X}_l^l in $\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l$. To prove it, we need to add the two cases for T-PAT-TAPP and T-PAT-TABS, which follow similar proofs to T-PAT-APP and T-PAT-ABS. Finally, we need to change the proof case for the T-PAT-BIND to account for captured local variables.

4.8 Polymorphic Multi-Stage Macro Calculus

In this section, we present the full *polymorphic multi-stage macro calculus*. It contains the unification of the *multi-stage macro calculus* λ^\blacktriangle , *polymorphic multi-stage calculus* F^\blacktriangle and the quoted constants extension of Section 4.3. Appendix A contains the complete soundness proofs of the calculus.

4.8.1 Syntax

Figure 4.11 shows the syntax of the calculus.

The calculus features syntax for program such as library bindings `def $x = [t]$ in p` and program evaluation `eval t` from Section 4.5. The calculus also features syntax for terms such as constants c , variables x , abstraction $\lambda x:T.t$, applications $t t$, fixpoint computations `fix t` , quotations `[t]` and splices `[t]` from Section 4.2. It includes the operations on quoted constants `lift t` and `unlift t with t or t` from Section 4.3. It contains type abstraction $\Lambda X.t$ and type application $t T$ from Section 4.7. It has quote pattern matching `t_s match \bar{X} [t_p] then t_t else t_e` as defined in Section 4.6 and `$\llbracket x \rrbracket_T^{\bar{X}_j^j \bar{x}_k:T_k^k}$` as defined in Section 4.7.

The syntax of types includes built-in types \mathbf{C} , function types of the form $T \rightarrow T$, and the type of quoted terms `[T]` from Section 4.2. It also includes type variables X from Sections 4.6 and 4.7 and type lambdas $\forall X.T$ from Section 4.7.

| |
|--|
| $\begin{aligned} \text{Program } p & ::= \text{def } x = [t] \text{ in } p \mid \text{eval } t \\ \\ \text{Term } t & ::= c \mid x \mid \lambda x:T.t \mid t t \mid \text{fix } t \mid [t] \mid [t] \\ & \quad \mid \text{lift } t \mid \text{unlift } t \text{ with } t \text{ or } t \\ & \quad \mid \Lambda X.t \mid t T \\ & \quad \mid t \text{ match } \bar{X} [t] \text{ then } t \text{ else } t \mid \llbracket x \rrbracket_T^{\bar{X}_j^j \bar{x}_k:T_k^k} \\ \\ \text{Type } T & ::= \mathbf{C} \mid T \rightarrow T \mid [T] \mid X \mid \forall X.T \end{aligned}$ |
|--|

Figure 4.11: Syntax

4.8.2 Environments

Figure 4.12 shows the environment definitions.

Environments Γ are lists of bindings $x :^i T$ as described in Section 4.2 and type variables X as described in Section 4.6. An environment Γ is well formed if $\vdash \Gamma$ **wf** as defined in Figure 4.13. A type T is well formed under an environment Γ if $\Gamma \vdash T$ **wf** as defined in Figure 4.14. A staging level i is defined as a non-negative integer as defined in Figure 4.1.

Pattern bindings Φ are lists of mappings of term names $x \rightarrow x$ as defined in Section 4.4 or mappings of type names $X \rightarrow X$ from Section 4.7. A Φ is well formed with respect to Γ_p and Γ_δ if $\Gamma_p \mid \Gamma_\delta \vdash \Phi$ **wf** as stated in Definition A.3.

Library typings Σ are lists of bindings $x : T$ and global stores Ω are lists of $x := t$ as described in Section 4.5. A store Ω is well typed with respect to Σ if $\Sigma \vdash \Omega$ **wf** as stated in Definition A.1.

Constraints are lists of equivalences between types $T = T$ as defined in Section 4.6. A constraint C is well formed under an environment Γ and pattern type variables \bar{X} if $\Gamma \mid \bar{X} \vdash C$ **wf** as defined in Figure 4.15.

| | | | |
|-----------------------------|--------------|-----------|---|
| <i>Typing environment</i> | Γ | $::=$ | $\emptyset \mid \Gamma, x :^i T \mid \Gamma, X$ |
| <i>Level</i> | i | \in | \mathbb{N}_0 |
| <i>Pattern bindings</i> | Φ | $::=$ | $\emptyset \mid \Phi, x \rightarrow x, X \rightarrow X$ |
| <i>Library typing</i> | Σ | $::=$ | $\emptyset \mid \Sigma, x : T$ |
| <i>Global store</i> | Ω | $::=$ | $\emptyset \mid \Omega, x := t$ |
| <i>Constraints</i> | C | $::=$ | $\emptyset \mid C, T = T$ |

Figure 4.12: Environments

$$\begin{array}{c}
 \vdash \emptyset \mathbf{wf} \quad (\text{WFE-EMPTY}) \\
 \\
 \frac{\Gamma \vdash T \mathbf{wf} \quad x \notin \text{dom}(\Gamma)}{\vdash \Gamma; x :^i T \mathbf{wf}} \quad (\text{WFE-VAR}) \qquad \frac{\vdash \Gamma \mathbf{wf} \quad X \notin \text{dom}(\Gamma)}{\vdash \Gamma; X \mathbf{wf}} \quad (\text{WFE-TVAR})
 \end{array}$$

Figure 4.13: Well-Formed Environment

$$\begin{array}{c}
 \frac{\Gamma \vdash T_1 \mathbf{wf} \quad \Gamma \vdash T_2 \mathbf{wf}}{\Gamma \vdash T_1 \rightarrow T_2 \mathbf{wf}} \quad (\text{WFT-ABS}) \qquad \frac{\vdash \Gamma \mathbf{wf}}{\Gamma \vdash \mathbf{C} \mathbf{wf}} \quad (\text{WFT-CONST}) \\
 \\
 \frac{\Gamma, X \vdash T \mathbf{wf}}{\Gamma \vdash \forall X. T \mathbf{wf}} \quad (\text{WFT-TABS}) \qquad \frac{\Gamma \vdash T \mathbf{wf}}{\Gamma \vdash [T] \mathbf{wf}} \quad (\text{WFT-QUOTED}) \\
 \\
 \frac{\vdash \Gamma \mathbf{wf} \quad X \in \Gamma}{\Gamma \vdash X \mathbf{wf}} \quad (\text{WFT-VAR})
 \end{array}$$

Figure 4.14: Well-Formed Type

$$\begin{array}{c}
 \frac{\vdash \Gamma; \bar{X} \mathbf{wf}}{\Gamma | \bar{X} \vdash \emptyset \mathbf{wf}} \quad (\text{WFC-EMPTY}) \\
 \\
 \frac{\Gamma \vdash T_1 \mathbf{wf} \quad \Gamma; \bar{X} \vdash T_2 \mathbf{wf} \quad \Gamma | \bar{X} \vdash C \mathbf{wf}}{\Gamma | \bar{X} \vdash C, T_1 = T_2 \mathbf{wf}} \quad (\text{WFC-EQ})
 \end{array}$$

Figure 4.15: Well-Formed Constraint

4.8.3 Typing

Figures 4.16 to 4.18 show the typing rules.

Program typing is performed with T-EVAL and T-DEF as described in Section 4.5.

Term typing for T-CONST, T-VAR, T-ABS, T-APP T-QUOTE and T-SPLICE is performed as described in Section 4.2. Typing for T-LIFT and T-UNLIFT is performed as described in Section 4.3. Typing for T-LINK is performed as described in Section 4.5. Typing for T-MATCH is performed as described in Section 4.6. Term typing for T-TABS and T-TAPP is performed as described in Section 4.7. We add the Σ environment from Section 4.5 to all the typing rules.

Pattern typing for T-PAT-CONST, T-PAT-VAR, T-PAT-ABS T-PAT-APP and T-PAT-FIX is performed as described in Section 4.4. Typing for T-PAT-TAPP, T-PAT-TABS and T-PAT-BIND is performed as described in Section 4.7. Typing for T-PAT-LINK is performed as described in Section 4.5. We add the Σ environment from Section 4.5 and Γ environment from Section 4.6 to all the typing rules.

| | |
|---|----------|
| $\frac{\Sigma \emptyset \vdash^0 t : T}{\Sigma \vdash \mathbf{eval} \ t : T}$ | (T-EVAL) |
| $\frac{\Sigma \emptyset \vdash^1 t : T_1 \quad \Sigma, x : T_1 \vdash p : T_2}{\Sigma \vdash \mathbf{def} \ x = [t] \ \mathbf{in} \ p : T_2}$ | (T-DEF) |

Figure 4.16: Program Typing

| | |
|---|--|
| | $\Sigma \Gamma \vdash^i \mathbf{c} : \mathbf{C}$ (T-CONST) |
| $\frac{x : ^i T \in \Gamma}{\Sigma \Gamma \vdash^i x : T}$ (T-VAR) | $\frac{x : T \in \Sigma}{\Sigma \Gamma \vdash^i x : T}$ (T-LINK) |
| $\frac{\Sigma \Gamma, x : ^i T_1 \vdash^i t_2 : T_2}{\Sigma \Gamma \vdash^i \lambda x : T_1. t_2 : T_1 \rightarrow T_2}$ (T-ABS) | $\frac{\Sigma \Gamma \vdash^i t_1 : T_1 \rightarrow T_2 \quad \Sigma \Gamma \vdash^i t_2 : T_1}{\Sigma \Gamma \vdash^i t_1 t_2 : T_2}$ (T-APP) |
| $\frac{\Sigma \Gamma, X \vdash^i t : T}{\Sigma \Gamma \vdash^i \Lambda X. t : \forall X. T}$ (T-TABS) | $\frac{\Sigma \Gamma \vdash^i t : \forall X. T_2}{\Sigma \Gamma \vdash^i t T_1 : T_2 [T_1 / X]}$ (T-TAPP) |
| $\frac{\Sigma \Gamma \vdash^{i+1} t : T}{\Sigma \Gamma \vdash^i [t] : [T]}$ (T-QUOTE) | $\frac{\Sigma \Gamma \vdash^{i-1} t : [T] \quad i \geq 1}{\Sigma \Gamma \vdash^i [t] : T}$ (T-SPLICE) |
| $\frac{\Sigma \Gamma \vdash^i t : \mathbf{C}}{\Sigma \Gamma \vdash^i \mathbf{lift} t : [\mathbf{C}]}$ (T-LIFT) | $\frac{\Sigma \Gamma \vdash^i t : T \rightarrow T}{\Sigma \Gamma \vdash^i \mathbf{fix} t : T}$ (T-FIX) |
| $\frac{\Sigma \Gamma \vdash^i t_1 : [\mathbf{C}] \quad \Sigma \Gamma \vdash^i t_2 : \mathbf{C} \rightarrow T \quad \Sigma \Gamma \vdash^i t_3 : T}{\Sigma \Gamma \vdash^i \mathbf{unlift} t_1 \mathbf{with} t_2 \mathbf{or} t_3 : T}$ (T-UNLIFT) | |
| $\frac{\Sigma \Gamma \vdash^i t_s : [T_p] \quad \Sigma \Gamma; \bar{X} \emptyset \vdash^{i+1} t_p : T_p \dashv \Gamma_t \quad \Sigma \Gamma; \bar{X}; \Gamma_t \vdash^i t_t : T \quad \Sigma \Gamma \vdash^i t_e : T}{\Sigma \Gamma \vdash^i t_s \mathbf{match} \bar{X} [t_p] \mathbf{then} t_t \mathbf{else} t_e : T}$ (T-MATCH) | |

Figure 4.17: Term Typing

| | |
|--|---------------|
| $\frac{}{\Sigma \Gamma \Gamma_p \vdash^i \mathbf{c} : \mathbf{C} \dashv \emptyset}$ | (T-PAT-CONST) |
| $\frac{x : {}^i T \in \Gamma_p}{\Sigma \Gamma \Gamma_p \vdash^i x : T \dashv \emptyset}$ | (T-PAT-VAR) |
| $\frac{x : T \in \Sigma}{\Sigma \Gamma \Gamma_p \vdash^i x : T \dashv \emptyset}$ | (T-PAT-LINK) |
| $\frac{\Sigma \Gamma \Gamma_p, x : {}^i T_1 \vdash^i t : T_2 \dashv \Gamma_t}{\Sigma \Gamma \Gamma_p \vdash^i \lambda x : T_1. t : T_1 \rightarrow T_2 \dashv \Gamma_t}$ | (T-PAT-ABS) |
| $\frac{\Sigma \Gamma \Gamma_p \vdash^i t_1 : T_1 \rightarrow T_2 \dashv \Gamma_{t_1} \quad \Sigma \Gamma \Gamma_p \vdash^i t_2 : T_1 \dashv \Gamma_{t_2}}{\Sigma \Gamma \Gamma_p \vdash^i t_1 t_2 : T_2 \dashv \Gamma_{t_1}; \Gamma_{t_2}}$ | (T-PAT-APP) |
| $\frac{\Sigma \Gamma \Gamma_p \vdash^i t : T \rightarrow T \dashv \Gamma_t}{\Sigma \Gamma \Gamma_p \vdash^i \mathbf{fix} t : T \dashv \Gamma_t}$ | (T-PAT-FIX) |
| $\frac{\Sigma \Gamma \Gamma_p, X \vdash^i t : T \dashv \Gamma_t}{\Sigma \Gamma \Gamma_p \vdash^i \Lambda X. t : \forall X. T \dashv \Gamma_t}$ | (T-PAT-TABS) |
| $\frac{\Sigma \Gamma \Gamma_p \vdash^i t : \forall X. T_2 \dashv \Gamma_t}{\Sigma \Gamma \Gamma_p \vdash^i t T_1 : T_2[T_1/X] \dashv \Gamma_t}$ | (T-PAT-TAPP) |
| $\frac{\overline{X_j \in \Gamma_p}^j \quad \overline{x_k : {}^i T_k \in \Gamma_p}^k}{\Sigma \Gamma \Gamma_p \vdash^i \llbracket x \rrbracket_T^{\overline{X_j}^j \overline{x_k : T_k}^k} : T \dashv \emptyset, x : {}^{i-1} \overline{\forall X_j.}^j \overline{[T_k] \rightarrow}^k [T]}$ | (T-PAT-BIND) |

Figure 4.18: Pattern Typing

4.8.4 Operational Semantics

Figures 4.19 to 4.24 show the operational semantics.

Program evaluation E-EVAL, E-MACRO and E-COMPILE is performed as described in Section 4.5.

Term evaluation E-APP-1, E-APP-2, E-BETA, E-ABS, E-FIX, E-FIX-RED, E-QUOTE, E-SPLICE and E-SPLICE-RED is performed as described in Section 4.2. Type abstraction evaluation E-TAPP, E-TBETA and E-TABS is performed as described in Section 4.7. Reference to global definition evaluation E-LINK is performed as described in Section 4.5. Quoted constant evaluation E-LIFT, E-LIFT-CONST, E-UNLIFT-SCRUT, E-UNLIFT-SUCC, E-UNLIFT-FAIL, E-UNLIFT-WITH and E-UNLIFT-OR is performed as described in Section 4.3. Pattern match evaluation E-MATCH-SUCC, E-MATCH-FAIL, E-MATCH-SCRUT, E-MATCH-THEN and E-MATCH-ELSE is performed as described in Section 4.4 with the additional \bar{X} introduced in Section 4.6.

Pattern evaluation E-PAT is performed as described in Section 4.7, which generalizes the semantics presented in Sections 4.4 and 4.6. Structural matching for E-PAT-CONST, E-PAT-VAR, E-PAT-APP, E-PAT-ABS E-PAT-FIX is performed as described in Section 4.4. Structural matching for E-PAT-LINK is performed as described in Section 4.5. Structural matching for E-PAT-TABS and E-PAT-TAPP as performed as described in Section 4.7. Structural matching for E-PAT-BIND is performed as described in Section 4.7, which generalizes the semantics presented in Section 4.4. For all rules except E-PAT-LINK, constraints are collected as described in Section 4.6. We collect empty constraints for E-PAT-LINK following the same reasoning as E-PAT-VAR. Likewise, local type variables are collected as described in Section 4.7. We collect empty local type variables for E-PAT-LINK following the same reasoning of E-PAT-VAR. Unification of constraints U-EMPTY, U-EQ, U-PAT-VAR, U-ABS, U-TABS and U-QUOTE is performed as described in Section 4.7, which generalizes the semantics introduced in Section 4.6.

| | |
|--|-------------|
| $\frac{t \xrightarrow{\Omega}^0 t'}{\text{eval } t \mid \Omega \longrightarrow \text{eval } t' \mid \Omega}$ | (E-EVAL) |
| $\frac{t \xrightarrow{\Omega}^1 t'}{\text{def } x = [t] \text{ in } p \mid \Omega \longrightarrow \text{def } x = [t'] \text{ in } p \mid \Omega}$ | (E-MACRO) |
| $\frac{\vdash^1 t \text{ vl}}{\text{def } x = [t] \text{ in } p \mid \Omega \longrightarrow p \mid \Omega, x := t}$ | (E-COMPILE) |

Figure 4.19: Program Operational Semantics

| | |
|--|---|
| $\frac{t_1 \longrightarrow_{\Omega}^i t'_1}{t_1 t_2 \longrightarrow_{\Omega}^i t'_1 t_2} \quad (\text{E-APP-1})$ | $\frac{\vdash^i t_1 \mathbf{vl} \quad t_2 \longrightarrow_{\Omega}^i t'_2}{t_1 t_2 \longrightarrow_{\Omega}^i t_1 t'_2} \quad (\text{E-APP-2})$ |
| $\frac{\vdash^0 t_2 \mathbf{vl}}{(\lambda x:T_1.t_1) t_2 \longrightarrow_{\Omega}^0 t_1[t_2/x]} \quad (\text{E-BETA})$ | $\frac{t \longrightarrow_{\Omega}^i t' \quad i \geq 1}{\lambda x:T.t \longrightarrow_{\Omega}^i \lambda x:T.t'} \quad (\text{E-ABS})$ |
| $\frac{x \in \text{dom}(\Omega)}{x \longrightarrow_{\Omega}^0 \Omega(x)} \quad (\text{E-LINK})$ | $\frac{t \longrightarrow_{\Omega}^i t'}{\mathbf{fix} t \longrightarrow_{\Omega}^i \mathbf{fix} t'} \quad (\text{E-FIX})$ |
| $\mathbf{fix} \lambda x:T.t \longrightarrow_{\Omega}^0 t[\mathbf{fix} \lambda x:T.t/x] \quad (\text{E-FIX-RED})$ | |
| $\frac{t \longrightarrow_{\Omega}^{i+1} t'}{[t] \longrightarrow_{\Omega}^i [t']} \quad (\text{E-QUOTE})$ | |
| $\frac{t \longrightarrow_{\Omega}^{i-1} t' \quad i \geq 1}{[t] \longrightarrow_{\Omega}^i [t']} \quad (\text{E-SPLICE})$ | $\frac{\vdash^{-1} t \mathbf{vl}}{[[t]] \longrightarrow_{\Omega}^{-1} t} \quad (\text{E-SPLICE-RED})$ |
| $\frac{t \longrightarrow_{\Omega}^i t'}{t T \longrightarrow_{\Omega}^i t' T} \quad (\text{E-TAPP})$ | $(\Lambda X.t) T \longrightarrow_{\Omega}^0 t[T/X] \quad (\text{E-TBETA})$ |
| $\frac{t \longrightarrow_{\Omega}^i t' \quad i \geq 1}{\Lambda X.t \longrightarrow_{\Omega}^i \Lambda X.t'} \quad (\text{E-TABS})$ | |
| $\frac{t \longrightarrow_{\Omega}^i t'}{\mathbf{lift} t \longrightarrow_{\Omega}^i \mathbf{lift} t'} \quad (\text{E-LIFT})$ | $\mathbf{lift} \mathbf{c} \longrightarrow_{\Omega}^0 [\mathbf{c}] \quad (\text{E-LIFT-CONST})$ |

Figure 4.20: Term Operational Semantics (a)

| | |
|--|------------------|
| $\frac{t_1 \longrightarrow_{\Omega}^i t'_1}{\text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \longrightarrow_{\Omega}^i \text{unlift } t'_1 \text{ with } t_2 \text{ or } t_3}$ | (E-UNLIFT-SCRUT) |
| $\text{unlift } [\mathbf{c}] \text{ with } t_2 \text{ or } t_3 \longrightarrow_{\Omega}^0 t_2 \mathbf{c}$ | (E-UNLIFT-SUCC) |
| $\frac{\vdash^0 t_1 \mathbf{vl} \quad t_1 \neq [\mathbf{c}]}{\text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \longrightarrow_{\Omega}^0 t_3}$ | (E-UNLIFT-FAIL) |
| $\frac{\vdash^i t_1 \mathbf{vl} \quad t_2 \longrightarrow_{\Omega}^i t'_2 \quad i \geq 1}{\text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \longrightarrow_{\Omega}^i \text{unlift } t_1 \text{ with } t'_2 \text{ or } t_3}$ | (E-UNLIFT-WITH) |
| $\frac{\vdash^i t_1 \mathbf{vl} \quad \vdash^i t_2 \mathbf{vl} \quad t_3 \longrightarrow_{\Omega}^i t'_3 \quad i \geq 1}{\text{unlift } t_1 \text{ with } t_2 \text{ or } t_3 \longrightarrow_{\Omega}^i \text{unlift } t_1 \text{ with } t_2 \text{ or } t'_3}$ | (E-UNLIFT-OR) |
| $\frac{\vdash^1 t_s \mathbf{vl} \quad \bar{X} \vdash t_s \equiv t_p \Rightarrow \sigma}{[t_s] \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t_e \longrightarrow_{\Omega}^0 \sigma(t_t)}$ | (E-MATCH-SUCC) |
| $\frac{\vdash^1 t_s \mathbf{vl} \quad \bar{X} \vdash t_s \equiv t_p \not\Rightarrow \sigma}{[t_s] \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t_e \longrightarrow_{\Omega}^0 t_e}$ | (E-MATCH-FAIL) |
| $\frac{t_s \longrightarrow_{\Omega}^i t'_s}{t_s \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t_e \longrightarrow_{\Omega}^i t'_s \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t_e}$ | (E-MATCH-SCRUT) |
| $\frac{\vdash^0 t_s \mathbf{vl} \quad t_t \longrightarrow_{\Omega}^i t'_t \quad i \geq 1}{t_s \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t_e \longrightarrow_{\Omega}^i t_s \text{ match } \bar{X} [t_p] \text{ then } t'_t \text{ else } t_e}$ | (E-MATCH-THEN) |
| $\frac{\vdash^0 t_s \mathbf{vl} \quad \vdash^0 t_t \mathbf{vl} \quad t_e \longrightarrow_{\Omega}^i t'_e \quad i \geq 1}{t_s \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t_e \longrightarrow_{\Omega}^i t_s \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t'_e}$ | (E-MATCH-ELSE) |

Figure 4.21: Term Operational Semantics (b)

$$\frac{\emptyset \vdash t_s \sqsupset t_p \Rightarrow \sigma_1 \mid C \mid \overline{X}_l^l \quad \overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C) \Rightarrow \sigma_2}{\overline{X}_i^i \vdash t_s \equiv t_p \Rightarrow \sigma_2 \circ \sigma_1} \quad (\text{E-PAT})$$

Figure 4.22: Pattern Semantics

$$\begin{array}{l} \Phi \vdash \mathbf{c} \sqsupset \mathbf{c} \Rightarrow [] \mid \emptyset \mid \emptyset \quad (\text{E-PAT-CONST}) \qquad \Phi \vdash x \sqsupset x \Rightarrow [] \mid \emptyset \mid \emptyset \quad (\text{E-PAT-LINK}) \\ \\ \Phi \vdash \Phi(x_p) \sqsupset x_p \Rightarrow [] \mid \emptyset \mid \emptyset \quad (\text{E-PAT-VAR}) \\ \\ \frac{\Phi \vdash t_{s_1} \sqsupset t_{p_1} \Rightarrow \sigma_1 \mid C_1 \mid \overline{X}_{l_1}^{l_1} \quad \Phi \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma_2 \mid C_2 \mid \overline{X}_{l_2}^{l_2}}{\Phi \vdash t_{s_1} t_{s_2} \sqsupset t_{p_1} t_{p_2} \Rightarrow \sigma_1 \circ \sigma_2 \mid C_1; C_2 \mid \overline{X}_{l_1}^{l_1}; \overline{X}_{l_2}^{l_2}} \quad (\text{E-PAT-APP}) \\ \\ \frac{\Phi, x_p \mapsto x_s \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l}{\Phi \vdash \lambda x_s : T_1. t_s \sqsupset \lambda x_p : T_2. t_p \Rightarrow \sigma \mid C, T_1 = T_2 \mid \overline{X}_l^l} \quad (\text{E-PAT-ABS}) \\ \\ \frac{\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l}{\Phi \vdash t_s T_s \sqsupset t_p T_p \Rightarrow \sigma \mid C, T_s = T_p \mid \overline{X}_l^l} \quad (\text{E-PAT-TAPP}) \\ \\ \frac{\Phi, X_p \mapsto X_s \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l}{\Phi \vdash \Lambda X_s. t_s \sqsupset \Lambda X_p. t_p \Rightarrow \sigma \mid C[X_s/X_p] \mid \overline{X}_l^l, X_s} \quad (\text{E-PAT-TABS}) \\ \\ \frac{\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l}{\Phi \vdash \mathbf{fix} t_s \sqsupset \mathbf{fix} t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l} \quad (\text{E-PAT-FIX}) \\ \\ \frac{FV(t_s) \cap \text{range}(\Phi) \subseteq \overline{\Phi(X_j)^j}; \overline{\Phi(x_k)^k} \quad t'_s = \Lambda \overline{X}_j^j. \left(\lambda x'_k : [T_k]^k. [t_s][[\overline{x'_k}/\Phi(x_k)]^k] \right) [\overline{X'_j}/\Phi(X_j)]^j}{\Phi \vdash t_s \sqsupset \llbracket x \rrbracket_T^{\overline{X}_j^j x_k : T_k^k} \Rightarrow [t'_s/x] \mid \{type(t_s) = T\} \mid \emptyset} \quad (\text{E-PAT-BIND}) \end{array}$$

Figure 4.23: Pattern Structural Matching

| | |
|--|-------------|
| $\overline{X}_l^l \vdash \text{unify}(\emptyset \emptyset) \Rightarrow []$ | (U-EMPTY) |
| $\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i C) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i C, T=T) \Rightarrow \sigma}$ | (U-EQ) |
| $\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i; \overline{X}_j^j C[T/X]) \Rightarrow \sigma \quad \text{ftv}(T) \cap \overline{X}_l^l = \emptyset}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i, X, \overline{X}_j^j C, T=X) \Rightarrow [T/X] \circ \sigma}$ | (U-PAT-VAR) |
| $\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i C, T_{s_1}=T_{p_1}, T_{s_2}=T_{p_2}) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i C, T_{s_1} \rightarrow T_{s_2}=T_{p_1} \rightarrow T_{p_2}) \Rightarrow \sigma}$ | (U-ABS) |
| $\frac{\overline{X}_l^l, X_1 \vdash \text{unify}(\overline{X}_i^i C, T_1=(T_2[X_1/X_2])) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i C, \forall X_1. T_1=\forall X_2. T_2) \Rightarrow \sigma}$ | (U-TABS) |
| $\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i C, T_s=T_p) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i C, [T_s]=[T_p]) \Rightarrow \sigma}$ | (U-QUOTE) |

Figure 4.24: Pattern Type Unification

4.8.5 Values

Figures 4.25 and 4.26 show the value definitions.

V-EVAL is the only value for programs p as described in Section 4.5. Terms can be values for V-CONST, V-VAR, V-ABS-0, V-ABS, V-APP, V-QUOTE, V-SPLICE and V-FIX as described in Section 4.2. Term values V-LIFT and V-UNLIFT are described in Section 4.3. Term values V-TABS-0, V-TABS and V-TAPP are described in Section 4.7. The term value for V-MATCH follows the description in Section 4.4 but uses the extra \bar{X} added in Section 4.6.

$$\frac{\vdash^0 t \mathbf{vl}}{\vdash \mathbf{eval} t \mathbf{vl}} \quad (\text{V-EVAL})$$

Figure 4.25: Program Values

| | | | |
|---|------------|---|------------|
| $\vdash^i \mathbf{c} \mathbf{vl}$ | (V-CONST) | $\frac{i \geq 1}{\vdash^i x \mathbf{vl}}$ | (V-VAR) |
| $\vdash^0 \lambda x:T.t \mathbf{vl}$ | (V-ABS-0) | $\frac{\vdash^i t \mathbf{vl} \quad i \geq 1}{\vdash^i \lambda x:T.t \mathbf{vl}}$ | (V-ABS) |
| $\vdash^0 \Lambda X.t \mathbf{vl}$ | (V-TABS-0) | $\frac{\vdash^i t \mathbf{vl} \quad i \geq 1}{\vdash^i \Lambda X.t \mathbf{vl}}$ | (V-TABS) |
| $\frac{\vdash^i t_1 \mathbf{vl} \quad \vdash^i t_2 \mathbf{vl} \quad i \geq 1}{\vdash^i t_1 t_2 \mathbf{vl}}$ | (V-APP) | $\frac{\vdash^i t \mathbf{vl} \quad i \geq 1}{\vdash^i t T \mathbf{vl}}$ | (V-TAPP) |
| $\frac{\vdash^{i+1} t \mathbf{vl}}{\vdash^i [t] \mathbf{vl}}$ | (V-QUOTE) | $\frac{\vdash^{i-1} t \mathbf{vl} \quad i \geq 2}{\vdash^i [t] \mathbf{vl}}$ | (V-SPLICE) |
| $\frac{\vdash^i t \mathbf{vl} \quad i \geq 1}{\vdash^i \mathbf{lift} t \mathbf{vl}}$ | (V-LIFT) | $\frac{\vdash^i t \mathbf{vl} \quad i \geq 1}{\vdash^i \mathbf{fix} t \mathbf{vl}}$ | (V-FIX) |
| $\frac{\vdash^i t_1 \mathbf{vl} \quad \vdash^i t_2 \mathbf{vl} \quad \vdash^i t_3 \mathbf{vl} \quad i \geq 1}{\vdash^i \mathbf{unlift} t_1 \text{ with } t_2 \text{ or } t_3 \mathbf{vl}}$ | | (V-UNLIFT) | |
| $\frac{\vdash^i t_s \mathbf{vl} \quad \vdash^i t_t \mathbf{vl} \quad \vdash^i t_e \mathbf{vl} \quad i \geq 1}{\vdash^i t_s \mathbf{match} \bar{X} [t_p] \text{ then } t_t \text{ else } t_e \mathbf{vl}}$ | | (V-MATCH) | |

Figure 4.26: Term Values

4.9 Concrete Syntax in Scala

The static checks as performed by the type-checker closely follow the rules of the calculus as described in this chapter. The choice of syntax for quotes $\{t\}$ and for splices $\$\{t\}$ follows the standard syntax rules of Scala's string interpolators `s"hello $world"` or `s"hello $\{\text{world}\}$ "` where `world` is spliced in the string. To lift or unlift values, we use the `Expr(x)` syntax which corresponds to `Expr.apply` in expression position and to `Expr.unapply` in pattern position. Global library function definitions such as

```
def x = [f 3] in
def y = [ $\lambda a:N.a$ ] in
def z = [fix  $\lambda rec:N.rec$ ] in ...
```

can be expressed in Scala as

```
def x: Int = f(3)
def y: Int => Int = (a: Int) => a
def z: Int = ...
```

To support quoted pattern matching, we extend the pattern syntax to allow the pattern $\{pat\}$. For example, the expression

```
s match [f [ $\llbracket x \rrbracket_N$ ]] then t else e
```

can be expressed in Scala as

```
s match
  case '{ f($x: Int) } => t
  case _ => e
```

where `f` is a global library reference and `$x` represents a locally closed bind pattern. Free variables in bind patterns, as in $\llbracket x \rrbracket_N^{y:N}$, can be specified as `$x(y) : Int`. The following term in *System F^Δ* that binds the body of a lambda to `x` and might have `y` as a free variable

```
s match [ $\lambda y:N.\llbracket x \rrbracket_N^{y:N}$ ] then t else e
```

can be expressed in Scala as

```
s match
  case '{ (y: Int) => $x(y):Int } => t
  case _ => e
```

Binding an applied function in a pattern

$$s \text{ match } [\llbracket f \rrbracket_{N \rightarrow N} 3] \text{ then } t \text{ else } e$$

can be expressed in Scala as

```
s match
  case '{ ($f: Int=>Int)(3) } => t
  case _ => e
```

Finally, patterns with type variables

$$s \text{ match } X [\llbracket f \rrbracket_{X \rightarrow N} \llbracket f \rrbracket_X] \text{ then } t \text{ else } e$$

can be expressed in Scala as

```
s match
  case '{ ($f: x=>Int)($a) } => t
  case _ => e

// or explicitly as

s match
  case '{ type x; ($f: `x`=>Int).apply($a: `x`) } => t
  case _ => e
```

4.10 Discussion and Related Work

LISP LISP has a very simple way to treat *programs as data* based on the uniform representation of programs as lists. Quotation turns fragments of unevaluated code into data: '42 is a number, 'a is a symbol, '(+ 3 4) is a list of the quoted constituents. Quasiquote—*with a backquote*—lets us escape inside a program fragment (for example, of a whole list) with a comma operator that can *unquote* and evaluate a part of the quasiquoted expression e.g., `(1 2 ,(+ 3 4)).

Racket Racket has a sophisticated macro system. Contrary to Scala, Racket is dynamically typed. Typed Racket will type-check all expressions at the run-time phase of the given module [84]. Despite these fundamental differences, it is worth noting that Racket supports pattern matching with quasiquotes (quasipatterns). Interestingly, Racket goes one step further: much like the quasiquote expression form, unquote and unquote-splicing escape back to normal patterns, which is something that we do not support.

Multi-stage programming Multi-stage programming transfers the concepts of quotes, quasi-quotes, unquotes and staged evaluation [33; 20] in a statically scoped, modularly type-safe environment [67]. Multi-stage programming, popularised by MetaML [82] and MetaOCaml [13; 40; 39], made generative programming easier [17], effectively narrowing the gap of writing complex solutions of code manipulation such as code optimizations [86] and DSL implementations [18; 83]. Fred McBride [47] highlights the need to bridge the gap of expressing *computer-aided manipulation of symbols*. Arguing that it is important to lower the cognitive barrier of reading and writing algebraic manipulators such as algebraic simplifiers and integrators, he develops the first pattern matching facility for LISP; a form that provides a *natural description* to increase the user's *problem solving potential*. MacroML [26] used the quotation system of MetaML to define macros. The two fundamental quasiquote operators in Scala 3 were inspired by MetaML/MacroML and BER MetaOCaml.

Template Haskell Haskell was introduced to metaprogramming using quasiquotes with Template Haskell [66]. Neither MetaOCaml, MetaML, or Template Haskell support pattern matching with quasiquotes. A formalization of this system was proposed for Typed Template Haskell [87], which solves some previously unsound uses of type classes. It elaborates the program into a version that has staged versions of type classes. This elaborated form resembles the explicit handling of type classes that Scala uses. This implies that, in Scala, there is an additional syntactic overhead to encode the same sound type classes. Additionally, [87] introduces the concept of *splice environments* to handle staged captured variables in the splice lazily. Given that Scala is not pure, we have to perform *splice normalization* (from Section 3.2.4), capturing the same environments but evaluating the splices eagerly.

Squid Squid [58; 57], a metaprogramming library for Scala, advances the state of the art of staging systems and puts quasiquotes at the center of user-defined optimizations. The user can pattern match over existing code and implement retroactive optimizations modularly. To the best of our knowledge, in an earlier version of Squid, Parreaux et al. [57] were the first to represent locally closed terms as HOAS functions. However, they later revisited the approach to instead track free variables in the type. While expressive, this approach requires advanced typing features to encode open expressions such as path-dependent types, singleton types, and intersection types.

Möebius Möebius [31] is a run-time polymorphic multi-stage programming calculus that supports code generation and analysis. Like our calculus, it supports System F-like polymorphism, HOAS patterns, and type variables in the patterns; but unlike our calculus, it uses open expressions and tracks free variables explicitly.

Modal logic Our calculus is closely related to λ° [19] and λ^\square [20]. These calculi capture the temporal and modal logic essence of multi-stage programming. They support code generation but not code analysis.

4.11 Future Work

Mechanization While the “pen and paper” proof in Appendix A shows with a good degree of confidence that the calculus is sound, it is always better to have a mechanized proof to double-check the correctness of the calculus.

Scope extrusion The calculus assumes the lack of side channels, for which we use dynamic scope extrusion detection. It is worth formalizing this mechanism and precisely describe how it can be implemented at run-time or expressed as a static check.

Subtyping The calculus could be extended with subtyping. It should be fairly straightforward to extend the core calculus with subtyping, with a covariant T for $[T]$. What is more challenging is the extension of pattern matching operations, where the patterns should match if the scrutinees are subtypes of the patterns. Type constraints would need to be extended to type bounds constraints.

Higher-kinded types Scala has higher-kinded types and therefore it would be interesting to formalize the interactions of these types with staging. Having higher-kinded types would also allow generalizing quoted pattern type variables to be higher-kinded.

Meta-metaprogramming Another way to extend the system would be to investigate the possibility to also allow matching on programs that use quotes, splices, and matches themselves. Adding pattern matching on quotes and splices seems to involve a more general notion of staged eta-expansion. In order to add support for matching on the match itself, we would require some form of meta-patterns. While both extensions are interesting, we expect the metatheory to be significantly more involved, and thus neither of the two features is implemented in Scala 3.

4.12 Conclusion

We presented the *multi-stage macro calculus* λ^\blacktriangle for well typed and hygienic multi-stage metaprogramming, which allows both generative and analytical macros. We introduced the *polymorphic multi-stage calculus* F^\blacktriangle extensions to add two kinds of polymorphism. We proved the soundness of the full *polymorphic multi-stage macro calculus* and implemented it in Scala 3.

Typed AST Reflection **Part III**

5 Virtual ADTs for Portable Metaprogramming

This chapter contains a published paper authored by Stucki, Brachthäuser, and Odersky [74]. Section on “use cases” was removed to avoid duplications.

Scala 3 introduced *TASTy* as a new high-level intermediate representation providing a portability layer between compiler versions [51; 52]. The *TASTy* format defines an abstract representation of fully elaborated Scala programs. Instead of directly producing JVM bytecode, the compiler first generates binaries containing trees in *TASTy* format. These high-level binaries enable portability and can be used in future compiler versions to generate bytecode, generate documentation, support IDEs, analyze the program, or transform the program.

Given the promised stability of the *TASTy* format, it also provides the perfect basis to support semantically driven metaprogramming. However, this format exposes low-level encoding details that are not relevant to metaprogrammers. A high-level abstraction over this intermediate representation could provide the same portability guarantees while encapsulating representation details.

Many use cases of metaprogramming require directly manipulating program trees (or *abstract syntax trees* – ASTs) represented with *algebraic data types* (ADT). It appears natural to define interfaces for those trees, which the compiler would then implement. Such an implementation of metaprogramming was the underlying design of Scala 2’s experimental macros. However, it also directly couples the metaprogramming interface with the internal representation. This implies that the compiler internal representations cannot freely evolve without breaking the metaprogramming interface.

The *ADT virtualization problem* can be described as being able to virtualize (and thus decouple) the underlying representation of an ADT, while maintaining the exact same user interface as a normal ADT. In particular, a solution to the ADT virtualization problem should satisfy the following requirements:

- *Type Hierarchies*: The interface must be able to represent hierarchical and mutually recursive families of types.
- *Abstraction*: The interface must be *directly* implementable by any *isomorphic* runtime representation.
- *Ergonomics*: The interface must: (1) provide an object-oriented API with (static) methods and fields, (2) allow instances to be type-tested at run-time, and (3) support deconstructing instances with pattern matching.

No previous solution that we are aware of satisfies the above constraints. In this chapter, we introduce a set of implementation techniques that allows us to encode Virtual ADTs in Scala 3. We evaluated the implementation techniques in the implementation of the Scala 3 compiler to decouple the compiler internal representation of ASTs from the metaprogramming API. While our design of Virtual ADTs arose in the context of the Scala 3 metaprogramming API, the technique can be applied to any other ADT that requires decoupling the interface from its implementation.

5.1 Scaling APIs with Virtual ADTs

Often, library APIs not only consist of methods but also of *types*. Especially in *object-oriented* (OO) programming languages: interfaces and classes are used as the primary decomposition, and they thus occur very naturally to design library APIs. Depending on the size and purpose of the API, the involved types range from single individual types to mutually recursive families of types to full ADTs.

As an example, the Scala 3 compiler uses a variant of the following (simplified) ADT to model fully elaborated ASTs, types, and constants:

```
sealed trait Tree
sealed trait TermTree extends Tree
case class Literal(const: Constant) extends TermTree
case class WhileDo(cond: Tree, body: Tree) extends TermTree
// ...
sealed trait Type
case class TypeRef(prefix: Type, name: String) extends Type
case class ConstantType(const: Constant) extends Type
// ...
case class Constant(value: Any, tpe: Type)
```

The ADT represents a *family of types*, grouped into `Tree`, `Type`, and `Constant`. The ADT is *mutually recursive*, since `ConstantType` mentions `Constant`, and `Constant` mentions `Type` in turn. Finally, the ADT is structured *hierarchically*: types can be leaf types, composite types, abstract types, or a combination thereof. The hierarchical structure immediately gives rise to subtyping relationships.

Sometimes it is desirable to *decouple* the actual implementation of such types from their declaration to prevent dependencies of users on concrete implementation details [55]. Abstraction can be motivated by the need to later being able to choose another underlying runtime representation, or to offer a stable interface while internally changing the structure of the types. As illustrated in the above example, in OO languages, algebraic data types can be mutually recursive families of types, with arbitrary intermediate hierarchies modeled in terms of interfaces and subtyping. This significantly complicates efforts to decouple the declaration of an ADT from its implementation.

In the case of the above compiler AST, the declaration of the ADT is directly coupled to its implementation. It is not possible to choose another runtime representation, or modify the type hierarchies internally, without breaking existing programs that are consuming or producing such trees. While for compiler internals this form of encapsulation might not be necessary, in other use cases it is. Such a use case is the metaprogramming reflection API of Scala 3.

In the rest of this section, we identify a set of features provided by Scala 3, which allows us to express *Virtual ADTs*, satisfying our above requirements, while maintaining the same interface for users of the ADT. In particular, we use:

- *Abstract types* to model the type hierarchy;
- *Type classes* to recover runtime type tests in patterns;
- *Extension methods* to add methods on abstract types;
- *Abstract objects* to encode companion objects;
- *Unapply methods* to support pattern matching;
- *Singleton types* to encode case objects.

In this section, we describe in detail how all of these language features cooperate to achieve the desired decoupling, while providing maximal convenience to users of the API. The complete encoding of Virtual ADTs might seem complicated. However, designers can individually choose from the following subsections which individual aspect to support in their API.

Running Example: Modeling ASTs

As could be seen in the previous example, in the Scala, language ADTs can be expressed using a combination of sealed traits (which in essence can be understood as interfaces) and case classes (which are immutable classes that come equipped with support for pattern matching and comparison). To describe our solution of virtualizing ADTs, we use another, much simpler, running example: Peano numbers¹.

```
sealed trait Natural:
  def plus(other: Natural): Natural =
    this match
      case Successor(pred) => pred.plus(Successor(other))
      case Zero => other

case object Zero extends Natural
case class Successor(pred: Natural) extends Natural
```

This example defines a type `Natural` (for *natural* numbers) with a single method for adding two numbers. Natural numbers are constructed using the dedicated object `Zero` and the constructor `Successor`, which takes the predecessor as its argument. Case classes also provide the necessary means to perform pattern matching as illustrated in the implementation of `plus`.

5.1.1 Abstract Types: Separating Interface from Implementation

We intend to provide a programming interface very similar to the one defined above, without superimposing a concrete implementation for any of the types. From the perspective of an API user, the interface should expose the same subtyping relationships and should have the same ergonomics for constructing instances, pattern matching on instances, and calling methods (like addition).

The first step is to remove the dependency on any particular runtime class by making those types abstract.

```
trait Peano:
  type Nat
  type 0 <: Nat
  type Succ <: Nat
```

```
class Impl extends Peano:
  type Nat = Natural
  type 0 = Zero.type
  type Succ = Successor
```

¹It should be pointed out that this (quite obviously) is not how a library for numerical computation would represent numbers.

With this representation we are allowed to choose any isomorphic representation we want for `Nat`, `0` and `Succ` as long as the subtyping relationship holds. We could implement the interface using the concrete `Natural`, `Zero` and `Successor` implementations (shown in `class Impl`); or even use `Int`², `BigInt`, or `Any` for all three types.

Path-Dependent Interface

Users of the Virtual ADT will always use the abstract `Peano` interface to access the types, as illustrated below. Using a path on a parameter, or using the *cake pattern* [29].

```
def user1(peano: Peano): T =  
  import peano.*  
  
  ...
```

```
trait User2:  
  val peano: Peano  
  import peano.*  
  
  ...
```

The `Nat` type can only be used as a path-dependent type (such as `peano.Nat`). This implies that a `p1.Nat` and a `p2.Nat` only have the same types if their paths `p1` and `p2` are the same. This ensures that we do not accidentally mix instances of `p1.Nat` in `p2.Nat`, which might have different runtime representations.

Contextual Abstraction

To hide the verbosity of passing the `Peano` module, we can mark it as a *contextual* parameter. Defining the parameter with `using` will make it implicitly available in the body of the method. At call site the compiler will search for an implicitly available instance of type `Peano`.

```
def one(using p: Peano): p.Succ =  
  p.Succ(p.0)  
  
def two(using p: Peano): p.Succ =  
  p.Succ(one)
```

To provide the instance of `Peano` we can use a `given` definition.

```
given Peano = ...  
  
one.plus(two)
```

Contextual abstractions are similar to implicits from Scala 2. They provide greater flexibility, ease of use, and enable new abstractions such as contextual function types [53].

²Primitive type specialization is a non-goal. Primitive types will be boxed.

5.1.2 TypeTest: Supporting Run-Time Type Tests

By using abstract types, instead of modeling the ADT using traits or classes, we lose the ability to perform runtime type tests. Type tests are performed on runtime classes. However, abstract types do not introduce those classes. This implies that the following code would not work out of the box.

```
nat match // nat: Nat
  case succ: Succ => ...
```

To be able to perform this runtime type testing without explicitly using an extractor, Scala 3 added the built-in type class `TypeTest`.

```
trait TypeTest[-S, T]:
  def unapply(x: S): Option[T & x.type]
```

The implementation of the method `unapply` is expected to check whether the scrutinee `x` of type `S` is also of type `T`, in which case it returns the argument (hence the intersection with the singleton type `x.type`), refined to `T`. Given an instance of `TypeTest[Nat, Succ]`, we can thus check at runtime if an instance of a `Nat` is an instance of a `Succ`.

For `Succ`, we can require such type test instances as follows:

```
trait Peano:
  ...
  given SuccTypeTest: TypeTest[Nat, Succ]
```

By marking it as `given`, the type test instances will be implicitly brought into scope where a pattern requires one.

```
nat match // nat: Nat
  case succ: Succ => ...
  // is transformed by the compiler into
  case SuccTypeTest(succ) => ... // succ: Succ
```

Below, we implement it by using the runtime class `test` for `Successor`. If the test is successful, we return an instance of `Some[Successor & x.type]`, otherwise we return `None`.

```
class Impl extends Peano:
  ...
  object SuccTypeTest extends TypeTest[Nat, Succ]:
    def unapply(x: Nat): Option[Succ & x.type] =
      x match
        case s: (Successor & x.type) => Some(s)
        case _ => None
```

5.1.3 Extension Methods: Restoring the Interface

By using a type alias instead of a trait or class, we also lose the ability to define methods directly on those instances. To add methods to our abstract types, we can use extension methods.

```
trait Peano:
  ...
  extension (n: Nat) def plus(m: Nat): Nat
```

This way, we can call the plus method on instances of Nat.

```
def example1(p: Peano)(nat: p.Nat): p.Nat =
  nat.plus(nat) // `plus` called as a method

def example2(p: Peano)(nat: p.Nat): p.Nat =
  p.plus(nat)(nat) // `plus` called explicitly
```

Placing the extension methods directly in the Peano trait works but has limitations. Type erasure can easily cause conflicts on methods with the same name defined on different types. For example, if plus is defined in Nat and Succ, they will both be erased to:

```
// JVM bytecode signature
public Object plus(Object n, Object m);
```

To prevent this problem, we group all extension methods of a type in one trait (NatMethods below) and make this trait implicitly available:

```
trait Peano:
  ...
  given NatMethods: NatMethods
  trait NatMethods:
    extension (n: Nat) def plus(m: Nat): Nat
```

This encoding will ensure that there will not be conflicts between methods of different types. Implementing the NatMethods interface requires an object that defines the extension methods.

```
class Impl extends Peano:
  ...
  object NatMethods extends NatMethods:
    extension (n: Nat) def plus(m: Nat): Nat = ...
```

Chapter 5. Virtual ADTs for Portable Metaprogramming

Extensions can also be defined collectively to avoid the overhead of declaring the receiver (e.g., `n: Nat`) several times.

```
extension (n: Nat)
  def plus(m: Nat): Nat
  def times(m: Nat): Nat
```

5.1.4 Abstract Objects: Encoding Companions

When we define a case class in Scala, we automatically get a companion object with an `apply` factory method that serves as constructor for this type. We can encode the same functionality using an abstract `val` that will play the role of the companion object.

```
trait Peano:
  ...
  // the encoded companion object
  val Succ: SuccCompanion
  // the API of companion object
  trait SuccCompanion:
    def apply(nat: Nat): Succ
```

`SuccCompanion` defines all the methods that are available in the companion while `val Succ` provides the reference to the implementation of the companion. This encoding provides an *abstract companion object* that can then be implemented as a regular object.

```
class Impl extends Peano:
  ...
  object Succ extends SuccCompanion:
    def apply(nat: Nat): Succ = Successor(nat)
```

To ensure that `SuccCompanion` is only used to implement `val Succ`, we can additionally include a `this:Succ self` type in the interface.

5.1.5 Unapply Methods: Extractors

The main purpose of case classes is to be used in pattern matching.

```
nat match // nat: Nat
  case succ @ Succ(pred) => // succ: Succ, pred: Nat
```

To allow pattern matching on Virtual ADTs, we also add an `unapply` method to our abstract companion object. Assuming that we have implemented type tests (Section 5.1.2), we can

delegate type testing and only decompose the type-refined result in the `unapply`.

```
trait SuccCompanion:
  ...
  // Can only deconstruct values of type Succ.
  // This always succeeds and returns Some of the predecessor.
  def unapply(x: Succ): Some[Nat]
```

For example, to match a value of type `Nat` against `Succ`, we first perform a type test to see if the scrutinee is of type `Succ`, and only then use the `SuccCompanion.unapply` method as an extractor to decompose the ADT. As the `unapply` method returns `Some[Nat]`, this part of the match is statically known to succeed and the value of the predecessor is directly extracted from the `Some` instance. We can apply the same technique for extractors of any arity: `true` is used for arity 0, `Some` is used for arity 1 and tuples (T_1, \dots, T_n) are used for arity $n > 1$.

5.1.6 Singletons: Case Objects

In the case of 0, we have two possible encodings. The first is to use the same encoding as shown for `Succ`. This implies that the object will be created by calling the `0()` constructor. Similarly, we would use the `0()` extractor in pattern matches. As an alternative, assuming that all implementations of the interface will use a singleton value for 0, we can also encode this in the interface.

```
trait Peano:
  ...
  val 0: Nat
```

Like with case objects, with this definition we can simply use `0` to refer to this singleton value or pattern match on it. Its singleton type will be `0.type` just like `Zero.type`.

```
class Impl extends Peano:
  ...
  val 0: Nat = Zero
```

5.1.7 Summary

To avoid coupling to one implementation, we defined the interface of an ADT with abstract types. We added methods to the abstract types by using extension methods, showed how to express companion objects as abstract objects, recovered pattern matching using `unapply`, and showed how to enable runtime type tests. The complete implementation of `Peano` can be in Figures 5.1 to 5.3.

```
trait Peano:
  type Nat
  given NatMethods: NatMethods
  trait NatMethods:
    extension (nat: Nat) def plus(other: Nat): Nat

  val 0: Nat

  type Succ <: Nat
  given SuccTypeTest: TypeTest[Nat, Succ]
  val Succ: SuccCompanion
  trait SuccCompanion:
    this: Succ.type =>
    def apply(nat: Nat): Succ
    def unapply(x: Succ): Some[Nat]
```

Figure 5.1: Virtual ADT Interface for Peano Numbers

```
class Impl extends Peano:
  type Nat = Natural
  object NatMethods extends NatMethods:
    extension (nat: Nat) def plus(other: Nat): Nat =
      nat match
        case Successor(pred) => pred.plus(Successor(other))
        case Zero => other

  val 0: Nat = Zero

  type Succ = Successor
  object SuccTypeTest extends TypeTest[Nat, Succ]:
    def unapply(x: Nat): Option[Succ & x.type] =
      x match
        case s: (Successor & x.type) => Some(s)
        case _ => None
  object Succ extends SuccCompanion:
    def apply(nat: Nat): Succ =
      Successor(nat)
    def unapply(succ: Succ): Some[Nat] =
      Some(succ.pred)
```

Figure 5.2: Virtual ADT Implementation with Case Classes

5.2 Discussion

In this section, we discuss the extensibility scenarios enabled by the abstraction of Virtual ADTs and report limitations.

5.2.1 Changing the Internal Representation

The motivation behind Virtual ADTs is to modify the internal representation while maintaining a stable interface with users of the API. The virtual ADT can thus be seen as a *view* [25] on the underlying representation.

Virtual ADTs enable refactorings In general, the underlying representation can always be replaced with another one that is *isomorphic*. Refactorings that fall into this category range from simple renamings, or moving of information from one class to another, to structural changes like switching from a sum-of-products representation to a product-of-sums representation. In order to support the interface provided by Virtual ADTs, and in particular to implement `TypeTest`, it is important that different variants of the ADT can be distinguished at runtime (either by runtime classes, custom tags, values, etc.).

Virtual ADTs hide implementation details Obviously, not all fields or methods on the underlying representation need to be exposed to the user. Furthermore, composite subtrees in the intermediate representation can be represented as a single node in the user API.

Virtual ADTs provide fine-grained access As mentioned earlier, the implementation of a Virtual ADT does not need to necessarily mirror the tree structure of the interface. This implies that the implementor of a Virtual ADT can choose a more efficient representation, while providing the tree interface to the API user. For example, the implementation can group multiple different nodes into a single tagged node, or even flatten whole trees into a flat uniform representation. As an example, we show how to instantiate the Peano Virtual ADT with `BigInt` as underlying implementation, making it possible to represent large naturals compactly.

This is an extreme case where the hierarchal structure of the Virtual ADT is collapsed into a single flat domain. Interestingly, to implement type tests (in `SuccTypeTest`) we cannot compare runtime classes anymore, but need to analyze the underlying integer value.

5.2.2 Changing the Interface

Even though the Virtual ADTs are designed to abstract over changes in the underlying implementation, they are also well-suited for adding functionality to the user-facing API. In particular, adding new types, objects, or methods are backward binary compatible changes. This allows API designers to evolve the interface definition without breaking existing code.

```
class BigIntImpl extends Peano:
  type Nat = BigInt
  object NatMethods extends NatMethods:
    extension (nat: Nat) def plus(other: Nat): Nat = nat + other

  val 0: BigInt = 0

  type Succ = BigInt
  object SuccTypeTest extends TypeTest[Nat, Succ]:
    def unapply(nat: Nat): Option[Succ & nat.type] =
      if nat > 0 then Some(nat) else None

  object Succ extends SuccCompanion:
    def apply(nat: Nat): Succ = nat + 1
    def unapply(succ: Succ): Some[Nat] = Some(succ - 1)
```

Figure 5.3: Virtual ADT Implementation with BigInt

5.2.3 Monomorphism

If we assume that users of the Virtual ADT always use a single implementation at a time, then all the calls to methods of this interface will be effectively monomorphic. This implies that the JVM JIT compiler will be able to devirtualize and possibly inline those method invocations [30; 59; 77; 68].

5.2.4 Limitations

Unfortunately, Scala defines some methods on `Any`, which is the top of Scala's subtyping lattice. These methods can be called on `Nat` directly, potentially breaking encapsulation. In particular, this implies that calls to `equals`, `hashCode`, ..., or `toString` will be directly performed on the underlying representation. Currently, there is no way to override the behavior of those methods and therefore the implementation of the Virtual ADT needs to work under those constraints. For example, the `toString` method might not print the appropriate ADT structure. To work around this limitation, we can add a `show` method to the API that displays the desired format. Furthermore, methods like `equals` and `hashCode` come with contracts that need to be satisfied by the chosen underlying representation. Finally, API designers should document additional contracts on the declaration of the Virtual ADT, such as the use of structural equality for comparison of ADT nodes.

5.3 Related Work

Virtual classes and family polymorphism Abstracting over and refining families of types is a long-studied subject on its own [22; 23].

Ernst et al. [23] present the *vc* calculus of *Virtual Classes*, where classes are members of other classes and can be overridden and specialized, much like methods in other languages. Since classes are virtual and can vary at runtime, depending on the concrete object instance, it is important to distinguish between classes nested in different instances. The *vc* calculus supports *family polymorphism* [22]: member classes nested within one class represent a family of types that can be refined in parallel. Every instance of such a grouping class gives rise to its own family of types that should not be confused with those of other instances.

With path-dependent object types at its core [4; 5; 50], Scala is equipped with a very similar mechanism, allowing to distinguish between types that have different paths. While very similar at the surface, the underlying motivation behind Virtual ADTs differs greatly from that of virtual classes. Virtual classes are designed with extensibility in mind: they allow a step-wise refinement of nested types in subclasses. They thus do not satisfy our requirement of abstraction. In contrast, Virtual ADTs are designed with modularity in mind: to improve maintainability, implementations are fully decoupled from the interface description of the ADT. In particular, virtual classes only allow *refining* classes, that is, subclassing member classes and adding new methods, while refinements are always bound to the original class structure. Virtual ADTs avoid the bound with the original class structure as illustrated by instantiating `Nat` with `BigInt`.

Scala 2 experimental macros [12; 11] As a direct descendant of Scala 2 macros, the Scala 3 reflection API uses a similar encoding but provides fundamental changes. The Scala 2 version used `ClassTag` for type tests, which was later shown to be unsound when used in this kind of API definition [71]. Instead of using extension methods (which would have been costly in Scala 2), the abstract type is set to be a subtype of the trait defining the methods. This forces the compiler to use this exact representation. Abstract companion objects are defined in a similar way, though we added an extra self type constraint. Attempts to modify the interface to handle two compilers were designed [46] but were replaced with the TASTy-based solution.

Type testing Scala 2 introduced `ClassTag` to generically create array. It was later retrofitted as a way to perform generic type tests, but this addition was later shown to be unsound. In particular, it was unsound for this use case, a limitation that was described in depth in [71]. Hence, in Scala 3 we introduced `TypeTest` to be able to provide sound type testing and properly support Virtual ADTs.

5.4 Future Work

Abstract object Instead of encoding the abstract object explicitly, Scala could introduce the `abstract object` concept natively. An abstract object would define an object that may have abstract members. Then on the implementation side, it would be implemented with a concrete object.

```
trait Peano:  
  ...  
  abstract object Succ:  
    def apply(nat: Nat): Succ  
    def unapply(x: Succ): Some[Nat]
```

Virtual ADTs If this pattern is used frequently it would be helpful if the language provided the abstraction natively. We could write code like in the example below and get all the encoding generated automatically.

```
trait Peano:  
  virtual trait Nat:  
    def plus(other: Nat): Nat  
  virtual case object 0 extends Nat  
  virtual case class Succ(pred: Nat) extends Nat
```

5.5 Conclusion

We showed how to encode a Virtual ADT in Scala 3 and how this encoding provides a way to decouple the interface from the implementation. We showed how Virtual ADTs were used to create the metaprogramming interface.

6 A TASTy Reflection Interface

Scala 3 introduced a new high-level intermediate representation that provides a portability layer between compiler versions. The TASTy format is the perfect basis to support semantically driven metaprogramming. A high-level abstraction over this intermediate representation provides the same portability guarantees as the TASTy format while encapsulating low-level representation details.

Many use cases of metaprogramming require directly manipulating program trees (or *abstract syntax trees* – ASTs) represented with *algebraic data types* (ADT). We use *Virtual ADTs* to abstract over the TASTy tree implementation of the compiler as discussed in Chapter 5.

In this chapter, we discuss the details of the TASTy format in Section 6.1. We provide an overview of all the reflection APIs in Section 6.2. We then discuss the interactions between the reflection API and multi-stage programming in Section 6.3. We show other applications of the reflection API in Sections 6.4 to 6.6. Finally, we discuss related and future work and conclude in Sections 6.7 to 6.9.

6.1 TASTy Binaries

The TASTy format is the *compact, lazy, extensible* and *precise* typed abstract syntax tree serialization format of Scala 3 [51; 52]. In the compiler, we use it for separate and incremental compilation, documentation generation, and code decompilation. We also use it for language servers in IDEs, and the metaprogramming API for quoted code.

TASTy is portable by design: it allows ASTs generated by one compiler to be accessed from any future compiler version. We leverage TASTy in some way or another for all metaprogramming features. TASTy contains the fully elaborated program ASTs, which is the program's most complete and detailed semantic representation. It also contains the documentation of definitions and positions of ASTs in the source. The binary representation is too low-level for direct usage. Instead, metaprogramming features use the compiler ASTs which map to TASTy.

6.2 Overview of the Reflection API

The reflection API exposes a view of the compiler's TASTy tree representation. This view mostly follows the encoding of the TASTy format but also adds some extra functionality that the compiler can provide.

Trees The API defines the `Tree` type hierarchy to represent the typed ASTs. A notable sub-hierarchy rooted in the `Definition` type contains: `ClassDef` for class definitions, `TypeDef` for type definitions, `DefDef` for method definitions, and `ValDef` for `val`, `var` and `lazy val` definitions. `Term` is a sub-hierarchy that contains all term expressions such as `Literal`, `New`, `Apply`, `If`, `Match`, and many more. The `TypeTree` sub-hierarchy represents types written in the source. We also have the `PackageClause` type to define the package declarations of the source. Finally, a `Tree` can be pattern-related such as `CaseDef`, `Unapply`, `Bind`, and `Literal` (both a term and a pattern).

Trees are encoded using a Virtual ADT and can therefore be used as patterns and reconstructed from their parts as if they were case classes. Trees define methods to extract their parts and other non-structural information such as positions. We can test for the type of a tree using pattern matching.

The reflection interface also provides a few utility classes on trees. `TreeMap` gives a way to transform trees. `TreeTraverser` provides a way to traverse trees without transforming them. `TreeAccumulator` gives a way to accumulate information from trees.

Types Our API defines the `TypeRepr` type hierarchy to represent types. Every `Term` and `TypeTree` has a type accessible with the `tpe` method. Types can be references to named types such as `TypeRef` and `TermRef`, literal singleton types such as `ConstantType`, type applications `AppliedType`, type refinements `Refinement`, union and intersection types `OrType/AndType`, among others. The sub-hierarchy `LambdaType` provides types for method and type lambdas definitions.

Constant Our API defines the `Constant` types to represent literal constants. These includes all primitive types, `String`, `Unit`, `null` and `classOf [T]`.

Symbol Symbols are unique identifiers for definitions. Each `Definition` tree defines a symbol that can be copied to a new version of the same definition during transformations. Terms such as `Select` and `Ident` contain references to the symbol of the definition to which they refer.

Symbols also contain extra information about definitions, such as owners, signatures, documentation, flags, positions, trees, and annotations. For methods and value definitions, the symbol gives information about the overriding relationships. A class symbol contains information about the parents and members (declared or inherited).

Flags Flags provide extra information about a symbol, such as modifiers and internal encoding details. The `Flags` API mimics a bitwise flag encoding used in the compiler. The `show` method exposes a human-readable representation of flags.

Signatures The Signature of a method symbol provides its bytecode signature.

Standard definitions The `defn` object provides fast accessors to many symbols defined in the language or standard library.

Implicits It is possible to programmatically trigger an implicit search using the `Implicits` module. It will return a `Term` if the search succeeds or give the reason for the failure (such as `NoMatchingImplicits`, `AmbiguousImplicits`, or `DivergingImplicit`).

Position Source `Positions` are ranges with `start` and `end` offset in the source file. Our API provides the line number and column of the position and the source code at that position. Every `Tree` has a `pos` method that returns the tree's position. Newly created trees get the position of the code that generated those trees. We also use positions for error reporting. In this case, it is often helpful to create a custom (more precise) position.

SourceFile `SourceFile` provides a simple abstraction over a physical or virtual source file. Since files might be virtual, for example in the REPL, we might not always have access to a file path. It also provides a way to access the source code when available.

Reporting The `report` module gives a way to report errors, warnings, and other information.

Printers The `Printer` type class provides a customizable way to print `Tree`, `TypeRepr` and `Constant` values. Printers come into two categories: source printers and structure printers. Source printers (`TreeCode`, `TypeReprCode`, `ConstantCode`, among other variants) show the code in source code format and are used by default. Structural printers (`TreeStructure`, `TypeReprStructure`, `ConstantStructure`) print the ADT structure. These reflect the code needed to construct or match these trees, types, or constants.

6.3 Multi-Stage Programming with Reflection

Multi-stage programming offers a powerful way to generate and analyze programs while ensuring strong static safety guarantees. This safety comes at the cost of the expressiveness of our system. For the use cases where extra expressivity is needed, we extend the multi-stage system with the reflection API to allow inspection and creation of typed ASTs.

Static and Dynamic Guarantees

For the reflection API, we trade-off some static guarantees for dynamic guarantees in order to be more expressive.

Well-typed The most evident trade-off is the static typing of expressions. For each `Expr [T]`, we have the static guarantee that the expression is well typed. On the other hand, a `Term` only knows its type dynamically. Assuming that a term is hygienic, well-scoped, and well-formed, we can safely transform that term into an `Expr [Any]`. We can use the `asExpr` method on `Term` to get its tree. Furthermore, we can cast expressions into `Expr [T]` using `asExprOf [T]` for a known `T` that can be checked dynamically at run-time using a `Type [T]`. When we do not know the type of an expression statically, we can use a quoted pattern with a type variable to name and retrieve this type.

Hygiene The reflection API is hygienic. The API does not provide ways to refer to a variable by its name; only symbolic references can be constructed.

Well-scoped The reflection API does not provide any static guarantees about the scoping of variable references. We can check these dynamically once we know the scope where the tree is used.

Well formed AST Every quote `'{ . . }` produces a well formed AST by construction. However, the TASTy API allows the construction of nonsensical tree structures that the compiler will not know how to handle. For example, we could create a nonsensical tree which would represent something like `new =>Int using New(ByName(TypeTree.of[Int]))`. Given the complexity of the language, these checks involve complex rules. We check all of them dynamically.

Dynamic Checks

-Ycheck The compiler contains infrastructure to check that the AST is well-formed, well typed, and well-scoped after each compilation phase. We need these checks for the reflection API; they run after macro expansion.

-Xcheck-macros While the `-Ycheck` infrastructure can catch many issues, we only execute it after macro expansion. `-Xcheck-macros` enables extra checks during the creation of ASTs. These provide better information for debugging, but come with a small performance cost.

Code Generation

Consider the following example where we want to generate a switch-like expression. Given an integer number, we want to index it into the list of expressions passed as arguments.

```
switch(n)("a", "b", ..., "z")
// expands to
n match
  case 0 => "a"
  case 1 => "b"
  ...
  case 25 => "z"
```

```
inline def switch[T](n: Int)(inline xs: T*): T = ${ switchExpr('n, 'xs) }

def switchExpr[T:Type](n: Expr[Int], xs: Expr[Seq[T]])(using Quotes): Expr[T] =
  xs match
    case Varargs(args) => mkSwitch(n, args.toList)
    case _ => '{ $xs($n) } // access argument from unpacked varargs

def mkSwitch[T: Type](n: Expr[Int], xs: List[Expr[T]])(using Quotes): Expr[T] =
  ...
```

Listing 6.1: def switch

While we get pretty far using multi-stage programming alone, there is no way to express the expression containing the match in a well typed quote. We know neither the number of cases nor the patterns statically. Instead, we use the reflection API to create this expression.

```
def mkSwitch[T: Type](n: Expr[Int], xs: List[Expr[T]])(using Quotes): Expr[T] =
  import quotes.reflect.* // `quotes` refers to the given `Quotes` instance
  val cases: List[CaseDef] =
    for (caseExpr, i) <- xs.zipWithIndex
      yield CaseDef(Literal(IntConstant(i)), None, caseExpr.asTerm)
  Match(n.asTerm, cases).asExprOf[T]
```

First, we enable reflection using `import quotes.reflect.*`. In the example this will import the `Tree`, `CaseDef`, `Match`, `Literal` and `IntConstant` classes, as well the `asTerm` and `asExprOf` methods. We can convert any expression `Expr[T]` to a term tree `Term` using `asTerm`. Using trees, we can construct arbitrary ASTs. In this case, we created the one for a `Match` expression. To make it an expression, we can use the `asExprOf` method, which will check that the expression has the expected type. The `asExprOf` call uses the given `Type[T]` to check the type of the expression.

Code Analysis

To continue with the previous example, consider the opposite operation of decomposing a switch expression into its values. Given a switch expression where indices are represented as case patterns, we want to return all cases in a sequence.

```
unswitch {
  n match
    case 0 => "a"
    case 1 => "b"
    ...
    case 25 => "z"
}
// expands to
(n, Seq("a", "b", ..., "z"))
```

We cannot write a quote pattern because we do not know the number of cases or the patterns statically. Again, the reflection API gives the necessary flexibility to decompose this expression.

```
inline def unswitch[T](inline x: T): (Int, Seq[T]) = ${ unswitchExpr('x) }

def unswitchExpr[T: Type](x: Expr[T])(using Quotes): Expr[(Int, Seq[T])] =
  import quotes.reflect.*
  x.asTerm match
    case Inlined(_, _, Block(nil, Match(scrut, cases))) =>
      val exprs: List[Expr[T]] = cases.zipWithIndex.map {
        case (CaseDef(Literal(IntConstant(i)), None, body), j) if i == j =>
          body.asExprOf[T]
        case (cse, _) => report.errorAndAbort("unexpected case: ", cse.pos)
      }
      scrut.asExpr match
        case '{ $scrutExpr: Int } => '{ ($scrutExpr, ${Expr.ofSeq(exprs)}) }
        case _ => report.errorAndAbort("not Int scrutinee", scrut.pos)
    case xAsTerm => report.errorAndAbort("not a match", xAsTerm.pos)
```

Listing 6.2: def unswitch

In this case, we can match on the Match tree to extract the scrutinee, and the cases, and collect all right-hand-side expressions, to create a sequence of Expr[T]. All the right-hand sides must be of type T because that is the type of the match. These expressions are well-scoped because the cases did not introduce any bindings. These expressions are also well formed because the overall match is well-formed. As a result, all dynamic checks will always succeed for this code.

In this example, we also see how we can use the tree positions to provide more precise error messages to the macro user. At the same time, we can pass an expression to emit an error at the position of the whole expression.

Note that we only needed to handle the cases with reflection. We can easily use quotes to recover the type of the scrutinee and create the resulting expression.

6.4 TASTy Inspector

The *TASTy Inspector* library¹ provides a way to inspect the contents of TASTy files using reflection. In the following example, the `TastyInspector` loads the given files, and prints the source representation of the ASTs to the standard output.

```
import scala.tasty.inspector.*

object PrintCode extends Inspector:
  def inspect(using Quotes)(tastys: List[Tasty[quotes.type]]): Unit =
    import quotes.reflect.*
    for tasty <- tastys do
      println("// " + tasty.path)
      println(tasty.ast.show)

TastyInspector.inspectTastyFiles(paths)(PrintCode)
```

Listing 6.3: object `PrintCode`

This interface provides the `Quotes` and a list with all the loaded ASTs as `Tree`. We bundle the tree and its path as a `Tasty` instance. Since the AST contains a class definition, we cannot pass the definition as an `Expr` as in macros. Alternatively, it would also be possible to use pattern matching on quoted code to match expressions contained within this AST, as done in Listing 6.2.

Scaladoc The Scala 3 documentation tool works by reading the TASTy files and generating documentation on a web page. To do so, it uses the TASTy inspector interface to access all definitions and their documentation.

¹Implemented in the Scala 3 Dotty project <https://github.com/lampepfl/dotty>. sbt library dependency `"org.scala-lang" %% "scala3-tasty-inspector" % scalaVersion.value`

6.5 Decompiler

When we decompile a program, we want to see a source version of the compiled program representing the same program. Scala 3 offers this functionality to allow users to decompile the TASTy binaries, using the `scalac -decompile` command.

This decompiler is trivial to implement using the reflection interface. We only need to load the AST and show it in its source representation. Doing so requires that the `PrintCode` of Listing 6.3 is a valid decompiler implementation. The only difference between the original source and the decompiled one is that the latter will be fully elaborated (explicit types, explicit implicits, explicit extension methods, ...). Hence it can also show the result of type inference and other typing elaborations.

```
package example
object Math:
  import math.Numeric.Implicits.infixNumericOps
  def sq[T: Numeric](x: T): T = x * x

/** Decompiled from ./example/Math.tasty */
package example {
  object Math {
    import scala.math.Numeric.Implicits.{infixNumericOps}
    def sq[T](x: T)(implicit evidence$1: scala.Numeric[T]): T =
      scala.math.Numeric.Implicits.infixNumericOps[T](x)(evidence$1).*(x)
  }
}
```

6.6 Macro Annotations

Macro annotations are annotations that trigger tree transformations. We designed the reflection API to support these kinds of transformations. At the time of writing, this feature is still in the prototype phase of development [7].

The aim is to be able to annotate a definition with a macro annotation. We apply the transformation defined in the annotation class to the annotated definition. For example, we could implement a macro annotation `@memoize` that automatically adds argument-based memoization to a normal `def`.

```
@memoize def fib(n: Int): Int =
  if n <= 1 then 1 else fib(n - 1) + fib(n - 2)
```

When we expand the macro annotation we can add new definitions such as a `fibCache` and modify the implementation of the annotated definition accordingly.

```

val fibCache = Map.empty[Int, Int]
def fib(n: Int): Int =
  if !fibCache.contains(n) then
    fibCache(n) = if n <= 1 then 1 else fib(n - 1) + fib(n - 2)
  fibCache(n)

```

Macro annotations are annotations that inherit from a `MacroAnnotation` trait.

```

trait MacroAnnotation extends StaticAnnotation:
  def transform(using Quotes)(tree: quotes.reflect.Definition):
    List[quotes.reflect.Definition]

```

The implementation of this logic uses the same `Quotes` context as inline macros. The main difference is that we manipulate definitions with unknown signatures. Therefore, we need to use the reflection API. While the creation and manipulation of definitions use reflection, we can use quotes and splices to generate well typed expressions that we will use in the definitions.

```

class memoize extends MacroAnnotation:
  override def transform(using Quotes)(tree: quotes.reflect.Definition):
    List[quotes.reflect.Definition] =
  import quotes.reflect._
  tree match
    case DefDef(name, params @ List(TermParamClause(List(param))), tpt, Some(impl)) =>
      (Ref(param.symbol).asExpr, impl.asExpr) match
        case ('{ $paramRef : arg }, '{ $implExpr : res })=>
          val cacheSymbol = Symbol.newVal(tree.symbol.owner, name + "Cache",
            TypeRepr.of[Map[arg, res]], Flags.EmptyFlags, Symbol.noSymbol)
          val fibCacheRef = Ref(cacheSymbol).asExprOf[Map[arg, res]]
          val cacheVal = ValDef(cacheSymbol, Some('{ Map.empty[arg, res] }.asTerm))
          val newDef = DefDef.copy(tree)(name, params, tpt, Some(
            '{
              if !$fibCacheRef.contains($paramRef) then
                $fibCacheRef($paramRef) = $implExpr
                $fibCacheRef($paramRef)
            }.asTerm
          ))
          List(cacheVal, newDef)
        case _ =>
          report.errorAndAbort("expected a `def` with a single parameter list")

```

6.7 Related Work

Scala 2 experimental macros As a direct descendent of the Scala 2 macros [12; 11], the Scala 3 reflection API uses a similar encoding but provides fundamental changes. Scala 2 used an ADT encoding that tied the reflection API to compiler internals. This made it hard to evolve the compiler without breaking the portability of the reflection API. Scala 3 uses TASTy as lingua franca to expose the ADTs and encodes them using Virtual ADTs, for portability purposes.

TASTy Query TASTy Query² is intended to be at the core of all compiler-independent tools that analyze TASTy. It defines ADTs for trees and types. It also provides classes representing symbols, signatures, names, and constants. The tool loads the complete TASTy definitions and returns them as trees to the user. It currently lacks some advanced functionalities that the compiler can provide to the Reflection API, such as subtyping comparisons. Another difference is that the compiler leverages the laziness of TASTy to avoid loading the full trees whenever possible. TASTy Query takes a more straightforward approach and loads all trees eagerly.

Compiler plugins Scala 3 provides two kinds of compiler plugins³: *standard* and *research* plugins. A *standard plugin* can insert a new compilation phase into the compiler. A *research plugin* can customize all compilation phases; these are only allowed under the experimental mode. Scala 2 provided the equivalent to the standard plugin but also provided *analyzer plugins* that ran during type-checking, allowing the plugin to influence type-checking. The research plugins can also influence typing by changing the implementation of the type-checker.

Plugins are more flexible than the reflection interface (macros, inspector, annotations) as they can modify the code in any step of the pipeline. However, they use the internal compiler APIs, which are more complex and have no portability guarantees between compiler versions.

²<https://github.com/scalacenter/tasty-query>

³<https://dotty.epfl.ch/docs/reference/changed-features/compiler-plugins.html>

6.8 Future Work

Reflection interface The design of the interface is complete. Nevertheless, it could still change in the future. Each time we add a new language feature to the language, we might need to add it to the interface. Similarly, if we extend the TASTy format, we must reflect this addition in the interface. We use the Virtual ADTs encoding to ensure that these additions are binary compatible.

Macro annotations Currently in the prototype phase, the macro annotations still need considerable work. Firstly, we need to specify what kinds of transformations should be allowed. Secondly, we need to design a robust interface that covers all the use cases we plan to support and allows for the evolution of the interface.

Portable plugin We already have plugins, but we must implement them using the internal compiler API. We could create a plugin interface that would use the reflection API. These plugins would be portable and usable by future versions of the compiler.

6.9 Conclusion

Reflecting on the TASTy trees provides a complete semantic view of the program. This gives the ability to reason about programs at the same level as the compiler. The API greatly extends the expressivity of multi-stage programming at the cost of verbosity and the loss of some static guaranties. We showed that the reflection API is integral to complete program analysis tools, such as documentation generation, decompilation, and general code inspection. It also has the potential to be applied to more general code transformation interfaces such as annotation macros and possibly portable plugins.

Epilogue

7 Academic Projects

We undertook a handful of academic projects since implementing the Scala 3 metaprogramming framework. This is a list of projects that we have worked on in our research group.

- **Staged Tagless Interpreters:** An early proof of concept project [43] focused on porting code from [15; 80] into Scala 3.
- **Strymonas:** A port of the Strymonas stream fusion library to Scala [42; 70].
- **String Interpolator Macros:** Several projects focused on using multi-stage programming macros on string interpolators [2; 36; 27]. The projects focused on code analysis and generation using multi-stage programming macros.
- **Scala 3 Decompiler:** A project focused on showing the capabilities of the *TASTy Inspector* and reflection API [10]. We now use it as the de facto TASTy decompiler.
- **Scala 3 Doc:** A project focused on creating a new documentation tool using *TASTy Inspector* and reflection API [1]. Now it has evolved into the de facto documentation tool for Scala 3.
- **DSLs:** Several projects focused on showing the DSL capabilities of multi-stage programming in Scala 3 [48; 44].
- **Mechanized Proof:** A mechanized proof [61] of an earlier and less expressive version of the macro calculus [76].
- **Macro Annotations for Scala 3:** A project focused on the design and implementation of new annotation macros based on the reflection API [7]. We could use multi-stage programming to generate and analyze expressions when implementing these macros.

8 Core Library, Tools and Community Projects

Core Libraries and Tools

The following tools that come included with Scala 3 use or extend the metaprogramming API. These projects were initially developed in our research group¹.

- **Scala 3 Doc** `scaladoc`
- **Scala Decompiler** `scalac -decompile`
- **Scala Staging** `scala.quoted.staging`
- **TASTy Inspector** `scala.tasty.inspector`

Community Projects

Scala 2/3 community Some projects are cornerstones of the Scala ecosystem and need to be ported to Scala 3 to allow a successful transition². The following non-exhaustive list contains projects that had Scala 2 macros that were ported by the community into Scala 3 macros. Most of these projects opted to port their macros using the reflection API due to its similarity with the previous implementation. Even though these macros were not designed with multi-stage programming in mind, many used multi-stage programming to make parts of the implementation shorter, safer, and more readable.

Airframe, Argonaut, Blindsight, Izumi-reflect, Jsoniter Scala, Kebs, Log4s, Minitest, Monocle, MUnit, PPrint, Proto, Quill, ScalaPy, ScalaTest + JUnit, ScalaTest, Scodec, Scodec-bits, Shapeless 3, SourceCode, Tapir, Zio, μ Pickle, μ Test, ...

Scala 3 community Despite Scala 3 still being in its early days, we have already seen several new projects leveraging Scala 3 macros and metaprogramming in general. Here is a non-exhaustive list of open-source projects using Scala 3 macros.

Dotty CPS Async, DottyTags, Iron, Less Funky Trees, Perspective, Scodec 2, SLInC, Scala Reflection, Shaka, TypeTrees, ...

¹Scala 3 Doc was then further developed by VirtusLab.

²Note that Scala 3 projects can use most macroless Scala 2 projects without a port

9 Conclusion

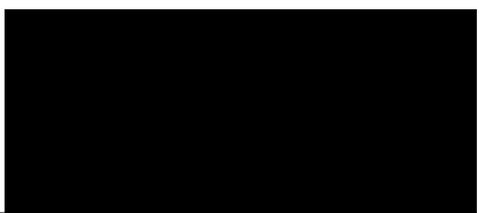
In this thesis, we demonstrated that it is possible to design, implement and use in production a *Portable Scalable Semantically Driven Metaprogramming System*. We used three semantically driven metaprogramming abstractions with different levels of expressiveness: *inline*, *multi-stage programming* and *typed AST reflection*. We showed how the system is scalable by trading static safety for expressiveness when needed. We also showed how a common portable AST representation (TASTy) can be used in all levels of metaprogramming to make the whole system portable.

Our multi-stage programming system is the first implementation to simultaneously support macros and run-time code generation. We use inline definitions to declare multi-stage macros without leaking implementation details. We designed and implemented a powerful quote pattern matching system to allow code analysis and transformation. We formalized our macro system with the λ^\blacktriangle and *System F[▲]* calculi. We proved soundness of the calculi.

We designed the *Virtual ADT* encoding to have an ergonomic, portable, and sound encoding for the AST reflection interface. To this end, we also improved Scala's generic type testing abstraction.

We released the system in Scala 3.0. This system allowed crucial core macro libraries of the Scala 2 ecosystem to be ported to Scala 3, allowing for a smooth transition between the two versions.

Appendix



A Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

We prove soundness of the calculus in terms of the standard theorems for progress (Theorems A.1 and A.2) and preservation (Theorems A.3 and A.4). Alternatively, modular proofs for the calculi of Sections 4.2, 4.4 and 4.5 can be found in [75].

A.1 Proof of Progress

Definition A.1 (Well-formed Ω).

$\Sigma \vdash \Omega$ **wf** if and only if $\text{dom}(\Sigma) = \text{dom}(\Omega) \wedge \forall i \in \mathbb{N}_0, x \in \text{dom}(\Omega). \Sigma \upharpoonright \emptyset \vdash^i \Omega(x) : \Sigma(x)$

Theorem A.1 (Progress for Programs).

If $\Sigma \vdash p : T$, then p is a value $\vdash p$ **vl** or, for any Ω such that $\Sigma \vdash \Omega$ **wf**, there exists p' and Ω' such that that $p \mid \Omega \longrightarrow p' \mid \Omega'$

Proof.

We perform induction on type derivation of $\Sigma \vdash p : T$.

Case T-EVAL where p is `eval t` If t is a value $\vdash^0 t$ **vl**, then p is a value $\vdash p$ **vl** by definition V-EVAL. Otherwise if t is not a value, by Theorem A.2 using $\Sigma \upharpoonright \emptyset \vdash^0 t : T$ and $\Sigma \vdash \Omega$ **wf** there exists a t' such that $t \xrightarrow{\Omega}^0 t'$; therefore we can take a step with E-EVAL.

Case T-DEF where p is `def $x = [t]$ in p_1` If t is a value $\vdash^1 t$ **vl**, then we can take a step with E-COMPILE, therefore there exists $p' = p_1$ and $\Omega' = \Omega, x := t$ such that $p \mid \Omega \longrightarrow p' \mid \Omega'$. Otherwise if t is not a value, by Theorem A.2 using $\Sigma \upharpoonright \emptyset \vdash^1 t : T_1 \rightarrow T_2$ and $\Sigma \vdash \Omega$ **wf** there exists a t' such that $t \xrightarrow{\Omega}^1 t'$. Then we can take a step with E-MACRO and therefore there exist $p' = \text{eval } t'$ and $\Omega' = \Omega$ such that $p \mid \Omega \longrightarrow p' \mid \Omega'$. ■

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

Definition A.2 (Restricted Typing Context).

$$\Gamma^{\geq 1} ::= \emptyset \mid \Gamma^{\geq 1}, x : {}^i T \mid \Gamma^{\geq 1}, X \quad \text{for } i \geq 1$$

Theorem A.2 (Progress for Terms).

If $\Sigma \vdash \Omega \mathbf{wf}$ and $\Sigma \mid \emptyset \vdash^i t : T$, then t is a value $\vdash^i t \mathbf{vl}$ or there exists t' such that $t \xrightarrow{\Omega}^i t'$

Proof.

The proof of progress trivially follows from Lemma A.2, by choosing $\Gamma^{\geq 1} = \emptyset$. ■

Lemma A.1 (Canonical Forms).

- If $\vdash^0 t \mathbf{vl}$ and $t : \mathbf{C}$, then $t = \mathbf{c}$ for some \mathbf{c} .
- If $\vdash^0 t \mathbf{vl}$ and $t : T_1 \rightarrow T_2$, then $t = \lambda x : T_1. t_1$ for some x and t_1 .
- If $\vdash^0 t \mathbf{vl}$ and $t : \forall X. T$, then $t = \Lambda X. t_1$ for some t_1 .
- If $\vdash^0 t \mathbf{vl}$ and $t : \lceil T \rceil$, then $t = \lceil t_1 \rceil$ for some t_1 .

Proof.

By case analysis on the value definition $\vdash^0 t \mathbf{vl}$. ■

Lemma A.2 (Extended Progress for Terms).

If $\Sigma \vdash \Omega \mathbf{wf}$ and $\Sigma \mid \Gamma^{\geq 1} \vdash^i t : T$, then t is a value $\vdash^i t \mathbf{vl}$ or there exists t' such that $t \xrightarrow{\Omega}^i t'$

Proof.

Perform induction on the typing derivation of $\Sigma \mid \Gamma^{\geq 1} \vdash^i t : T$.

Case T-CONST with $t = \mathbf{c}$

Trivial, t is a value $\vdash^i t \mathbf{vl}$ by definition V-CONST.

Case T-VAR with $t = x$

Sub-case $i = 0$ As $x : {}^0 T \notin \Gamma^{\geq 1}$ by definition of $\Gamma^{\geq 1}$, the precondition $\Sigma \mid \Gamma^{\geq 1} \vdash^0 x : T$ cannot hold.

Sub-case $i > 0$ Trivial, t is a value $\vdash^i t \mathbf{vl}$ by definition V-VAR.

Case T-LINK with $t = x$

Sub-case $i = 0$ From T-LINK we know that $x : T \in \Sigma$. By the premise $\Sigma \vdash \Omega \mathbf{wf}$ we have that $x \in \text{dom}(\Omega)$ and therefore there exists a $t' = \Omega(x)$ such that $t \xrightarrow{\Omega}^i t'$. Hence we can take a step using E-LINK.

Sub-case $i > 0$ Trivial, t is a value $\vdash^i t \mathbf{vl}$ by definition V-VAR.

Case T-ABS with $t = \lambda x:T.t_1$.

Sub-case $i = 0$ Trivial, t is a value $\vdash^0 t \mathbf{vI}$ by definition V-ABS-0.

Sub-case $i > 0$ If t_1 is a value, then t is a value $\vdash^i t \mathbf{vI}$ by definition V-ABS. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \xrightarrow{\Omega}^i t'_1$. Therefore there exists $t' = \lambda x:T_1.t'_1$ such that $t \xrightarrow{\Omega}^i t'$.

Case T-TABS with $t = \Lambda X.t_1$

Sub-case $i = 0$ Trivial, t is a value $\vdash^0 t \mathbf{vI}$ by definition V-TABS-0.

Sub-case $i > 0$ If t_1 is a value, then t is a value $\vdash^i t \mathbf{vI}$ by definition V-TABS. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \xrightarrow{\Omega}^i t'_1$. Therefore there exists $t' = \Lambda X.t'_1$ such that $t \xrightarrow{\Omega}^i t'$.

Case T-APP with $t = t_1 t_2$

Sub-case $i = 0$ If t_1 is not a value, then by induction hypothesis there exists a t'_1 such that $t_1 \xrightarrow{\Omega}^0 t'_1$. Therefore by E-APP-1 there exists a t' such that $t \xrightarrow{\Omega}^0 t'$. If t_1 is a value and t_2 is not a value, then by induction hypothesis there exists a t'_2 such that $t_2 \xrightarrow{\Omega}^0 t'_2$. Therefore by E-APP-2 there exists a t' such that $t \xrightarrow{\Omega}^0 t'$. Otherwise, if t_1 and t_2 are values, by the Lemma A.1 using $t_1 : T_1 \rightarrow T$ from T-APP, we know that $t_1 = \lambda x:T_1.t_3$. Therefore we take a step using E-BETA to get a $t' = t_3[t_2/x]$ such that $t \xrightarrow{\Omega}^0 t'$.

Sub-case $i > 0$ If t_1 and t_2 are values, then t is a value by definition V-APP. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \xrightarrow{\Omega}^i t'_1$. Therefore there exists $t' = t'_1 t_2$ such that $t \xrightarrow{\Omega}^i t'$. Lastly, if t_2 is not a value, then by induction hypothesis there exists a term t'_2 such that $t_2 \xrightarrow{\Omega}^i t'_2$. Therefore there exists $t' = t_1 t'_2$ such that $t \xrightarrow{\Omega}^i t'$.

Case T-TAPP with $t = t_1 T_1$

Sub-case $i = 0$ If t_1 is not a value, then by induction hypothesis there exists a t'_1 such that $t_1 \xrightarrow{\Omega}^0 t'_1$. Therefore by E-TAPP there exists a t' such that $t \xrightarrow{\Omega}^0 t'$. Otherwise, if t_1 is a value, then $t_1 : \forall X.T_2$ by T-APP. By the Lemma A.1 $t_1 = \lambda X.T_2$, therefore we can use E-TBETA to get a $t' = t_1[T_1/X]$ such that $t \xrightarrow{\Omega}^0 t'$.

Sub-case $i > 0$ If t_1 is values, then t is a value by definition V-TAPP. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \xrightarrow{\Omega}^i t'_1$. Therefore there exists $t' = t'_1 T_1$ such that $t \xrightarrow{\Omega}^i t'$.

Case T-FIX with $t = \mathbf{fix} t_1$

Sub-case $i = 0$ If t_1 is not a value, then by induction hypothesis there exists a t'_1 such that $t_1 \xrightarrow{\Omega}^0 t'_1$. Therefore by E-FIX there exists a t' such that $t \xrightarrow{\Omega}^0 t'$. Otherwise, if t_1 is a value, then $t_1 : T \rightarrow T$ by T-FIX. By the Lemma A.1 $t_1 = \lambda x:T.t_2$, therefore we can use E-FIX-RED to get a $t' = t_2[\mathbf{fix} \lambda x:T.t_2/x]$ such that $t \xrightarrow{\Omega}^0 t'$.

Sub-case $i > 1$ If t_1 is a value, then t is a value by definition V-FIX. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \xrightarrow{\Omega}^i t'_1$. Therefore there exists $t' = \mathbf{fix} t'_1$ such that $t \xrightarrow{\Omega}^i t'$.

Case T-QUOTE with $t = \lceil t_1 \rceil$ If t_1 is a value, then t is a value by definition V-QUOTE. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \longrightarrow_{\Omega}^{i+1} t'_1$. Therefore there exists $t' = \lceil t'_1 \rceil$ such that $t \longrightarrow_{\Omega}^i t'$.

Case T-SPLICE with $t = \lfloor t_1 \rfloor$

Sub-case $i = 0$ Term t cannot be typed.

Sub-case $i = 1$ If t_1 is a value, then by Lemma A.1 we know that $t_1 = \lceil t_2 \rceil$ for some t_2 , thus $t = \lfloor \lceil t_2 \rceil \rfloor$. By E-SPLICE-RED we can take a step $\lfloor \lceil t_2 \rceil \rfloor \longrightarrow_{\Omega}^1 t_2$. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \longrightarrow_{\Omega}^0 t'_1$. Therefore there exists $t' = \lfloor t'_1 \rfloor$ such that $t \longrightarrow_{\Omega}^1 t'$.

Sub-case $i \geq 2$ If t_1 is a value, then t is a value by definition V-SPLICE. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \longrightarrow_{\Omega}^{i-1} t'_1$. Therefore there exists $t' = \lfloor t'_1 \rfloor$ such that $t \longrightarrow_{\Omega}^i t'$.

Case T-LIFT with $t = \mathbf{lift} \ t_1$ As T-LIFT is the only typing rule that fits this case, the premise must be of the form $\Sigma \mid \Gamma^{\geq 1} \vdash^i \mathbf{lift} \ t_1 : \mathbf{[C]}$. From T-LIFT we can also deduce that $\Sigma \mid \Gamma^{\geq 1} \vdash^i t_1 : \mathbf{C}$.

Sub-case $i = 0$ and $\vdash^i t_1 \ \mathbf{vl}$ Given that $\Sigma \mid \Gamma^{\geq 1} \vdash^0 t_1 : \mathbf{C}$, by the Lemma A.1 we know that $t_1 = \mathbf{c}$ for some \mathbf{c} , therefore we can use E-LIFT-CONST to get a $t' = \mathbf{[c]}$ such that $t \longrightarrow_{\Omega}^0 t'$.

Sub-case $i > 0$ and $\vdash^i t_1 \ \mathbf{vl}$ Then t is a value by definition V-LIFT.

Otherwise The term t_1 is not a value. By induction hypothesis there exists a term t'_1 such that $t_1 \longrightarrow_{\Omega}^i t'_1$. Therefore there exists $t' = \mathbf{lift} \ t_1$ such that $t \longrightarrow_{\Omega}^i t'$.

Case T-UNLIFT with $t = \mathbf{unlift} \ t_1 \ \mathbf{with} \ t_2 \ \mathbf{or} \ t_3$ T-UNLIFT is the only typing rule that fits this case, the premise must be of the form $\Sigma \mid \Gamma^{\geq 1} \vdash^i \mathbf{unlift} \ t_1 \ \mathbf{with} \ t_2 \ \mathbf{or} \ t_3 : T$. From T-UNLIFT we can also deduce that $\Sigma \mid \Gamma^{\geq 1} \vdash^i t_1 : \mathbf{[C]}$, $\Sigma \mid \Gamma^{\geq 1} \vdash^i t_2 : \mathbf{C} \rightarrow T$ and $\Sigma \mid \Gamma^{\geq 1} \vdash^i t_3 : T$.

Sub-case $i = 0$ If t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \longrightarrow_{\Omega}^i t'_1$. Therefore there exists $t' = \mathbf{unlift} \ t'_1 \ \mathbf{with} \ t_2 \ \mathbf{or} \ t_3$ such that $t \longrightarrow_{\Omega}^i t'$. Otherwise, if t_1 is a value, it can be of the form $t_1 = \mathbf{[c]}$ where we can take a step with E-UNLIFT-SUCC; or it can be of the form $t_1 \neq \mathbf{[c]}$ where we can take a step with E-UNLIFT-FAIL.

Sub-case $i > 0$ If t_1, t_2 and t_3 are values, then t is a value by definition V-UNLIFT. Otherwise, if t_1 is not a value, then by induction hypothesis there exists a term t'_1 such that $t_1 \longrightarrow_{\Omega}^i t'_1$. Therefore there exists $t' = \mathbf{unlift} \ t'_1 \ \mathbf{with} \ t_2 \ \mathbf{or} \ t_3$ such that $t \longrightarrow_{\Omega}^i t'$. Otherwise, if t_2 is not a value, then by induction hypothesis there exists a term t'_2 such that $t_2 \longrightarrow_{\Omega}^i t'_2$. Therefore there exists $t' = \mathbf{unlift} \ t_1 \ \mathbf{with} \ t'_2 \ \mathbf{or} \ t_3$ such that $t \longrightarrow_{\Omega}^i t'$. Otherwise, if t_3 is not a value, then by induction hypothesis there exists a term t'_3 such that $t_3 \longrightarrow_{\Omega}^i t'_3$. Therefore there exists $t' = \mathbf{unlift} \ t_1 \ \mathbf{with} \ t_2 \ \mathbf{or} \ t'_3$ such that $t \longrightarrow_{\Omega}^i t'$.

Case T-MATCH $t = t_s \ \mathbf{match} \ \bar{X} \ \lceil t_p \rceil \ \mathbf{then} \ t_t \ \mathbf{else} \ t_e$

Sub-case $i = 0$ If t_s is not a value, then by induction hypothesis there exists a t'_s such that $t_s \longrightarrow_{\Omega}^0 t'_s$. Therefore by E-MATCH-SCRUT there exists a t' such that $t \longrightarrow_{\Omega}^0 t'$. Otherwise, if t_s is values, by Lemma A.1 using $t_s : \lceil T_1 \rceil$ from T-MATCH we know that $t_s = \lceil t_{s_2} \rceil$ for some t_{s_2} . If evaluation of $\bar{X} \vdash t_{s_2} \equiv t_p \Rightarrow \sigma$ can succeed then we can take a with E-MATCH-SUCC. Otherwise, if $\bar{X} \vdash t_{s_2} \equiv t_p \not\Rightarrow \sigma$, we can take a step with E-MATCH-FAIL.

Sub-case $i \geq 1$ If t_s is not a value, then by induction hypothesis there exists a t'_s such that $t_s \xrightarrow{i}_{\Omega} t'_s$. Therefore by E-MATCH-SCRUT there exists a $t' = t'_s \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t_e$ such that $t \xrightarrow{i}_{\Omega} t'$. Otherwise, if t_t is not a value, then by induction hypothesis there exists a t'_t such that $t_t \xrightarrow{i}_{\Omega} t'_t$. Therefore by E-MATCH-THEN there exists a $t' = t_s \text{ match } \bar{X} [t_p] \text{ then } t'_t \text{ else } t_e$ such that $t \xrightarrow{i}_{\Omega} t'$. Otherwise, if t_e is not a value, then by induction hypothesis there exists a t'_e such that $t_e \xrightarrow{i}_{\Omega} t'_e$. Therefore by E-MATCH-ELSE there exists a $t' = t_s \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t'_e$ such that $t \xrightarrow{i}_{\Omega} t'$. Otherwise, t_s , t_t and t_e are values and therefore t is a value by definition V-MATCH. ■

A.2 Proof of Preservation

Theorem A.3 (Preservation for Programs).

If $\Sigma \vdash p : T$, $\Sigma \vdash \Omega \text{ wf}$ and $p \mid \Omega \longrightarrow p' \mid \Omega'$,
then there exists a Σ' such that $\Sigma' \supseteq \Sigma$, $\Sigma' \vdash p' : T$ and $\Sigma' \vdash \Omega' \text{ wf}$

Proof.

Proof by case analysis on the type derivation of $\Sigma \vdash p : T$.

From the premises, we know that

$$\Sigma \vdash p : T \tag{1}$$

$$p \mid \Omega \longrightarrow p' \mid \Omega' \tag{2}$$

$$\Sigma \vdash \Omega \text{ wf} \tag{3}$$

Case T-EVAL $\Sigma \vdash \text{eval } t : T$ From T-EVAL we know that

$$\Sigma \mid \emptyset \vdash^0 t : T \tag{4}$$

The only evaluation step is E-EVAL, therefore

$$t \xrightarrow{0}_{\Omega} t' \tag{5}$$

$$p' = \text{eval } t' \tag{6}$$

$$\Omega' = \Omega \tag{7}$$

Therefore by Theorem A.4 using Eqs. (3) to (5) implies

$$\Sigma \mid \emptyset \vdash^0 t' : T \tag{8}$$

Using T-EVAL with premise Eq. (8) we get

$$\Sigma \vdash \text{eval } t' : T \tag{9}$$

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

From Eqs. (6) and (9) we get

$$\Sigma \vdash \mathbf{eval} \ p' : T \quad (10)$$

Using $\Sigma' = \Sigma$ and Eq. (7), we have that Eqs. (3) and (10) are equivalent to

$$\begin{aligned} \Sigma' \vdash \mathbf{eval} \ p' : T \\ \Sigma' \vdash \Omega' \ \mathbf{wf} \end{aligned}$$

Case T-DEF $\Sigma \vdash \mathbf{def} \ x = [t] \ \mathbf{in} \ p_1 : T$ From T-DEF we know

$$\Sigma | \emptyset \vdash^1 t : T_1 \quad (11)$$

$$\Sigma, x : T_1 \vdash p_1 : T \quad (12)$$

Sub-case E-MACRO

From E-Macro we know

$$p' = \mathbf{def} \ x = [t'] \ \mathbf{in} \ p_1 \quad (13)$$

$$t \xrightarrow{\Omega}^1 t' \quad (14)$$

$$\Omega' = \Omega \quad (15)$$

Therefore by Theorem A.4: Eqs. (3), (11) and (14) we have

$$\Sigma | \emptyset \vdash^1 t' : T_1 \quad (16)$$

From E-MACRO with Eqs. (12) and (16) we have

$$\Sigma \vdash \mathbf{def} \ x = [t'] \ \mathbf{in} \ p_1 : T \quad (17)$$

From Eqs. (13) and (17) we get

$$\Sigma \vdash p' : T \quad (18)$$

Using $\Sigma' = \Sigma$ and Eq. (15), we have that Eqs. (3) and (18) are equivalent to

$$\begin{aligned} \Sigma' \vdash \mathbf{eval} \ p' : T \\ \Sigma' \vdash \Omega' \ \mathbf{wf} \end{aligned}$$

Sub-case E-COMPILE

We E-COMPILE know that

$$p' = p_1 \quad (19)$$

$$\vdash^1 t \ \mathbf{v1} \quad (20)$$

$$\Omega' = \Omega, x := t \quad (21)$$

From Eqs. (12) and (19) we get

$$\Sigma, x : T_1 \vdash p' : T \quad (22)$$

From Eq. (3) and Definition A.1, we have that

$$\Sigma, x : T_1 \vdash \Omega, x := t \mathbf{wf} \quad (23)$$

Using $\Sigma' = \Sigma, x : T_1$ and Eq. (21), we have that Eqs. (22) and (23) are equivalent to

$$\begin{aligned} \Sigma' \vdash \mathbf{eval} \ p' : T \\ \Sigma' \vdash \Omega' \mathbf{wf} \end{aligned}$$

■

Lemma A.3 (Σ -Weakening).

If $\Sigma \vdash p : T$ and $\Sigma' \supseteq \Sigma$, then $\Sigma' \vdash p : T$

Proof.

Proof by straight-forward induction over the typing derivation. ■

Theorem A.4 (Preservation for Terms).

If $\Sigma \mid \Gamma \vdash^i t : T$, $t \longrightarrow_{\Omega}^i t'$ and $\Sigma \vdash \Omega \mathbf{wf}$, then $\Sigma \mid \Gamma \vdash^i t' : T$

Proof.

The proof proceeds by induction on the typing derivation, followed by inversion on the reduction step taken.

If $\Sigma \mid \Gamma \vdash^i t : T$, $t \longrightarrow_{\Omega}^i t'$ and $\Sigma \vdash \Omega \mathbf{wf}$ then

Case T-CONST $\Sigma \mid \Gamma \vdash^i c : C$ Cannot take a step.

Case T-VAR $\Sigma \mid \Gamma \vdash^i x : T$ Cannot take a step.

Case T-LINK $\Sigma \mid \Gamma \vdash^i x : T$ From T-LINK we know $x : T \in \Sigma$. The only congruence sub-case is E-LINK, which implies that $x \in \text{dom}(\Omega)$. Therefore from Definition A.1 we have $\Sigma \mid \emptyset \vdash^i \Omega(x) : \Sigma(x)$ which is equivalent to $\Sigma \mid \emptyset \vdash^i \Omega(x) : T$. By weakening we have $\Sigma \mid \Gamma \vdash^i \Omega(x) : T$.

Case T-ABS $\Sigma \mid \Gamma \vdash^i \lambda x : T_1. t_2 : T_1 \rightarrow T_2$ The congruence sub-case E-ABS immediately follows from the induction hypothesis.

Case T-TABS $\Sigma \mid \Gamma \vdash^i \Lambda X. t : \forall X. T_1$ The congruence sub-case E-TABS immediately follows from the induction hypothesis.

Case T-APP $\Sigma \mid \Gamma \vdash^i t_1 \ t_2 : T$ The congruence sub-cases E-APP-1 and E-APP-2 immediately follow from the induction hypothesis. To show sub-case E-BETA, we need to construct a typing derivation for $t_3[t_2/x]$, given $t_1 = \lambda x : T_2. t_3$ and $\vdash^0 t_2 \mathbf{v!}$, and premises (a) $\Sigma \mid \Gamma \vdash^0 \lambda x : T_2. t_3 : T_2 \rightarrow T$, (b) $\Sigma \mid \Gamma \vdash^0 t_2 : T_2$. From (a) we get (by inversion), (c) $\Sigma \mid \Gamma, x : T_2 \vdash^0 t_3 : T$. By applying the Lemma A.13 with (b) and (c), we obtain $\Sigma \mid \Gamma \vdash^0 t_3[t_2/x] : T$.

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

Case T-TAPP $\Sigma|\Gamma \vdash^i t_1 T_1 : T_2[T_1/X]$ The congruence sub-case E-TAPP immediately follows from the induction hypothesis. To show sub-case E-TBETA, we need to construct a typing derivation for $t_2[T_1/X]$, given $t_1 = \Lambda X.t_2$ and premises (a) $\Sigma|\Gamma \vdash^0 \Lambda X.t_2 : \forall X.T_2$. From T-TABS and (a) have the premise (b) $\Sigma|\Gamma, X \vdash^0 t_2 : T_2$. From T-TAPP we also assume that (c) $\Gamma \vdash T_1$ **wf**. By applying the Lemma A.16 with (b) and (c), we obtain $\Sigma|\Gamma \vdash^0 t_2[T_1/X] : T_2[T_1/X]$.

Case T-FIX $\Sigma|\Gamma \vdash^i \mathbf{fix} t : T$ The congruence sub-case E-FIX immediately follows from the induction hypothesis. To show sub-case E-FIX-RED, we need to construct a typing derivation for $t_1[\mathbf{fix} \lambda x:T.t_1/x]$, given $t = \lambda x:T.t_1$, and premises (a) $\Sigma|\Gamma \vdash^0 \lambda x:T.t_1 : T \rightarrow T$, (b) $\Sigma|\Gamma \vdash^i \mathbf{fix} \lambda x:T.t_1 : T$. From (a) we get (by inversion), (c) $\Gamma, \Sigma|x:^0 T \vdash^0 t_1 : T$. By applying the SUBSTITUTION LEMMA with (b) and (c), we obtain $\Sigma|\Gamma \vdash^0 t_1[\mathbf{fix} \lambda x:T.t_1/x] : T$.

Case T-QUOTE $\Sigma|\Gamma \vdash^i [t] : [T]$ The congruence sub-case E-QUOTE immediately follows from the induction hypothesis.

Case T-SPLICE $\Sigma|\Gamma \vdash^i [t_1] : T$ The congruence sub-case E-SPLICE immediately follows from the induction hypothesis. To show sub-case E-SPLICE-RED (where $i = 1$), we need to construct the typing derivation of t_1 where $t_1 = [t_2]$. From T-SPLICE we know that $\Sigma|\Gamma \vdash^0 [t_2] : [T]$, therefore by T-QUOTE we obtain $\Sigma|\Gamma \vdash^1 t_2 : T$

Case T-LIFT $\Sigma|\Gamma \vdash^i \mathbf{lift} t : [C]$ The congruence sub-case E-LIFT immediately follows from the induction hypothesis. To show sub-case E-LIFT-CONST we need to construct the typing derivation of $[c]$. We know that $\Sigma|\Gamma \vdash^1 c : C$ from T-CONST, therefore by T-QUOTE we have that $\Sigma|\Gamma \vdash^0 [c] : [C]$.

Case T-UNLIFT $\Sigma|\Gamma \vdash^i \mathbf{unlift} t_1 \mathbf{with} t_2 \mathbf{or} t_3 : T$ The congruence sub-cases E-UNLIFT-SCRUT, E-UNLIFT-WITH and E-UNLIFT-OR immediately follows from the induction hypothesis. The congruence sub-case E-UNLIFT-FAIL follows directly from T-UNLIFT. The congruence sub-case E-UNLIFT-SUCC follows from T-UNLIFT and T-APP.

Case T-MATCH $\Sigma|\Gamma \vdash^i t_s \mathbf{match} \overline{X} [t_p] \mathbf{then} t_t \mathbf{else} t_e : T$ The congruence sub-case E-MATCH-SCRUT immediately follows from the induction hypothesis. The congruence sub-case E-MATCH-FAIL immediately follows from T-MATCH. The congruence sub-case E-MATCH-SUCC immediately follows from the Lemma A.4. ■

Lemma A.4 (Preservation for Match).

If $\Sigma|\Gamma \vdash^0 [t_s] \mathbf{match} \overline{X}_i^i [t_p] \mathbf{then} t_t \mathbf{else} t_e : T$ and $\vdash^1 t_s \mathbf{vl}$ and $\overline{X}_i^i \vdash t_s \equiv t_p \Rightarrow \sigma$, then $\Sigma|\Gamma \vdash^0 \sigma(t_t) : T$

Proof.

Assuming premises

$$\vdash^1 t_s \mathbf{vl} \tag{1}$$

$$\Sigma|\Gamma \vdash^0 [t_s] \mathbf{match} \overline{X}_i^i [t_p] \mathbf{then} t_t \mathbf{else} t_e : T \tag{2}$$

$$\overline{X}_i^i \vdash t_s \equiv t_p \Rightarrow \sigma \tag{3}$$

From Eq. (2) we know

$$\frac{\Sigma |\Gamma \vdash^1 t_s : T_1 \quad \Sigma |\Gamma; \overline{X}_i^i | \emptyset \vdash^1 t_p : T_1 \dashv \Gamma_t \quad \Sigma |\Gamma; \overline{X}_i^i; \Gamma_t \vdash^0 t_t : T}{\Sigma |\Gamma \vdash^0 [t_s] \mathbf{match} \overline{X}_i^i [t_p] \mathbf{then} t_t \mathbf{else} t_e : T} \text{T-MATCH} \quad (4)$$

From Eq. (3) we know

$$\frac{\emptyset \vdash t_s \sqsupset t_p \Rightarrow \sigma_1 | C | \overline{X}_l^l \quad \overline{X}_l^l \vdash \mathit{unify}(\overline{X}_i^i | C) \Rightarrow \sigma_2 \quad \sigma = \sigma_2 \circ \sigma_1}{\overline{X}_i^i \vdash t_s \equiv t_p \Rightarrow \sigma} \quad (5)$$

By Definition 2 we have

$$\emptyset | \emptyset \vdash \emptyset \mathbf{wf} \quad (6)$$

From Lemma A.7 with $\Gamma_\delta = \emptyset$, $\Gamma_p = \emptyset$, $\Phi = \emptyset$, premises of Eq. (4), the first premise of Eq. (5) and Eq. (6) it follows that

$$\Sigma |\Gamma; \overline{X}_i^i \vdash^0 \sigma_1(t_t) : T \quad (7)$$

Using Lemma A.8 with $\Gamma_\delta = \emptyset$, $\Gamma_p = \emptyset$, the first two premises of Eq. (4), the first premise of Eq. (5) and Eq. (6) we get

$$\Gamma; \overline{X}_l^l | \overline{X}_i^i \vdash C \mathbf{wf} \quad (8)$$

Then using Lemma A.9 with the second premise of Eq. (5) and Eqs. (7) and (8) we get

$$\Sigma |\Gamma \vdash^0 \sigma_2(\sigma_1(t_t)) : T$$

which is equivalent to

$$\Sigma |\Gamma \vdash^0 \sigma_2 \circ \sigma_1(t_t) : T$$

■

Lemma A.5 (Well-Formed Constraint Shuffle).

If $\Gamma | \overline{X}_i^i, X; \overline{X}_j^j \vdash C \mathbf{wf}$ then $\Gamma | \overline{X}_i^i; \overline{X}_j^j, X \vdash C \mathbf{wf}$

Proof.

Perform induction on the constraint well-formedness derivation of constraint well formedness.

Case WFC-EMPTY C is \emptyset From the premise we have $\vdash \overline{X}_i^i, X; \overline{X}_j^j \mathbf{wf}$. Hence we also know that $\vdash \overline{X}_i^i; \overline{X}_j^j, X \mathbf{wf}$ and therefore $\Gamma | \overline{X}_i^i; \overline{X}_j^j, X \vdash \emptyset \mathbf{wf}$ holds by WFC-EMPTY.

Case WFC-EQ C is $C_2, T_1 = T_2$ Follows from induction hypothesis and Lemma A.11.

■

Lemma A.6 (Type Well-Formedness Weakening).

If $\Gamma \vdash T$ **wf** then $\Gamma, X \vdash T$ **wf**

Proof.

Proof by straight-forward induction over the typing derivation. ■

Definition A.3 (Well-formedness of Φ).

We say Φ is well formed with respect to Γ_p and Γ_δ , written $\Gamma_p \mid \Gamma_\delta \vdash \Phi$ **wf**, if and only if Φ is a bijection between $\text{dom}(\Gamma_p)$ and $\text{dom}(\Gamma_\delta)$, such that $\text{dom}(\Gamma_p) \cap \text{dom}(\Gamma_\delta) = \emptyset$ and $\forall x_p :^1 T \in \Gamma_p. \Phi(x_p) :^1 T \in \Gamma_\delta$.

Lemma A.7 (Preservation of Pattern Reduction).

If Eqs. (1) to (5) hold, then $\Sigma \mid \Gamma \vdash^0 \sigma(t) : T$

$$\Sigma \mid \Gamma; \Gamma_\delta \vdash^1 t_s : T_s \tag{1}$$

$$\Sigma \mid \Gamma \mid \Gamma_p \vdash^1 t_p : T_p \dashv \Gamma_t \tag{2}$$

$$\Sigma \mid \Gamma; \Gamma_t \vdash^0 t : T \tag{3}$$

$$\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X}_t^l \tag{4}$$

$$\Gamma_p \mid \Gamma_\delta \vdash \Phi \text{ wf} \tag{5}$$

Proof.

From the premise of the Lemma we assume that Eqs. (1) to (5) hold. The proof is performed by induction on the typing derivations of the patterns $\Sigma \mid \Gamma \mid \Gamma_p \vdash^1 t_p : T_p \dashv \Gamma_t$.

Case T-PAT-CONST $\Sigma \mid \Gamma \mid \Gamma_p \vdash^1 \mathbf{c} : \mathbf{C} \dashv \emptyset$

The only pattern rule that applies is E-PAT-CONST which implies that

$$\sigma = [] \tag{6}$$

As $\Gamma_t = \emptyset$, we have that

$$\Sigma \mid \Gamma \vdash^0 t : T \tag{7}$$

Eqs. (6) and (7) imply

$$\Sigma \mid \Gamma \vdash^0 \sigma(t) : T$$

Case T-PAT-VAR $\Sigma \mid \Gamma \mid \Gamma_p \vdash^1 x : T_p \dashv \emptyset$

From T-PAT-VAR we know that

$$\Gamma_t = \emptyset \tag{8}$$

Eq. (4) can be one of the follow two cases:

Sub-case E-PAT-VAR with $\Phi \vdash \Phi(x_p) \sqsupset x_p \Rightarrow \sigma \mid \emptyset \mid \overline{X}_l^l$

From E-PAT-VAR we know that

$$\sigma = [] \quad (9)$$

Eqs. (3) and (8) imply

$$\Sigma \mid \Gamma \vdash^0 t : T \quad (10)$$

Eqs. (9) and (10) imply

$$\Sigma \mid \Gamma \vdash^0 \sigma(t) : T$$

Sub-case E-PAT-LINK As $x : ^i T_s \in \Gamma$ implies that $x : ^i T_s \notin \Gamma_\delta$ by Eq. (5), we have that $t_p \neq x$ which implies that this rule cannot be applied.

Case T-PAT-LINK $\Sigma \mid \Gamma \mid \Gamma_p \vdash^1 x : T_p \dashv \emptyset$

From T-PAT-LINK we know that

$$\Gamma_t = \emptyset \quad (11)$$

Eq. (4) can be one of the follow two:

Sub-case E-PAT-VAR with $\Phi \vdash \Phi(x_p) \sqsupset x_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l$ As $x : T_s \in \Sigma$ implies that $x : ^i T_s \notin \Gamma_\delta$ by Eq. (5), therefore there is no $\Phi(x)$ which implies that this rule cannot be applied.

Sub-case E-PAT-LINK

From E-PAT-LINK we know that

$$\sigma = [] \quad (12)$$

Eqs. (3) and (11) imply

$$\Sigma \mid \Gamma \vdash^0 t : T \quad (13)$$

Eqs. (12) and (13) imply

$$\Sigma \mid \Gamma \vdash^0 \sigma(t) : T$$

Case T-PAT-ABS $\Sigma \mid \Gamma \mid \Gamma_p \vdash^1 \lambda x_p : T_{p_1}. t_{p_2} : T_{p_1} \dashv T_{p_2} \dashv \Gamma_t$

The only pattern rule that applies is E-PAT-ABS therefore from Eq. (4) we know

$$\frac{\Phi, x_p \mapsto x_s \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma \mid C_1 \mid \overline{X}_l^l}{\Phi \vdash \lambda x_s : T_{s_1}. t_{s_2} \sqsupset \lambda x_p : T_{p_1}. t_{p_2} \Rightarrow \sigma \mid C_1, T_{s_1} = T_{p_1} \mid \overline{X}_l^l} \text{E-PAT-ABS} \quad (14)$$

From the premise (T-ABS) of Eq. (1) we know

$$\Sigma \mid \Gamma; \Gamma_\delta, x_s : ^1 T_{s_1} \vdash^1 t_{s_2} : T_{s_2} \quad (15)$$

From the premise (T-PAT-ABS) of Eq. (2) we know

$$\Sigma \mid \Gamma \mid \Gamma_p, x_p : ^1 T_{p_1} \vdash^1 t_{p_2} : T_{p_2} \dashv \Gamma_t \quad (16)$$

From Eq. (5) and Definition A.3 and the $T_{s_1} = T_{p_1}$ constraint we know that

$$\Gamma_p, x_p : ^1 T_{p_1} \mid \Gamma_\delta, x_s : ^1 T_{s_1} \vdash \Phi, x_p \mapsto x_s \text{ wf} \quad (17)$$

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

Using the induction hypothesis with premise of Eq. (14) and Eqs. (3) and (15) to (17) we get

$$\Sigma | \Gamma \vdash^0 \sigma(t) : T$$

Case T-PAT-APP $\Sigma | \Gamma | \Gamma_p \vdash^1 t_{p_1} t_{p_2} : T_p \dashv \Gamma_{t_1}; \Gamma_{t_2}$

The only pattern rule that applies to this type derivation is E-PAT-APP and therefore from Eq. (4) we know

$$\frac{\Phi \vdash t_{s_1} \sqsupset t_{p_1} \Rightarrow \sigma_1 | C_1 | \overline{X_{l_1}^{l_1}} \quad \Phi \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma_2 | C_2 | \overline{X_{l_2}^{l_2}}}{\Phi \vdash t_{s_1} t_{s_2} \sqsupset t_{p_1} t_{p_2} \Rightarrow \sigma_2 \circ \sigma_1 | C_1; C_2 | \overline{X_{l_1}^{l_1}}; \overline{X_{l_2}^{l_2}}} \text{E-PAT-APP} \quad (18)$$

We will first prove that $\Sigma | \Gamma; \Gamma_{t_2} \vdash^0 \sigma_1(t) : T$ using induction hypothesis over the following statements:

- By T-APP we have $\Sigma | \Gamma; \Gamma_\delta \vdash^1 t_{s_1} : T_{s_2} \rightarrow T_s$
which can be weakened to $\Sigma | (\Gamma; \Gamma_{t_2}); \Gamma_\delta \vdash^1 t_{s_1} : T_{s_2} \rightarrow T_s$
- By T-PAT-APP we have $\Sigma | \Gamma | \Gamma_p \vdash^1 t_{p_1} : T_{p_2} \rightarrow T_p \dashv \Gamma_{t_1}$
which can be weakened to $\Sigma | (\Gamma; \Gamma_{t_2}) | \Gamma_p \vdash^1 t_{p_1} : T_{p_2} \rightarrow T_{p_3} \dashv \Gamma_{t_1}$
- As $\Gamma_t = \Gamma_{t_1}; \Gamma_{t_2}$, Eq. (3) is equivalent to $\Sigma | (\Gamma; \Gamma_{t_2}); \Gamma_{t_1} \vdash^0 t : T$
- From the first premise of Eq. (18) we know $\Phi \vdash t_{s_1} \sqsupset t_{p_1} \Rightarrow \sigma | C_1 | \overline{X_{l_1}^{l_1}}$
- From Eq. (5) we know $\Gamma_p | \Gamma_\delta \vdash \Phi \mathbf{wf}$

and therefore we have that

$$\Sigma | \Gamma; \Gamma_{t_2} \vdash^0 \sigma_1(t) : T \quad (19)$$

Now we will prove that $\Sigma | \Gamma \vdash^0 \sigma_2 \circ \sigma_1(t) : T$ using induction hypothesis over the following statements:

- By T-APP we have $\Sigma | \Gamma; \Gamma_\delta \vdash^1 t_{s_1} : T_{s_2} \rightarrow T_s$
- By T-PAT-APP we have $\Sigma | \Gamma | \Gamma_p \vdash^1 t_{p_2} : T_{p_2} \rightarrow T_p \dashv \Gamma_{t_2}$
- From Eq. (19) we have $\Sigma | \Gamma; \Gamma_{t_2} \vdash^0 \sigma_1(t) : T$
- From the second premise of Eq. (18) we have $\Phi \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma_2 | C_2 | \overline{X_{l_2}^{l_2}}$
- From Eq. (5) we have $\Gamma_p | \Gamma_\delta \vdash \Phi \mathbf{wf}$

and therefore we have that

$$\Sigma | \Gamma \vdash^0 \sigma_2 \circ \sigma_1(t) : T$$

Case T-PAT-TABS $\Sigma | \Gamma | \Gamma_p \vdash^1 \Lambda X_p. t_{p_2} : \forall X_p. T_{p_2} \dashv \Gamma_t$

The only pattern rule that applies is E-PAT-TABS therefore from Eq. (4) we know

$$\frac{\Phi, X_p \mapsto X_s \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma | C_1 | \overline{X_l^l}}{\Phi \vdash \Lambda X_s. t_{s_2} \sqsupset \Lambda X_p. t_{p_2} \Rightarrow \sigma | C_1 [X_s / X_p] | \overline{X_l^l}, X_s} \text{E-PAT-TABS} \quad (20)$$

From the premise of T-TABS, Eq. (1) implies

$$\Sigma | \Gamma; \Gamma_\delta, X_s \vdash^1 t_{s_2} : T_{s_2} \quad (21)$$

From the premise of T-PAT-TABS, Eq. (2) implies

$$\Sigma |\Gamma | \Gamma_p, X_p \vdash^1 t_{p_2} : T_{p_2} \dashv \Gamma_t \quad (22)$$

From Eq. (5) and Definition A.3 we know that

$$\Gamma_p, X_p | \Gamma_\delta, X_s \vdash \Phi, X_p \mapsto X_s \mathbf{wf} \quad (23)$$

Using the induction hypothesis with premise of Eq. (20) and Eqs. (3) and (21) to (23) we get

$$\Sigma |\Gamma \vdash^0 \sigma(t) : T$$

Case T-PAT-TAPP $\Sigma |\Gamma | \Gamma_p \vdash^1 t_{p_1} : T_{p_1} [T_p/X] \dashv \Gamma_t$

The only pattern rule that applies is E-PAT-TAPP therefore Eq. (4) is equivalent to

$$\frac{\Phi \vdash t_{s_1} \boxtimes t_{p_1} \Rightarrow \sigma | C_1 | \overline{X}_l^l}{\Phi \vdash t_{s_1} : T_{s_1} \boxtimes t_{p_1} : T_{p_1} \Rightarrow \sigma | C_1, T_{s_1} = T_{p_1} | \overline{X}_l^l} \text{E-PAT-TAPP} \quad (24)$$

From the premise of T-TAPP, Eq. (1) implies

$$\Sigma |\Gamma; \Gamma_\delta \vdash^1 t_{s_1} : \forall X_s. T_{s_2} \quad (25)$$

From the premise of T-PAT-TAPP, Eq. (2) implies

$$\Sigma |\Gamma | \Gamma_p \vdash^1 t_{p_1} : \forall X_p. T_{p_2} \dashv \Gamma_t \quad (26)$$

Using the induction hypothesis with premise of Eq. (24) and Eqs. (3), (5), (25) and (26) we get

$$\Sigma |\Gamma \vdash^0 \sigma(t) : T$$

Case T-PAT-FIX $\Sigma |\Gamma | \Gamma_p \vdash^1 \mathbf{fix} \ t : T_p \dashv \Gamma_t$ The only $\Phi \vdash t_s \boxtimes t_p \Rightarrow \sigma | C | \overline{X}_l^l$ rule that applies is E-PAT-FIX, the proof follows directly from the induction hypothesis.

Case T-PAT-BIND

From Eq. (2) we know

$$\frac{\overline{X}_j \in \Gamma_p^j \quad \overline{x}_k :^1 T_k \in \Gamma_p^k}{\Sigma |\Gamma | \Gamma_p \vdash^1 \llbracket x \rrbracket_{T_p}^j \overline{x}_k : T_k^k : T_p \dashv \emptyset, x :^0 \forall X_j. \overline{[T_k]} \rightarrow^k [T_p]} \text{T-PAT-BIND} \quad (27)$$

The only pattern rule that applies is E-PAT-BIND therefore

$$\sigma = \overline{[\forall X_j. \left(\overline{[\lambda x'_k : [T_k]]} \cdot [t_s] \overline{[x'_k] / \Phi(x_k)} \right) [X_j / \Phi(X_j)]^j / x]} \quad (28)$$

$$FV(t_s) \cap \text{range}(\Phi) \subseteq \overline{\Phi(X_j)}^j; \overline{\Phi(x_k)}^k \quad (29)$$

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

From Eq. (3) we know

$$\Sigma |\Gamma, x :^0 \overline{\forall X_j}^j \overline{[T_k]}^k \rightarrow [T_p] \vdash^0 t : T \quad (30)$$

We trivially know that

$$\overline{x'_k :^0 [T_k] \in \Gamma; \overline{X_j}^j; \overline{x'_k :^0 [T_k]}^k; \overline{\Phi(X_j)}^j}^k \quad (31)$$

Note the nested repetition in $\dots \overline{x'_k :^0 [T_k]}^k \dots$. This implies that we have one \in for each k and each one has an environment containing a $x'_k :^0 [T_k]$ for each k .

Therefore from Eq. (31) can derive

$$\frac{\overline{x'_k :^0 [T_k] \in \Gamma; \overline{X_j}^j; \overline{x'_k :^0 [T_k]}^k; \overline{\Phi(X_j)}^j}^k}{\Sigma |\Gamma; \overline{X_j}^j; \overline{x'_k :^0 [T_k]}^k; \overline{\Phi(X_j)}^j \vdash^0 x'_k : [T_k]} \text{T-VAR} \quad \frac{\Sigma |\Gamma; \overline{X_j}^j; \overline{x'_k :^0 [T_k]}^k; \overline{\Phi(X_j)}^j \vdash^0 x'_k : [T_k]}{\Sigma |\Gamma; \overline{X_j}^j; \overline{x'_k :^0 [T_k]}^k; \overline{\Phi(X_j)}^j \vdash^1 [x'_k] : T_k} \text{T-SPLICE} \quad (32)$$

From $\overline{x_k :^1 T_k \in \Gamma_p}^k$ and Eq. (5) we know that for every k

$$\overline{\Phi(x_k) :^1 T_k \in \Gamma_\delta}^k \quad (33)$$

From $\overline{X_j \in \Gamma_p}^j$ and Eq. (5) we know that for every j

$$\overline{\Phi(X_j) \in \Gamma_\delta}^j \quad (34)$$

From Eqs. (1), (29), (33) and (34) we can deduce that

$$\Sigma |\Gamma; \overline{\Phi(X_j)}^j; \overline{\Phi(x_k) :^1 T_k}^k \vdash^1 t_s : T_s \quad (35)$$

From T-QUOTE and Eq. (35) we get

$$\Sigma |\Gamma; \overline{\Phi(X_j)}^j; \overline{\Phi(x_k) :^1 T_k}^k \vdash^0 [t_s] : [T_s] \quad (36)$$

By weakening Eq. (36) we can get

$$\Sigma |\Gamma; \overline{X_j}^j; \overline{x'_k :^0 [T_k]}^k; \overline{\Phi(X_j)}^j; \overline{\Phi(x_k) :^1 T_k}^k \vdash^0 [t_s] : [T_s] \quad (37)$$

Using Lemma A.14 with Eqs. (32) and (37) we get

$$\Sigma |\Gamma; \overline{X_j}^j; \overline{x'_k :^0 [T_k]}^k; \overline{\Phi(X_j)}^j \vdash^0 [t_s] [\overline{[x'_k] / \Phi(x_k)}^k] : [T_s] \quad (38)$$

For every j , we can use WFT-VAR to instantiate

$$\overline{\Gamma; X_j \vdash X'_j \mathbf{wf}^j} \quad (39)$$

Eq. (39) can be weakened to

$$\overline{\Gamma; \overline{X}_j^j; x'_k :^0 [T_k]^k \vdash X'_j \mathbf{wf}}^j \quad (40)$$

Using Lemma A.18 with Eqs. (38) and (40) we get

$$\Sigma | \Gamma; \overline{X}_j^j; x'_k :^0 [T_k]^k \vdash^0 [t_s] [[x'_k] / \Phi(x_k)]^k [\overline{X}_j^j / \Phi(X_j)]^j : [T_s] \quad (41)$$

From Eq. (41) we can derive

$$\frac{\frac{\Sigma | \Gamma; \overline{X}_j^j; x'_k :^0 [T_k]^k \vdash^0 [t_s] [[x'_k] / \Phi(x_k)]^k [\overline{X}_j^j / \Phi(X_j)]^j : [T_s]}{\Sigma | \Gamma; \overline{X}_j^j \vdash^0 \lambda x'_k : [T_k]^k . [t_s] [[x'_k] / \Phi(x_k)]^k : [T_k] \rightarrow^k [T_s]} \text{T-ABS}}{\Sigma | \Gamma \vdash^0 \forall X_j .^j \left(\lambda x'_k : [T_k]^k . [t_s] [[x'_k] / \Phi(x_k)]^k \right) [\overline{X}_j^j / \Phi(X_j)]^j : \forall X_j .^j [T_k] \rightarrow^k [T_s]} \text{T-TABS}} \quad (42)$$

Using Lemma A.13 with Eqs. (30) and (42) we finally get

$$\Sigma | \Gamma \vdash^0 \sigma(t) : T$$

■

Lemma A.8 (Constraint of Pattern Reduction).

If Eqs. (1) to (4) then $\Gamma; \overline{X}_l^l; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash C \mathbf{wf}$

$$\Sigma | \Gamma; \Gamma_\delta \vdash^1 t_s : T_s \quad (1)$$

$$\Sigma | \Gamma; \overline{X}_i^i | \Gamma_p \vdash^1 t_p : T_p \dashv \Gamma_t \quad (2)$$

$$\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma | C | \overline{X}_l^l \quad (3)$$

$$\Gamma_p | \Gamma_\delta \vdash \Phi \mathbf{wf} \quad (4)$$

Proof.

From the premise of the Lemma we assume that Eqs. (1) to (4) hold. The proof is performed by induction on the typing derivations of the patterns $\Sigma | \Gamma; \overline{X}_i^i | \Gamma_p \vdash^1 t_p : T_p \dashv \Gamma_t$.

Case T-PAT-CONST $\Sigma | \Gamma; \overline{X}_i^i | \Gamma_p \vdash^1 \mathbf{c} : C \dashv \emptyset$ The only pattern rule that applies is E-PAT-CONST which implies that $C = \emptyset$ and $\overline{X}_l^l = \emptyset$. From Eq. (2) we also have that $\vdash \Gamma; \overline{X}_i^i; \Gamma_p \mathbf{wf}$ which can be weakened to $\vdash \Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \mathbf{wf}$. Therefore by WFC-EMPTY we can derive $\Gamma; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash \emptyset \mathbf{wf}$ which is equivalent to $\Gamma; \overline{X}_l^l; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash C \mathbf{wf}$.

Case T-PAT-VAR $\Sigma | \Gamma; \overline{X}_i^i | \Gamma_p \vdash^1 x : T_p \dashv \emptyset$ The only pattern rule that applies is E-PAT-VAR which implies that $C = \emptyset$ and $\overline{X}_l^l = \emptyset$. From Eq. (2) we also have that $\vdash \Gamma; \overline{X}_i^i; \Gamma_p \mathbf{wf}$ which can be

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

weakened to $\vdash \Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \mathbf{wf}$. Therefore by WFC-EMPTY we can derive $\Gamma; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash \emptyset \mathbf{wf}$ which is equivalent to $\Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C \mathbf{wf}$.

Case T-PAT-LINK $\Sigma \mid \Gamma; \overline{X}_i^i \mid \Gamma_p \vdash^1 x : T_p \dashv \emptyset$ The only pattern rule that applies is E-PAT-LINK which implies that $C = \emptyset$. From Eq. (2) we also have that $\vdash \Gamma; \overline{X}_i^i; \Gamma_p \mathbf{wf}$ which can be weakened to $\vdash \Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \mathbf{wf}$. Therefore by WFC-EMPTY we can derive $\Gamma; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash \emptyset \mathbf{wf}$ which is equivalent to $\Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C \mathbf{wf}$.

Case T-PAT-ABS

In this we have

$$\frac{\Sigma \mid \Gamma; \overline{X}_i^i \mid \Gamma_p, x_p :^1 T_{p_2} \vdash^1 t_{p_2} : T_p \dashv \Gamma_t \quad \Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_{p_2} \mathbf{wf}}{\Sigma \mid \Gamma; \overline{X}_i^i \mid \Gamma_p \vdash^1 \lambda x_p : T_{p_2}. t_{p_2} : T_{p_2} \dashv T_{p_1} \dashv \Gamma_t} \text{T-PAT-ABS} \quad (5)$$

The only pattern rule that applies is E-PAT-ABS therefore from Eq. (3) we know

$$\frac{\Phi, x_p \mapsto x_s \vdash t_{s_2} \boxtimes t_{p_2} \Rightarrow \sigma \mid C_1 \mid \overline{X}_l^l}{\Phi \vdash \lambda x_s : T_{s_2}. t_{s_2} \boxtimes \lambda x_p : T_{p_2}. t_{p_2} \Rightarrow \sigma \mid C_1, T_{s_2} = T_{p_2} \mid \overline{X}_l^l} \text{E-PAT-ABS} \quad (6)$$

From Eq. (1) we know

$$\frac{\Sigma \mid \Gamma; \Gamma_\delta, x_s :^1 T_{s_2} \vdash^1 t_{s_2} : T_s \quad \Gamma; \Gamma_\delta \vdash T_{s_2} \mathbf{wf}}{\Sigma \mid \Gamma; \Gamma_\delta \vdash^1 \lambda x_s : T_{s_2}. t_{s_2} : T_s} \text{T-ABS} \quad (7)$$

From Eq. (4) and Definition A.3 and the $T_{s_2} = T_{p_2}$ constraint we know that

$$\Gamma_p, x_p :^1 T_{p_2} \mid \Gamma_\delta, x_s :^1 T_{s_2} \vdash \Phi, x_p \mapsto x_s \mathbf{wf} \quad (8)$$

From the induction hypothesis with premises of Eqs. (5) to (7) and Eq. (8) we get

$$\Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C_1 \mathbf{wf} \quad (9)$$

$\Gamma; \Gamma_\delta \vdash T_{s_2} \mathbf{wf}$ and $\Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_{p_2} \mathbf{wf}$ can be weakened to

$$\Gamma; \overline{X}_l^l; \Gamma_\delta \vdash T_{s_2} \mathbf{wf} \quad (10)$$

$$\Gamma; \overline{X}_l^l; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_{p_2} \mathbf{wf} \quad (11)$$

Therefore using WFC-EQ we can derive

$$\frac{\Gamma; \overline{X}_l^l; \Gamma_\delta \vdash T_{s_2} \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_{p_2} \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C_1 \mathbf{wf}}{\Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C_1, T_{s_2} = T_{p_2} \mathbf{wf}}$$

Case T-PAT-APP

In this case we have

$$\frac{\Sigma |\Gamma; \overline{X}_i^i | \Gamma_p \vdash^1 t_{p_1} : T_{p_2} \rightarrow T_p \dashv \Gamma_{t_1} \quad \Sigma |\Gamma; \overline{X}_i^i | \Gamma_p \vdash^1 t_{p_2} : T_{p_2} \dashv \Gamma_{t_2}}{\Sigma |\Gamma; \overline{X}_i^i | \Gamma_p \vdash^1 t_{p_1} t_{p_2} : T_p \dashv \Gamma_t} \text{T-PAT-APP} \quad (12)$$

The only pattern rule that applies to this type derivation is E-PAT-APP and therefore from Eq. (3) we know

$$\frac{\Phi \vdash t_{s_1} \sqsupset t_{p_1} \Rightarrow \sigma_1 | C_1 | \overline{X}_{l_1}^{l_1} \quad \Phi \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma_2 | C_2 | \overline{X}_{l_2}^{l_2}}{\Phi \vdash t_{s_1} t_{s_2} \sqsupset t_{p_1} t_{p_2} \Rightarrow \sigma_2 \circ \sigma_1 | C_1; C_2 | \overline{X}_{l_1}^{l_1}; \overline{X}_{l_2}^{l_2}} \text{E-PAT-APP} \quad (13)$$

From Eq. (1) we know

$$\frac{\Sigma |\Gamma; \Gamma_\delta \vdash^1 t_{s_1} : T_{s_2} \rightarrow T_s \quad \Sigma |\Gamma; \Gamma_\delta \vdash^1 t_{s_2} : T_{s_2}}{\Sigma |\Gamma; \Gamma_\delta \vdash^1 t_{s_1} t_{s_2} : T_s} \text{T-APP} \quad (14)$$

Using the induction hypothesis with the first premises of Eqs. (12) to (14) and Eq. (4) we get

$$\Gamma; \overline{X}_{l_1}^{l_1}; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash C_1 \mathbf{wf} \quad (15)$$

Eq. (15) can be weakened to

$$\Gamma; \overline{X}_{l_1}^{l_1}; \overline{X}_{l_2}^{l_2}; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash C_1 \mathbf{wf} \quad (16)$$

Using the induction hypothesis with second premises of Eqs. (12) to (14) and Eq. (4) we get

$$\Gamma; \overline{X}_{l_2}^{l_2}; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash C_2 \mathbf{wf} \quad (17)$$

Eq. (17) can be weakened to

$$\Gamma; \overline{X}_{l_1}^{l_1}; \overline{X}_{l_2}^{l_2}; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash C_2 \mathbf{wf} \quad (18)$$

Using Lemma A.12 with Eqs. (16) and (18) we get

$$\Gamma; \overline{X}_{l_1}^{l_1}; \overline{X}_{l_2}^{l_2}; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash C_1; C_2 \mathbf{wf}$$

which is equivalent to

$$\Gamma; \overline{X}_l^l; \Gamma_\delta | \overline{X}_i^i; \Gamma_p \vdash C \mathbf{wf}$$

Case T-PAT-TABS

In this case we have

$$\frac{\Sigma |\Gamma | \Gamma_p, X_p \vdash^1 t_{p_2} : T_p \dashv \Gamma_t}{\Sigma |\Gamma; \overline{X}_i^i | \Gamma_p \vdash^1 \Lambda X_p. t_{p_2} : T_p \dashv \Gamma_t} \text{T-PAT-TABS} \quad (19)$$

The only pattern rule that applies is E-PAT-TABS therefore from Eq. (3) we know

$$\frac{\Phi, X_p \mapsto X_s \vdash t_{s_2} \sqsupset t_{p_2} \Rightarrow \sigma | C_1 | \overline{X}_{l_1}^{l_1}}{\Phi \vdash \Lambda X_s. t_{s_2} \sqsupset \Lambda X_p. t_{p_2} \Rightarrow \sigma | C_1 [X_s / X_p] | \overline{X}_{l_1}^{l_1}, X_s} \text{E-PAT-TABS} \quad (20)$$

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

From Eq. (1) we know

$$\frac{\Sigma |\Gamma; \Gamma_\delta, X_s \vdash^1 t_{s_2} : T_s}{\Sigma |\Gamma; \Gamma_\delta \vdash^1 t_s : T_s} \text{T-TABS} \quad (21)$$

From Eq. (4) and Definition A.3 we know that

$$\Gamma_p, X_p \mid \Gamma_\delta, X_s \vdash \Phi, X_p \mapsto X_s \mathbf{wf} \quad (22)$$

From the induction hypothesis with premises of Eqs. (19) to (21) and Eq. (22) we get

$$\Gamma; \overline{X_{l_1}}^l; \Gamma_\delta, X_s \mid \overline{X_i}^i; \Gamma_p, X_p \vdash C_1 \mathbf{wf} \quad (23)$$

By substituting X_p with X_s in Eq. (23) we get that

$$\Gamma; \overline{X_{l_1}}^l; \Gamma_\delta, X_s \mid \overline{X_i}^i; \Gamma_p \vdash C_1[X_s/X_p] \mathbf{wf} \quad (24)$$

Eq. (24) can be weakened to

$$\Gamma; \overline{X_{l_1}}^l, X_s; \Gamma_\delta \mid \overline{X_i}^i; \Gamma_p \vdash C_1[X_s/X_p] \mathbf{wf}$$

which is equivalent to

$$\Gamma; \overline{X_l}^l; \Gamma_\delta \mid \overline{X_i}^i; \Gamma_p \vdash C \mathbf{wf}$$

Case T-PAT-TAPP $\Sigma |\Gamma; \overline{X_i}^i \mid \Gamma_p \vdash^1 t_{p_1} T_{p_2} : T_p \dashv \Gamma_t$

The only pattern rule that applies is E-PAT-TAPP therefore Eq. (3) is equivalent to

$$\frac{\Phi \vdash t_{s_1} \sqsupset t_{p_1} \Rightarrow \sigma \mid C_1 \mid \overline{X_l}^l}{\Phi \vdash t_{s_1} T_{s_2} \sqsupset t_{p_1} T_{p_2} \Rightarrow \sigma \mid C_1, T_{s_2} = T_{p_2} \mid \overline{X_l}^l} \quad (25)$$

From Eq. (1) we know

$$\frac{\Sigma |\Gamma; \Gamma_\delta \vdash^1 t_{s_1} : T_{s_1} \quad \Gamma; \Gamma_\delta \vdash T_s \mathbf{wf}}{\Sigma |\Gamma; \Gamma_\delta \vdash^1 t_{s_1} T_{s_2} : T_s} \quad (26)$$

From Eq. (2) we know

$$\frac{\Sigma |\Gamma \mid \Gamma_p \vdash^1 t_{p_1} : T_{p_1} \dashv \Gamma_t \quad \Gamma; \Gamma_\delta; \overline{X_i}^i; \Gamma_p \vdash T_p \mathbf{wf}}{\Sigma |\Gamma; \overline{X_i}^i \mid \Gamma_p \vdash^1 t_{p_1} T_{p_2} : T_p \dashv \Gamma_t} \text{T-PAT-TAPP} \quad (27)$$

Using the induction hypothesis with premises of Eqs. (25) to (27) and Eq. (4) we get

$$\Gamma; \overline{X_l}^l; \Gamma_\delta \mid \overline{X_i}^i; \Gamma_p \vdash C_1 \mathbf{wf} \quad (28)$$

$\Gamma; \Gamma_\delta \vdash T_s \mathbf{wf}$ and $\Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_p \mathbf{wf}$ can be weakened to

$$\Gamma; \overline{X}_l^l; \Gamma_\delta \vdash T_{s_2} \mathbf{wf} \quad (29)$$

$$\Gamma; \overline{X}_l^l; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_{p_2} \mathbf{wf} \quad (30)$$

Therefore we can derive

$$\frac{\Gamma; \overline{X}_l^l; \Gamma_\delta \vdash T_{s_2} \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_{p_2} \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C_1 \mathbf{wf}}{\Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C_1, T_{s_2} = T_{p_2} \mathbf{wf}}$$

which is equivalent to

$$\Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C \mathbf{wf}$$

Case T-PAT-FIX The only $\Phi \vdash t_s \sqsupset t_p \Rightarrow \sigma \mid C \mid \overline{X}_l^l$ rule that applies is E-PAT-FIX, the proof follows directly from the induction hypothesis.

Case T-PAT-BIND

We know that

$$\frac{\dots \quad \Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_p \mathbf{wf} \quad \vdash \Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \mathbf{wf}}{\Sigma \mid \Gamma; \overline{X}_i^i \mid \Gamma_p \vdash^1 \llbracket x \rrbracket_{T_p}^{\overline{X}_j^j \overline{x}_k: \overline{T}_k^k} : T_p \dashv \Gamma_t \emptyset, x :^0 \forall \overline{X}_j^j \llbracket T_k \rrbracket \rightarrow^k \llbracket T_1 \rrbracket} \quad (31)$$

Using premise of Eq. (31) we can derive

$$\frac{\vdash \Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \mathbf{wf}}{\Gamma; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash \emptyset \mathbf{wf}} \text{WFC-EMPTY} \quad (32)$$

The only pattern rule that applies is E-PAT-BIND therefore Eq. (3) is equivalent to

$$\Phi \vdash t_s \sqsupset \llbracket x \rrbracket_{T_p}^{\overline{X}_j^j \overline{x}_k: \overline{T}_k^k} \Rightarrow [t'_s / x] \mid \{type(t_s) = T_p\} \mid \emptyset \quad (33)$$

From Eq. (1) we know that

$$\Gamma; \Gamma_\delta \vdash T_s \mathbf{wf} \quad (34)$$

From Eq. (1) we also know the type of t_s is T_s , therefore Eq. (34) is equivalent to

$$\Gamma; \Gamma_\delta \vdash type(t_s) \mathbf{wf} \quad (35)$$

Therefore we can derive

$$\frac{\Gamma; \Gamma_\delta \vdash type(t_s) \mathbf{wf} \quad \Gamma; \Gamma_\delta; \overline{X}_i^i; \Gamma_p \vdash T_p \mathbf{wf} \quad \Gamma; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash \emptyset \mathbf{wf}}{\Gamma; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash \{type(t_s) = T_p\} \mathbf{wf}} \text{WFC-EQ}$$

which is equivalent to

$$\Gamma; \overline{X}_l^l; \Gamma_\delta \mid \overline{X}_i^i; \Gamma_p \vdash C \mathbf{wf} \quad \blacksquare$$

Lemma A.9 (Pattern Constraints Unification).

If $\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i | C) \Rightarrow \sigma$ and $\Gamma; \overline{X}_l^l | \overline{X}_i^i \vdash C$ **wf** and $\Gamma \vdash T$ **wf** and $\Sigma | \Gamma; \overline{X}_i^i \vdash^0 t : T$, then $\Sigma | \Gamma \vdash^0 \sigma(t) : T$

Proof.

Assuming premises of the Lemma

$$\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i | C) \Rightarrow \sigma \quad (1)$$

$$\Gamma; \overline{X}_l^l | \overline{X}_i^i \vdash C \text{ wf} \quad (2)$$

$$\Sigma | \Gamma; \overline{X}_i^i \vdash^0 t : T \quad (3)$$

$$\Gamma \vdash T \text{ wf} \quad (4)$$

We perform a proof by induction on the unification derivation of $\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i | C) \Rightarrow \sigma$.

Case U-EMPTY

Trivial as \overline{X}_i^i is empty and σ is the empty substitution.

Case U-EQ

In this case C is C_2 , $T_1 = T_1$ and therefore have

$$\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i | C_2) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i | C_2, T_1 = T_1) \Rightarrow \sigma} \text{U-EQ} \quad (5)$$

From the derivation of Eq. (2) we know

$$\frac{\dots \quad \Gamma; \overline{X}_l^l | \overline{X}_i^i \vdash C_2 \text{ wf}}{\Gamma; \overline{X}_l^l | \overline{X}_i^i \vdash C_2, T_1 = T_1 \text{ wf}} \quad (6)$$

Therefore by induction hypothesis using premises of Eqs. (5) and (6) and Eqs. (3) and (4) we get

$$\Sigma | \Gamma \vdash^0 \sigma(t) : T$$

Case U-PAT-VAR

In this case C is C_2 , $T_1 = X$ and $\overline{X} = \overline{X}_{i'}^{i'}$, $X; \overline{X}_{i''}^{i''}$ and $\sigma = [T_1/X] \circ \sigma_1$. Therefore have

$$\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_{i'}^{i'}; \overline{X}_{i''}^{i''} | C_2[T_1/X]) \Rightarrow \sigma_1 \quad \text{ftw}(T_1) \cap \overline{X}_l^l = \emptyset}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_{i'}^{i'}, X; \overline{X}_{i''}^{i''} | C_2, T_1 = X) \Rightarrow [T_1/X] \circ \sigma_1} \quad (7)$$

From the derivation of Eq. (2) we know

$$\frac{\Gamma; \overline{X}_l^l \vdash T_1 \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \overline{X}_{i'}^{i'}, X; \overline{X}_{i''}^{i''} \vdash X \mathbf{wf} \quad \Gamma; \overline{X}_l^l \mid \overline{X}_{i'}^{i'}, X; \overline{X}_{i''}^{i''} \vdash C_2 \mathbf{wf}}{\Gamma; \overline{X}_l^l \mid \overline{X}_{i'}^{i'}, X; \overline{X}_{i''}^{i''} \vdash C_2, T_1 = X \mathbf{wf}} \quad (8)$$

Using Lemma A.5 on $\Gamma; \overline{X}_l^l \mid \overline{X}_{i'}^{i'}, X; \overline{X}_{i''}^{i''} \vdash C_2 \mathbf{wf}$ we get

$$\Gamma; \overline{X}_l^l \mid \overline{X}_{i'}^{i'}; \overline{X}_{i''}^{i''}, X \vdash C_2 \mathbf{wf} \quad (9)$$

Using Lemma A.20 on Eq. (9) and $\Gamma; \overline{X}_l^l \vdash T_1 \mathbf{wf}$ we get

$$\Gamma; \overline{X}_l^l \mid \overline{X}_{i'}^{i'}; \overline{X}_{i''}^{i''} \vdash C_2[T_1/X] \mathbf{wf} \quad (10)$$

Using Lemma A.11 on $\Sigma \mid \Gamma; \overline{X}_{i'}^{i'}, X; \overline{X}_{i''}^{i''} \vdash^0 t : T$ we get

$$\Sigma \mid \Gamma, X; \overline{X}_{i'}^{i'}; \overline{X}_{i''}^{i''} \vdash^0 t : T \quad (11)$$

Therefore by induction hypothesis using premise of Eq. (7) and Eqs. (4), (10) and (11) we get

$$\Sigma \mid \Gamma, X \vdash^0 \sigma_1(t) : T \quad (12)$$

From $\Gamma; \overline{X}_l^l \vdash T_1 \mathbf{wf}$ and $ftv(T_1) \cap \overline{X}_l^l = \emptyset$ we can deduce that

$$\Gamma \vdash T_1 \mathbf{wf} \quad (13)$$

Using Lemma A.15 on Eqs. (12) and (13) we get

$$\Sigma \mid \Gamma \vdash^0 [T_1/X] \circ \sigma_1(t) : T[T_1/X] \quad (14)$$

Given that $\Gamma \vdash T \mathbf{wf}$ we know that $T[T_1/X]$ is equivalent to T . We also know that $\sigma = [T_1/X] \circ \sigma_1$. Therefore Eq. (14) is equivalent to

$$\Sigma \mid \Gamma \vdash^0 \sigma(t) : T$$

Case U-ABS

In this case C is $C_2, T_{s_1} \rightarrow T_{s_2} = T_{p_1} \rightarrow T_{p_2}$ and therefore have

$$\frac{\overline{X}_l^l \vdash \mathit{unify}(\overline{X}_i^i \mid C_2, T_{s_1} = T_{p_1}, T_{s_2} = T_{p_2}) \Rightarrow \sigma}{\overline{X}_l^l \vdash \mathit{unify}(\overline{X}_i^i \mid C_2, T_{s_1} \rightarrow T_{s_2} = T_{p_1} \rightarrow T_{p_2}) \Rightarrow \sigma} \text{U-ABS} \quad (15)$$

From the derivation of Eq. (2) we know

$$\frac{\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2 \mathbf{wf} \quad \frac{\Gamma; \overline{X}_l^l \vdash T_{s_1} \mathbf{wf} \quad \Gamma; \overline{X}_l^l \vdash T_{s_2} \mathbf{wf}}{\Gamma; \overline{X}_l^l \vdash T_{s_1} \rightarrow T_{s_2} \mathbf{wf}} \quad \frac{\Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash T_{p_1} \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash T_{p_2} \mathbf{wf}}{\Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash T_{p_1} \rightarrow T_{p_2} \mathbf{wf}}}{\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2, T_{s_1} \rightarrow T_{s_2} = T_{p_1} \rightarrow T_{p_2} \mathbf{wf}} \quad (16)$$

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

Using the premises from Eq. (16) we can derive

$$\frac{\frac{\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2 \mathbf{wf} \quad \Gamma; \overline{X}_l^l \vdash T_{s_1} \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash T_{p_1} \mathbf{wf}}{\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2, T_{s_1}=T_{p_1} \mathbf{wf}} \quad \Gamma; \overline{X}_l^l \vdash T_{s_2} \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash T_{p_2} \mathbf{wf}}{\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2, T_{s_1}=T_{p_1}, T_{s_2}=T_{p_2} \mathbf{wf}} \quad (17)$$

Therefore by induction hypothesis using premise of Eq. (15) and Eqs. (2), (3) and (17) we get

$$\Sigma \mid \Gamma \vdash^0 \sigma(t) : T$$

Case U-TABS

In this case C is $C_2, \forall X_1. T_1 = \forall X_2. T_2$ and therefore have

$$\frac{\overline{X}_l^l, X_1 \vdash \text{unify}(\overline{X}_i^i \mid C_2, T_1 = (T_2[X_1/X_2])) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C_2, \forall X_1. T_1 = \forall X_2. T_2) \Rightarrow \sigma} \text{U-TABS} \quad (18)$$

From the derivation of Eq. (2) we know

$$\frac{\frac{\Gamma; \overline{X}_l^l, X_1 \vdash T_1 \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \overline{X}_i^i, X_2 \vdash T_2 \mathbf{wf}}{\Gamma; \overline{X}_l^l \vdash \forall X_1. T_1 \mathbf{wf} \quad \Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash \forall X_2. T_2 \mathbf{wf}} \quad \Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2 \mathbf{wf}}{\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2, \forall X_1. T_1 = \forall X_2. T_2 \mathbf{wf}} \quad (19)$$

Premises of Eq. (19) can be weakened to

$$\Gamma; \overline{X}_l^l, X_1; \overline{X}_i^i, X_2 \vdash T_2 \mathbf{wf} \quad (20)$$

$$\Gamma; \overline{X}_l^l, X_1 \mid \overline{X}_i^i \vdash C_2 \mathbf{wf} \quad (21)$$

Now, we can substitute X_2 with X_1 in Eq. (20) to remove it from the environment

$$\Gamma; \overline{X}_l^l, X_1; \overline{X}_i^i \vdash T_2[X_1/X_2] \mathbf{wf} \quad (22)$$

Therefore from Eqs. (19), (21) and (22) we can derive

$$\frac{\Gamma; \overline{X}_l^l, X_1 \vdash T_1 \mathbf{wf} \quad \Gamma; \overline{X}_l^l, X_1; \overline{X}_i^i \vdash T_2[X_1/X_2] \mathbf{wf} \quad \Gamma; \overline{X}_l^l, X_1 \mid \overline{X}_i^i \vdash C_2 \mathbf{wf}}{\Gamma; \overline{X}_l^l, X_1 \mid \overline{X}_i^i \vdash C_2, T_1 = T_2[X_1/X_2] \mathbf{wf}} \quad (23)$$

Therefore by induction hypothesis using premise of Eq. (18) and Eqs. (2), (3) and (23) we get

$$\Sigma \mid \Gamma \vdash^0 \sigma(t) : T$$

Case U-QUOTE

In this case C is $C_2, \forall X_1. T_1 = \forall X_2. T_2$ and therefore have

$$\frac{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C_2, T_s = T_p) \Rightarrow \sigma}{\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C_2, [T_s] = [T_p]) \Rightarrow \sigma} \text{U-QUOTE} \quad (24)$$

From the derivation of Eq. (2) we know

$$\frac{\frac{\Gamma; \overline{X}_l^l \vdash T_s \text{ wf}}{\Gamma; \overline{X}_l^l \vdash [T_s] \text{ wf}} \quad \frac{\Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash T_p \text{ wf}}{\Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash [T_p] \text{ wf}} \quad \Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2 \text{ wf}}{\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2, [T_s] = [T_p] \text{ wf}} \quad (25)$$

Using the premises of Eq. (25) we can derive

$$\frac{\Gamma; \overline{X}_l^l \vdash T_s \text{ wf} \quad \Gamma; \overline{X}_l^l; \overline{X}_i^i \vdash T_p \text{ wf} \quad \Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2 \text{ wf}}{\Gamma; \overline{X}_l^l \mid \overline{X}_i^i \vdash C_2, T_s = T_p \text{ wf}} \quad (26)$$

Therefore by induction hypothesis using premise of Eq. (24) and Eqs. (3), (4) and (26) we get

$$\Sigma \mid \Gamma \vdash^0 \sigma(t) : T$$

■

Lemma A.10 (Unification Locality).

If $\overline{X}_l^l \vdash \text{unify}(\overline{X}_i^i \mid C) \Rightarrow \sigma$ and $X \in \overline{X}_l^l$, then $X \notin \text{image}(\sigma)$

Proof.

Proof by straight-forward induction over the unification derivation. The case U-PAT-VAR is the only one to add a new mapping and it explicitly states $\text{ftv}(T) \cap \overline{X}_l^l = \emptyset$.

■

Lemma A.11 (Well-Formed Type Weakeneing).

If $\Gamma_1; \Gamma_2, X; \Gamma_3 \vdash T \text{ wf}$ then $\Gamma_1, X; \Gamma_2; \Gamma_3 \vdash T \text{ wf}$

Proof.

Type well-formedness is preserved when environment is permuted *up to well-formedness* of the environment, since the well-formedness judgment only looks up types in the environment and is not sensitive to their order.

■

Lemma A.12 (Constraint Union).

If $\Gamma \mid \bar{X} \vdash C_1$ **wf** and $\Gamma \mid \bar{X} \vdash C_2$ **wf** then $\Gamma \mid \bar{X} \vdash C_1; C_2$ **wf**

Proof.

Trivial proof by induction on the derivation $\Gamma \mid \bar{X} \vdash C_1$ **wf**. ■

Lemma A.13 (Substitution).

$\forall i, j \in \mathbb{N}_0$, if $\Sigma \mid \Gamma \vdash^j t_1 : T_1$ and $\Sigma \mid \Gamma, x :^j T_1 \vdash^i t_2 : T_2$ then $\Sigma \mid \Gamma \vdash^i t_2[t_1/x] : T_2$

Proof.

From the premises we have that $\forall i, j \in \mathbb{N}_0$,

$$\Sigma \mid \Gamma \vdash^j t_1 : T_1 \tag{1}$$

$$\Sigma \mid \Gamma, x :^j T_1 \vdash^i t_2 : T_2 \tag{2}$$

Proof by induction on the typing derivation of $\Sigma \mid \Gamma, x :^j T_1 \vdash^i t_2 : T_2$.

Case T-CONST $\Sigma \mid \Gamma, x :^j T_1 \vdash^i \mathbf{c} : \mathbf{C}$

The substitution yields \mathbf{c} which is trivially typeable with T-CONST as $\Sigma \mid \Gamma \vdash^i \mathbf{c} : \mathbf{C}$.

Case T-VAR $\Sigma \mid \Gamma, x :^j T_1 \vdash^i y : T_2$

Sub-case $i = j$ If $x \neq y$, substitution will return y and therefore $\Sigma \mid \Gamma \vdash^j y : T_2$. If $x = y$, substitution will return t_1 , which implies $\Sigma \mid \Gamma \vdash^j t_1 : T_2$ where $T_1 = T_2$.

Sub-case $i \neq j$ Then $x \neq y$, otherwise premise Eq. (2) (i.e., $\Sigma \mid \Gamma, x :^j T_1 \vdash^i x : T_2$) would lead to a contradiction. Hence the substitution will return y and $\Sigma \mid \Gamma \vdash^j y : T_2$.

Case T-LINK $\Sigma \mid \Gamma, x :^j T_1 \vdash^i x : T$

Premise does not hold as x is defined in Σ and not in $\Gamma, x :^j T_1$.

Case T-ABS $\Sigma \mid \Gamma, x :^j T_1 \vdash^i \lambda y : T_3. t_3 : T_3 \rightarrow T_4$

From Eq. (2), we know that

$$\Sigma \mid \Gamma, x :^j T_1, y :^i T_3 \vdash^i t_3 : T_4 \tag{3}$$

Since $x \neq y$, we can use the following permutation of the environment of Eq. (3)

$$\Sigma \mid \Gamma, y :^i T_3, x :^j T_1 \vdash^i t_3 : T_4 \tag{4}$$

Eq. (1) can be weakened to

$$\Sigma \mid \Gamma, y :^i T_3 \vdash^j t_1 : T_1 \tag{5}$$

Using the induction hypothesis on Eqs. (4) and (5) we get

$$\Sigma \mid \Gamma, y :^i T_3 \vdash^i t_3[t_1/x] : T_4 \tag{6}$$

Therefore we can derive

$$\frac{\Sigma|\Gamma, y: {}^i T_3 \vdash^i t_3[t_1/x]: T_4}{\Sigma|\Gamma \vdash^i \lambda y: T_3.(t_3[t_1/x]): T_3 \rightarrow T_4} \text{T-ABS} \quad (7)$$

Finally, by definition of substitution Eq. (7) is equivalent to

$$\Sigma|\Gamma \vdash^i (\lambda y: T_3.t_3)[t_1/x]: T_3 \rightarrow T_4$$

Case T-APP $\Sigma|\Gamma, x: {}^j T_1 \vdash^i t_3 t_4: T$

Typing of $\Sigma|\Gamma \vdash^i t_3[t_1/x] t_4[t_1/x]: T_2$ follows directly from the induction hypothesis and T-APP.

Case T-TABS $\Sigma|\Gamma, x: {}^j T_1 \vdash^i \Lambda X.t: \forall X.T_3$

Typing of $\Sigma|\Gamma \vdash^i \Lambda X.t[t_1/x]: \forall X.T_3$ follows directly from the induction hypothesis and T-TABS.

Case T-TAPP $\Sigma|\Gamma, x: {}^j T_1 \vdash^i t T_3: T_4[T_3/X]$

Typing of $\Sigma|\Gamma \vdash^i t[t_1/x] T_3: T_4[T_3/X]$ follows directly from the induction hypothesis and T-TABS.

Case T-FIX $\Sigma|\Gamma, x: {}^j T_1 \vdash^i \mathbf{fix} t_3: T$

Typing of $\Sigma|\Gamma \vdash^i \mathbf{fix} t_3[t_1/x]: T_2$ follows directly from the induction hypothesis and T-FIX.

Case T-QUOTE $\Sigma|\Gamma, x: {}^j T_1 \vdash^i [t_3]: [T]$

Typing of $\Sigma|\Gamma \vdash^i [t_3[t_1/x]]: T_2$ follows directly from the induction hypothesis and T-QUOTE.

Case T-SPLICE $\Sigma|\Gamma, x: {}^j T_1 \vdash^i [t_3]: T$

Sub-case $i = 0$ Premise Eq. (2) does not hold as T-SPLICE expects an $i \geq 1$.

Sub-case $i > 0$

Typing $\Sigma|\Gamma \vdash^i [t_3[t_1/x]]: T_2$ follows directly from the induction hypothesis and T-SPLICE.

Case T-LIFT $\Sigma|\Gamma, x: {}^0 T_1 \vdash^i \mathbf{lift} t_3: [\mathbf{C}]$

Typing $\Sigma|\Gamma \vdash^i \mathbf{lift} t_3[t_1/x]: \mathbf{C}$ follows directly from the induction hypothesis and T-LIFT.

Case T-UNLIFT $\Sigma|\Gamma, x: {}^0 T_1 \vdash^i \mathbf{unlift} t_3 \mathbf{with} t_4 \mathbf{or} t_5: [\mathbf{C}]$

Typing judgment $\Sigma|\Gamma \vdash^i \mathbf{unlift} t_3[t_1/x] \mathbf{with} t_4[t_1/x] \mathbf{or} t_5[t_1/x]: \mathbf{C}$ follows directly from the induction hypothesis and T-UNLIFT.

Case T-MATCH with $\Sigma|\Gamma, x: {}^j T_1 \vdash^i t_s \mathbf{match} \bar{X} [t_p] \mathbf{then} t_t \mathbf{else} t_e: T_2$

We know that

$$\frac{\Sigma|\Gamma, x: {}^j T_1 \vdash^i t_s: T_3 \quad \dots \quad \Sigma|\Gamma, x: {}^j T_1; \bar{X}; \Gamma_t \vdash^i t_t: T_2 \quad \Sigma|\Gamma, x: {}^j T_1 \vdash^i t_e: T_2}{\Sigma|\Gamma, x: {}^j T_1 \vdash^i t_s \mathbf{match} \bar{X} [t_p] \mathbf{then} t_t \mathbf{else} t_e: T_2} \text{T-MATCH} \quad (8)$$

Using induction hypothesis on Eq. (1) and the first and last premise of Eq. (8) we get

$$\Sigma|\Gamma \vdash^i t_s[t_1/x]: T_3 \quad (9)$$

$$\Sigma|\Gamma \vdash^i t_e[t_1/x]: T_2 \quad (10)$$

We can weaken $\Sigma|\Gamma, x: {}^j T_1; \bar{X}; \Gamma_t \vdash^i t_t: T_2$ to

$$\Sigma|\Gamma; \bar{X}; \Gamma_t, x: {}^j T_1 \vdash^i t_t: T_2 \quad (11)$$

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

Using induction hypothesis on Eqs. (1) and (11) we get

$$\Sigma |\Gamma, \bar{X}; \Gamma_t \vdash^i t_t[t_1/x] : T_2 \quad (12)$$

Therefore we can derive

$$\frac{\Sigma |\Gamma \vdash^i t_s[t_1/x] : T_3 \quad \cdots \quad \Sigma |\Gamma, \bar{X}; \Gamma_t \vdash^i t_t[t_1/x] : T_2 \quad \Sigma |\Gamma \vdash^i t_e[t_1/x] : T_2}{\Sigma |\Gamma \vdash^i t_s[t_1/x] \text{ match } \bar{X} [t_p] \text{ then } t_t[t_1/x] \text{ else } t_e[t_1/x] : T_2} \quad (13)$$

Finally, definition of substitution Eq. (13) is equivalent to

$$\Sigma |\Gamma \vdash^i (t_s \text{ match } \bar{X} [t_p] \text{ then } t_t \text{ else } t_e)[t_1/x] : T_2$$

■

Lemma A.14 (Multi-Substitution).

$\forall i, j \in \mathbb{N}_0$, if $\overline{\Sigma |\Gamma \vdash^j t_k : T_k^k}$ and $\Sigma |\Gamma, x_k :^j T_k^k \vdash^i t : T$ then $\Sigma |\Gamma \vdash^i \overline{t[t_k/x_k]^k} : T$

Proof.

By well-formedness of Γ and the first premise we know that none of the terms t_k can have any x_k free. We can thus safely apply Substitution Lemma A.13 for each individual substitution $[t_k/x_k]$.

■

Lemma A.15 (Type Substitution Well-formedness).

If $\Gamma \vdash T_1 \mathbf{wf}$ and $\Gamma, X \vdash T_2 \mathbf{wf}$, then $\Gamma \vdash T_2[T_1/X] \mathbf{wf}$

Proof.

Assuming premises

$$\Gamma \vdash T_1 \mathbf{wf} \quad (1)$$

$$\Gamma, X \vdash T_2 \mathbf{wf} \quad (2)$$

Perform induction on the type well-formedness derivation of $\Gamma, X \vdash T_2 \mathbf{wf}$.

Case WFT-CONST $T_2 = \mathbf{C}$

Then by definition WFT-CONST we have $\Gamma \vdash \mathbf{C} \mathbf{wf}$. Therefore by definition of substitution we have $\Gamma \vdash \mathbf{C}[T_1/X] \mathbf{wf}$ which is equivalent to $\Gamma \vdash T_2[T_1/X] \mathbf{wf}$.

Case WFT-ABS $T_2 = T_3 \rightarrow T_4$

The typing judgment $\Gamma \vdash (T_3 \rightarrow T_4)[T_1/X] \mathbf{wf}$ follows directly from induction hypothesis, WFT-ABS and definition of substitution.

Case WFT-TABS $T_2 = \forall X_2. T_3$

From Eq. (2) know that

$$\frac{\Gamma, X, X_2 \vdash T_3 \mathbf{wf}}{\Gamma, X \vdash \forall X_2. T_3 \mathbf{wf}} \text{WFT-TABS} \quad (3)$$

Using Lemma A.11 on premise of Eq. (3) we get

$$\Gamma, X_2, X \vdash T_3 \mathbf{wf} \quad (4)$$

Using weakening Lemma A.6 on Eq. (1) we get

$$\Gamma, X_2 \vdash T_1 \mathbf{wf} \quad (5)$$

Using the induction hypothesis with Eqs. (4) and (5) we get

$$\Gamma, X_2 \vdash T_3[T_1/X] \mathbf{wf} \quad (6)$$

Therefore from Eq. (6) we can derive

$$\frac{\Gamma, X_2 \vdash T_3[T_1/X] \mathbf{wf}}{\Gamma \vdash \forall X_2. (T_3[T_1/X]) \mathbf{wf}} \text{WFT-TABS} \quad (7)$$

By definition of substitution Eq. (7) is equivalent to

$$\Gamma \vdash \forall X_2. T_3[T_1/X] \mathbf{wf}$$

Case WFT-QUOTED $T_2 = [T_3]$

The typing judgment $\Gamma \vdash [T_3][T_1/X] \mathbf{wf}$ follows directly from induction hypothesis, WFT-QUOTED and definition of substitution.

Case WFT-VAR $T_2 = X_2$

Sub-case $X = X_2$

In this case we have to show $\Gamma \vdash X_2[T_1/X_2] \mathbf{wf}$. By definition of substitution this is equivalent to $\Gamma \vdash T_1 \mathbf{wf}$ which hold by premise Eq. (1).

Sub-case $X \neq X_2$

From Eq. (2) know that

$$\frac{\frac{\vdash \Gamma \mathbf{wf} \quad X \notin \text{dom}(\Gamma)}{\vdash \Gamma, X \mathbf{wf}} \text{WFE-TVAR} \quad \frac{X_2 \in \Gamma}{X_2 \in \Gamma, X}}{\Gamma, X \vdash X_2 \mathbf{wf}} \text{WFT-VAR} \quad (8)$$

Therefore we can derive

$$\frac{\vdash \Gamma \mathbf{wf} \quad X_2 \in \Gamma}{\Gamma \vdash X_2 \mathbf{wf}} \text{WFT-VAR} \quad (9)$$

By definition of substitution Eq. (9) is equivalent to

$$\Gamma \vdash X_2[T_1/X] \mathbf{wf}$$

■

Lemma A.16 (Type Substitution).

$\forall i \in \mathbb{N}_0$, if $\Gamma_1 \vdash T_1 \mathbf{wf}$ and $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i t : T$ then $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i t[T_1/X] : T[T_1/X]$

Proof.

From the premises we have that $\forall i \in \mathbb{N}_0$,

$$\Gamma_1 \vdash T_1 \mathbf{wf} \quad (1)$$

$$\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i t : T \quad (2)$$

We perform a proof by induction on the typing derivation of $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i t : T$.

Case T-CONST $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i \mathbf{c} : \mathbf{C}$

Follows directly from T-CONST.

Case T-VAR $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i x : T$

From the premise of T-VAR we have that $x :^i T \in \Gamma_1, X; \Gamma_2$.

Sub-case $x :^i T \in \Gamma_1$

Then by well-formedness $X \notin \text{ftv}(T)$ and the substitution trivially holds.

Sub-case $x :^i T \in \Gamma_2$

In this cases we know that $x :^i T[T_1/X] \in \Gamma_2[T_1/X]$ and therefore by T-VAR we have that $\Sigma | \Gamma_1, (\Gamma_2[T_1/X]) \vdash^i x[T_1/X] : T[T_1/X]$.

Case T-LINK $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i x : T$

From the premise of T-LINK we have that $x : T$. Therefore by T-LINK $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i x : T$ holds. We also know that $\text{ftv}(T) = \emptyset$ because it is defined in Σ , this implies that $T[T_1/X] = T$. We additionally know from the definition of type substitution that $x[T_1/X] = x$. Therefore we can conclude $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i x[T_1/X] : T[T_1/X]$.

Case T-ABS $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i \lambda x : T_3. t_3 : T_3 \rightarrow T_4$

From T-ABS we have $\Sigma | \Gamma_1, X; \Gamma_2, x :^i T_3 \vdash^i t_3 : T_4$. Therefore by induction hypothesis we have that $\Sigma | \Gamma_1; (\Gamma_2, x :^i T_3[T_1/X]) \vdash^i t_3[T_1/X] : T_4[T_1/X]$. Which is equivalent to the typing judgment $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]), (x :^i T_3[T_1/X]) \vdash^i t_3[T_1/X] : T_4[T_1/X]$. Using this as premise of T-TABS we get $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i \lambda x : (T_3[T_1/X]). (t_3[T_1/X]) : (T_3[T_1/X]) \rightarrow (T_4[T_1/X])$. Therefore by definition of substitution we get $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i \lambda x : T_3. t_3[T_1/X] : T_3 \rightarrow T_4[T_1/X]$.

Case T-APP $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i t_1 t_2 : T$

Follows directly from induction hypothesis and T-APP.

Case T-TABS $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i \Lambda X_2. t_1 : \forall X_2. T_3$

From T-TABS we have $\Sigma | \Gamma_1, X; \Gamma_2, X_2 \vdash^i t_1 : T_3$ which is weakened to $\Sigma | \Gamma_1, X_2, X; \Gamma_2 \vdash^i t_1 : T_3$. Therefore by induction hypothesis we have that $\Sigma | \Gamma_1, X_2; (\Gamma_2[T_1/X]) \vdash^i t_1[T_1/X] : T_3[T_1/X]$. From T-TABS we have $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i \Lambda X_2. (t_1[T_1/X]) : \forall X_2. (T_3[T_1/X])$. Using the definition of type substitution we get $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i \Lambda X_2. t_1[T_1/X] : \forall X_2. T_3[T_1/X]$.

Case T-TAPP $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i t T_2 : T_3[T_2/X_2]$

Follows directly from induction hypothesis and T-APP.

Case T-FIX $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i \mathbf{fix} t_1 : T$

Follows directly from induction hypothesis and T-FIX.

Case T-QUOTE $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i [t_1] : [T]$

Follows directly from induction hypothesis and T-QUOTE.

Case T-SPLICE $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i [t_1] : T$

From T-SPLICE we know that an $i \geq 1$. The proof follows directly from induction hypothesis and T-SPLICE.

Case T-LIFT $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i \mathbf{lift} t_1 : [C]$

Follows directly from induction hypothesis and T-LIFT.

Case T-UNLIFT $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i \mathbf{unlift} t_1 \mathbf{with} t_2 \mathbf{or} t_3 : [C]$

Follows directly from induction hypothesis and T-UNLIFT.

Case T-MATCH $\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i t_s \mathbf{match} \bar{X} [t_p] \mathbf{then} t_t \mathbf{else} t_e : T$

From T-MATCH we have

$$\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i t_s : [T_p] \quad (3)$$

$$\Sigma | \Gamma_1, X; \Gamma_2; \bar{X} | \emptyset \vdash^{i+1} t_p : T_p \dashv \Gamma_t \quad (4)$$

$$\Sigma | \Gamma_1, X; \Gamma_2; \bar{X}; \Gamma_t \vdash^i t_t : T \quad (5)$$

$$\Sigma | \Gamma_1, X; \Gamma_2 \vdash^i t_e : T \quad (6)$$

Therefore by induction hypothesis applied on Eq. (1) and Eqs. (3), (5) and (6) we have that

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i t_s[T_1/X] : [T_p][T_1/X] \quad (7)$$

$$\Sigma | \Gamma_1; (\Gamma_2; \bar{X}; \Gamma_t[T_1/X]) \vdash^i t_t[T_1/X] : T[T_1/X] \quad (8)$$

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i t_e[T_1/X] : T[T_1/X] \quad (9)$$

By distributing the substitution in Eq. (8) we get

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]); \bar{X}; (\Gamma_t[T_1/X]) \vdash^i t_t[T_1/X] : T[T_1/X] \quad (10)$$

By Lemma A.19 using premise Eq. (4) we get

$$\Sigma | \Gamma_1; (\Gamma_2; \bar{X}[T_1/X]) | \emptyset \vdash^{i+1} t_p[T_1/X] : T_p[T_1/X] \dashv \Gamma_t[T_1/X] \quad (11)$$

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

Eq. (11) is equivalent to

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]); \overline{X} | \emptyset \vdash^{i+1} t_p[T_1/X] : T_p[T_1/X] \dashv \Gamma_t[T_1/X] \quad (12)$$

From T-MATCH with premises Eqs. (7), (9), (10) and (12) we have

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i t_s[T_1/X] \mathbf{match} \overline{X} [t_p[T_1/X]] \mathbf{then} t_t[T_1/X] \mathbf{else} t_e[T_1/X] : T[T_1/X] \quad (13)$$

By definition of type substitution Eq. (13) is equivalent to

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) \vdash^i (t_s \mathbf{match} \overline{X} [t_p] \mathbf{then} t_t \mathbf{else} t_e)[T_1/X] : T[T_1/X]$$

■

Lemma A.17 (Well-Formed Weak Type Substitution).

If $\Gamma \vdash T_1 \mathbf{wf}$ and $\Gamma \vdash T_2 \mathbf{wf}$, then $\Gamma \vdash T_2[T_1/X] \mathbf{wf}$

Proof.

Premise $\Gamma \vdash T_2 \mathbf{wf}$ can be weakened to $\Gamma, X \vdash T_2 \mathbf{wf}$. Therefore using Lemma A.15 with $\Gamma \vdash T_1 \mathbf{wf}$ and $\Gamma, X \vdash T_2 \mathbf{wf}$ we get $\Gamma \vdash T_2[T_1/X] \mathbf{wf}$.

■

Lemma A.18 (Type Multi-Substitution).

$\forall i \in \mathbb{N}_0$, if $\overline{X_j}^j \mathbf{wf}$ and $\Sigma | \Gamma, \overline{X_j}^j \vdash^i t : T$ then $\Sigma | \Gamma \vdash^i t[\overline{X_j}^j] : T[\overline{X_j}^j]$

Proof.

By well-formedness of Γ and the first premise we know that none of the terms T_j can have any X_j free. We can thus safely apply Substitution Lemma A.16 for each individual substitution $[T_j/X_j]$.

■

Lemma A.19 (Pattern Type Substitution).

$\forall i \in \mathbb{N}$, if $\Gamma_1 \vdash T_1 \mathbf{wf}$ and $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i t : T \dashv \Gamma_t$
then $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i t[T_1/X] : T[T_1/X] \dashv \Gamma_t[T_1/X]$

Proof.

Proof by induction on the typing derivations of the patterns $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i t : T \dashv \Gamma_t$.

Case T-PAT-CONST $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i \mathbf{c} : \mathbf{C} \dashv \emptyset$ Follows directly from T-PAT-CONST.

Case T-PAT-VAR $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i x : T \dashv \emptyset$

From the premises of T-PAT-VAR we have $x :^i T \in \Gamma_p$ and $\Gamma_1, X; \Gamma_2, \Gamma_p \vdash T \mathbf{wf}$. Therefore we also

know that $x : T[T_1/X] \in \Gamma_p[T_1/X]$ and $\Gamma_1, (\Gamma_2[T_1/X]); (\Gamma_p[T_1/X]) \vdash T[T_1/X]$ **wf**. By T-PAT-VAR we can derive $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i x : T[T_1/X] \dashv \emptyset$. By definition of type substitution we get $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i x : T[T_1/X] \dashv \emptyset[T_1/X]$.

Case T-PAT-LINK $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i x : T \dashv \emptyset$ From the premise of T-PAT-LINK we have that $x : T$. Therefore by T-PAT-LINK $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i x : T \dashv \emptyset$ holds. We also know that $fv(T) = \emptyset$ because it is defined in Σ , this implies that $T[T_1/X] = T$. We additionally know from the definition of type substitution that $x[T_1/X] = x$ and $\emptyset[T_1/X] = \emptyset$. Therefore we can conclude $\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i x[T_1/X] : T[T_1/X] \dashv \emptyset[T_1/X]$.

Case T-PAT-ABS $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i \lambda x : T_2. t_2 : T_2 \rightarrow T_3 \dashv \Gamma_t$
From the premise of T-PAT-ABS we have that $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p, x : T_2 \vdash^i t_2 : T_3 \dashv \Gamma_t$. Therefore by induction hypothesis we get

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p, x : T_2[T_1/X] \vdash^i t_2[T_1/X] : T_3[T_1/X] \dashv \Gamma_t[T_1/X]$$

Distributing the substitution in Γ_p we get

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | (\Gamma_p[T_1/X]), x : T_2[T_1/X] \vdash^i t_2[T_1/X] : T_3[T_1/X] \dashv \Gamma_t[T_1/X]$$

Therefore by T-PAT-ABS we can derive

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i \lambda x : (T_2[T_1/X]). (t_2[T_1/X]) : (T_2[T_1/X]) \rightarrow (T_3[T_1/X]) \dashv \Gamma_t[T_1/X]$$

Finally, by definition of type substitution we get

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i \lambda x : T_2. t_2[T_1/X] : T_2 \rightarrow T_3[T_1/X] \dashv \Gamma_t[T_1/X]$$

Case T-PAT-APP $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i t_1 t_2 : T \dashv \Gamma_t; \Gamma_{t_2}$

Follows directly from induction hypothesis, T-PAT-APP and definition of substitution.

Case T-PAT-TABS $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i \Lambda X_2. t_2 : \forall X_2. T_2 \dashv \Gamma_t$

From the premise of T-PAT-TABS we have that $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p, X_2 \vdash^i t_2 : T_2 \dashv \Gamma_t$. Therefore by induction hypothesis we get

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p, X_2[T_1/X] \vdash^i t_2[T_1/X] : T_2[T_1/X] \dashv \Gamma_t[T_1/X]$$

Which is equivalent to

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | (\Gamma_p[T_1/X]), X_2 \vdash^i t_2[T_1/X] : T_2[T_1/X] \dashv \Gamma_t[T_1/X]$$

Therefore by T-PAT-TABS we can derive

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i \Lambda X_2. (t_2[T_1/X]) : \forall X_2. (T_2[T_1/X]) \dashv \Gamma_t[T_1/X]$$

Finally, by definition of type substitution we get

$$\Sigma | \Gamma_1; (\Gamma_2[T_1/X]) | \Gamma_p[T_1/X] \vdash^i \Lambda X_2. t_2[T_1/X] : \forall X_2. T_2[T_1/X] \dashv \Gamma_t[T_1/X]$$

Appendix A. Soundness Proof of the Polymorphic Multi-Stage Macro Calculus

Case T-PAT-FIX $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i \mathbf{fix} \ t : T \dashv \Gamma_t$

Follows directly from induction hypothesis, T-PAT-FIX and definition of substitution.

Case T-PAT-BIND $\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i \llbracket x \rrbracket_T^{\overline{X_j^j x_k: T_k^k}} : T \dashv \emptyset, x : \forall X_j. \overline{[T_k] \rightarrow}^k [T]$

From the premise of T-PAT-BIND we have that

$$\frac{\overline{X_j \in \Gamma_p^j} \quad \overline{x_k : T_k \in \Gamma_p^k} \quad \Gamma_1, X; \Gamma_2 \vdash T \mathbf{wf}}{\Sigma | \Gamma_1, X; \Gamma_2 | \Gamma_p \vdash^i \llbracket x \rrbracket_T^{\overline{X_j^j x_k: T_k^k}} : T \dashv \emptyset, x : \forall X_j. \overline{[T_k] \rightarrow}^k [T]} \text{T-PAT-BIND} \quad (1)$$

Form $\Gamma_1, X; \Gamma_2 \vdash T \mathbf{wf}$ and $\Gamma_1 \vdash T_1 \mathbf{wf}$ we know that

$$\Gamma_1; (\Gamma_2 [T_1 / X]) \vdash T [T_1 / X] \mathbf{wf} \quad (2)$$

We can substitute X in $\overline{X_j \in \Gamma_p^j}$ and $\overline{x_k : T_k \in \Gamma_p^k}$ to get

$$\overline{X_j \in \Gamma_p [T_1 / X]^j} \quad (3)$$

$$\overline{x_k : T_k [T_1 / X] \in \Gamma_p [T_1 / X]^k} \quad (4)$$

Therefore by T-PAT-BIND we can derive

$$\frac{\overline{X_j \in \Gamma_p [T_1 / X]^j} \quad \overline{x_k : T_k [T_1 / X] \in \Gamma_p [T_1 / X]^k} \quad \Gamma_1; (\Gamma_2 [T_1 / X]) \vdash T [T_1 / X] \mathbf{wf}}{\Sigma | \Gamma_1; (\Gamma_2 [T_1 / X]) | (\Gamma_p [T_1 / X]) \vdash^i \llbracket x \rrbracket_{T [T_1 / X]}^{\overline{X_j^j x_k: T_k [T_1 / X]^k}} : T [T_1 / X] \dashv \emptyset, x : \forall X_j. \overline{[T_k [T_1 / X]] \rightarrow}^k [T [T_1 / X]]}$$

Finally, by definition of type substitution we get

$$\Sigma | \Gamma_1; (\Gamma_2 [T_1 / X]) | (\Gamma_p [T_1 / X]) \vdash^i \llbracket x \rrbracket_T^{\overline{X_j^j x_k: T_k^k}} [T_1 / X] : T [T_1 / X] \dashv (\emptyset, x : \forall X_j. \overline{[T_k] \rightarrow}^k [T]) [T_1 / X]$$

■

Lemma A.20 (Constraint Substitution).

If $\Gamma | \overline{X}, X \vdash C \mathbf{wf}$ and $\Gamma \vdash T \mathbf{wf}$, then $\Gamma | \overline{X} \vdash C [T / X] \mathbf{wf}$

Proof.

Assuming premises

$$\Gamma | \overline{X}, X \vdash C \mathbf{wf} \quad (1)$$

$$\Gamma \vdash T \mathbf{wf} \quad (2)$$

Perform induction on the constraint well-formedness derivation of $\Gamma | \overline{X}, X \vdash C \mathbf{wf}$.

Case WFC-EMPTY $\Gamma \mid \bar{X}, X \vdash \emptyset \mathbf{wf}$

Then we have that $C = \emptyset$ and therefore we also have $C[T/X] = \emptyset[T/X] = \emptyset$. Therefore we want to prove that $\Gamma \mid \bar{X} \vdash C[T/X] \mathbf{wf}$ which is equivalent to $\Gamma \mid \bar{X} \vdash \emptyset \mathbf{wf}$ which holds by WFC-EMPTY.

Case WFC-EQ $\Gamma \mid \bar{X}, X \vdash C_1, T_1 = T_2 \mathbf{wf}$

From WFC-EQ we have the following premises

$$\frac{\Gamma \vdash T_1 \mathbf{wf} \quad \Gamma; \bar{X}, X \vdash T_2 \mathbf{wf} \quad \Gamma \mid \bar{X}, X \vdash C_1 \mathbf{wf}}{\Gamma \mid \bar{X}, X \vdash C_1, T_1 = T_2 \mathbf{wf}} \text{WFC-EQ} \quad (3)$$

Using the induction hypothesis with $\Gamma \mid \bar{X}, X \vdash C_1 \mathbf{wf}$ and Eq. (2) we get

$$\Gamma \mid \bar{X} \vdash C_1[T/X] \mathbf{wf} \quad (4)$$

Using Lemma A.17 with $\Gamma \vdash T_1 \mathbf{wf}$ and Eq. (2) we get

$$\Gamma \vdash T_1[T/X] \mathbf{wf} \quad (5)$$

Using Lemma A.15 with $\Gamma; \bar{X}, X \vdash T_2 \mathbf{wf}$ and Eq. (2) we get

$$\Gamma; \bar{X} \vdash T_2[T/X] \mathbf{wf} \quad (6)$$

Therefore we can derive

$$\frac{\Gamma \vdash T_1[T/X] \mathbf{wf} \quad \Gamma; \bar{X} \vdash T_2[T/X] \mathbf{wf} \quad \Gamma \mid \bar{X} \vdash C_1[T/X] \mathbf{wf}}{\Gamma \mid \bar{X} \vdash (C_1[T/X]), (T_1[T/X]) = (T_2[T/X]) \mathbf{wf}} \text{WFC-EQ} \quad (7)$$

By definition of constraint substitution Eq. (7) is equivalent to

$$\Gamma \mid \bar{X} \vdash (C_1, T_1 = T_2)[T/X] \mathbf{wf}$$

■

Bibliography

- [1] B. J. Abate. Implement a documentation tool for Dotty using Tasty. Technical report, EPFL, 2019. M.Sc. Semester Project.
- [2] S. Alemanno. Implementing the f string interpolator using Dotty macros. Technical report, EPFL, 2019. URL <https://infoscience.epfl.ch/record/267528>. B.Sc. Semester Project.
- [3] A. Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 2010. ISBN 9780321635365.
- [4] N. Amin, T. Rompf, and M. Odersky. Foundations of path-dependent types. *SIGPLAN Not.*, 49(10):233–249, Oct. 2014. ISSN 0362-1340. doi: 10.1145/2714064.2660216. URL <https://doi.org/10.1145/2714064.2660216>.
- [5] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. *The Essence of Dependent Object Types*, pages 249–272. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30936-1. doi: 10.1007/978-3-319-30936-1_14. URL https://doi.org/10.1007/978-3-319-30936-1_14.
- [6] L. Andersen, S. Chang, and M. Felleisen. Super 8 languages for making movies (functional pearl). *Proc. ACM Program. Lang.*, 1(ICFP), Aug. 2017. doi: 10.1145/3110274. URL <https://doi.org/10.1145/3110274>.
- [7] Z. Ang. Macro annotations for Scala 3. Master's thesis, EPFL, 2022. URL <https://infoscience.epfl.ch/record/294615?&ln=en>.
- [8] H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993. ISBN 0198537611.
- [9] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of computer programming*, 16(2):151–195, 1991.
- [10] T. Bordenca. Dotty Decompiler. Technical report, EPFL, 2019. URL <https://infoscience.epfl.ch/record/292828>. M.Sc. Semester Project.
- [11] E. Burmako. Scala macros: Let our powers combine! on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala, SCALA '13*, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320641. doi: 10.1145/2489837.2489840. URL <https://doi.org/10.1145/2489837.2489840>.
- [12] E. Burmako. *Unification of Compile-Time and Runtime Metaprogramming in Scala*. PhD thesis, EPFL, 2017.

Bibliography

- [13] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proc. of the 2nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 57–76, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-20102-5.
- [14] J. Carette and O. Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *International Conference on Generative Programming and Component Engineering*, pages 256–274. Springer, 2005.
- [15] J. CARETTE, O. KISELYOV, and C.-C. SHAN. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009. doi: 10.1017/S0956796809007205.
- [16] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua. In Search of a Program Generator to Implement Generic Transformations for High-performance Computing. *Sci. Comput. Program.*, 62(1):25–46, Sept. 2006. ISSN 0167-6423.
- [17] K. Czarnecki, K. Østerbye, and M. Völter. Generative programming. In J. Hernández and A. Moreira, editors, *Object-Oriented Technology ECOOP 2002 Workshop Reader*, pages 15–29, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-36208-1.
- [18] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. *DSL Implementation in MetaOCaml, Template Haskell, and C++*, pages 51–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-25935-0. doi: 10.1007/978-3-540-25935-0_4. URL https://doi.org/10.1007/978-3-540-25935-0_4.
- [19] R. Davies. A temporal logic approach to binding-time analysis. *J. ACM*, 64(1), mar 2017. ISSN 0004-5411. doi: 10.1145/3011069. URL <https://doi.org/10.1145/3011069>.
- [20] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, may 2001. ISSN 0004-5411. doi: 10.1145/382780.382785. URL <https://doi.org/10.1145/382780.382785>.
- [21] L. EPFL. Scala 3 Compiler. <https://github.com/lampepfl/dotty>, 2021.
- [22] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, pages 303–326, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45337-6.
- [23] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 270–282, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595930272. doi: 10.1145/1111037.1111062. URL <https://doi.org/10.1145/1111037.1111062>.
- [24] M. Flatt. Composable and compilable macros: You want it when? In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 72–83, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134878. doi: 10.1145/581478.581486. URL <https://doi.org/10.1145/581478.581486>.
- [25] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17–es, May 2007. ISSN 0164-0925. doi: 10.1145/1232420.1232424. URL <https://doi.org/10.1145/1232420.1232424>.

- [26] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In *Proc. of the Sixth ACM SIGPLAN International Conference on Functional Programming, ICFP '01*, pages 74–85, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0.
- [27] P. V. Gorilskij. Implement string interpolator inline unapply. Technical report, EPFL, 2022. URL <https://infoscience.epfl.ch/record/294614?&ln=en>. M.Sc. Semester Project.
- [28] T. P. Hart. MACRO definitions for LISP. <https://dspace.mit.edu/handle/1721.1/6111>, October 1963.
- [29] J. Hunt. *Cake Pattern*, pages 115–119. Springer International Publishing, Cham, 2013. ISBN 978-3-319-02192-8. doi: 10.1007/978-3-319-02192-8_13. URL https://doi.org/10.1007/978-3-319-02192-8_13.
- [30] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. *SIGPLAN Not.*, 35(10):294–310, Oct. 2000. ISSN 0362-1340. doi: 10.1145/354222.353191. URL <https://doi.org/10.1145/354222.353191>.
- [31] J. Jang, S. G lineau, S. Monnier, and B. Pientka. M ebius: Metaprogramming using contextual types: The stage where System F can pattern match on itself. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi: 10.1145/3498700. URL <https://doi.org/10.1145/3498700>.
- [32] S. P. Jones. Template haskell, 14 years on. <https://slidetodoc.com/template-haskell-14-years-on-simon-peyton-jones/>, August 2016.
- [33] U. J rning and W. L. Scherlis. Compilers and staging transformations. In *Proc. of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '86*, pages 86–96, New York, NY, USA, 1986. ACM.
- [34] Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shifting the stage: Staging with delimited control. In *Proc. of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '09*, pages 111–120. ACM, 2009. ISBN 978-1-60558-327-3.
- [35] Y. Kameyama, O. Kiselyov, and C. chieh Shan. Combinators for impure yet hygienic code generation. *Science of Computer Programming*, 112:120–144, 2015. ISSN 0167-6423. doi: <https://doi.org/10.1016/j.scico.2015.08.007>. Selected and extended papers from Partial Evaluation and Program Manipulation 2014.
- [36] Y. Kammoun. XML String Interpolator for Dotty. Technical report, EPFL, 2019. URL <https://infoscience.epfl.ch/record/267527>. M.Sc. Semester Project.
- [37] A. J. Kennedy. Functional pearl pickler combinators. *J. Funct. Program.*, 14(6):727–739, nov 2004. ISSN 0956-7968. doi: 10.1017/S0956796804005209. URL <https://doi.org/10.1017/S0956796804005209>.
- [38] O. Kiselyov. The design and implementation of ber metaocaml. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*, pages 86–102, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07151-0.
- [39] O. Kiselyov. Reconciling abstraction with high performance: A MetaOCaml approach. *Foundations and Trends®in Programming Languages*, 5(1):1–101, 2018. ISSN 2325-1107.

Bibliography

- [40] O. Kiselyov and C.-c. Shan. The metaocaml files - status report and research proposal. In *ACM SIGPLAN Workshop on ML*, 2010.
- [41] O. Kiselyov and J. Yallop. let (rec) insertion without effects, lights or magic. *CoRR*, abs/2201.00495, 2022. URL <https://arxiv.org/abs/2201.00495>.
- [42] O. Kiselyov, A. Biboudis, N. Palladinos, and Y. Smaragdakis. Stream Fusion, to Completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL '17*, page 285–299, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009880. URL <https://doi.org/10.1145/3009837.3009880>.
- [43] B. Knuchel. Staged Tagless Interpreters in Dotty. Technical report, EPFL, 2019. URL <https://infoscience.epfl.ch/record/264990>. B.Sc. Semester Project.
- [44] S. Le Bail-Collet. Stackful Coroutines for Scala 3. Technical report, EPFL, 2020. B.Sc. Semester Project.
- [45] Y. Lilis and A. Savidis. A survey of metaprogramming languages. *ACM Comput. Surv.*, 52(6), Oct. 2019. ISSN 0360-0300. doi: 10.1145/3354584. URL <https://doi.org/10.1145/3354584>.
- [46] F. Liu and E. Burmako. Two approaches to portable macros. Technical report, EPFL, 2017. URL <https://infoscience.epfl.ch/record/231413w>.
- [47] F. McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, Queen's University of Belfast, 1970.
- [48] V. Mihaescu. A SQL to C compiler in Scala 3.0. Master's thesis, EPFL, 2020. M.Sc. Semester Project.
- [49] S. Monnier and Z. Shao. Inlining as staged computation. *J. Funct. Program.*, 13(3):647–676, may 2003. ISSN 0956-7968. doi: 10.1017/S0956796802004616. URL <https://doi.org/10.1017/S0956796802004616>.
- [50] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40531-3. doi: 10.1007/978-3-540-45070-2_10. URL https://dx.doi.org/10.1007/978-3-540-45070-2_10.
- [51] M. Odersky, E. Burmako, and D. Petrashko. A TASTY Alternative. Technical report, EPFL, 2016. URL <https://infoscience.epfl.ch/record/226194>.
- [52] M. Odersky, E. Burmako, and D. Petrashko. TASTY Reference Manual. Technical report, EPFL, 2016. URL <https://infoscience.epfl.ch/record/226193>.
- [53] M. Odersky, O. Blanvillain, F. Liu, A. Biboudis, H. Miller, and S. Stucki. Simplicity: Foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017. doi: 10.1145/3158130. URL <https://doi.org/10.1145/3158130>.
- [54] M. Odersky, A. Boruch-Gruszecki, J. I. Brachthäuser, E. Lee, and O. Lhoták. Safer Exceptions for Scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala, SCALA 2021*, page 1–11, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391139. doi: 10.1145/3486610.3486893. URL <https://doi.org/10.1145/3486610.3486893>.

- [55] D. L. Parnas. *On the Criteria to Be Used in Decomposing Systems into Modules*, pages 479–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-642-48354-7. doi: 10.1007/978-3-642-48354-7_20. URL https://doi.org/10.1007/978-3-642-48354-7_20.
- [56] L. Parreaux, A. Shaikhha, and C. E. Koch. Squid: Type-safe, hygienic, and reusable quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, pages 56–66, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355292. doi: 10.1145/3136000.3136005. URL <https://doi.org/10.1145/3136000.3136005>.
- [57] L. Parreaux, A. Shaikhha, and C. E. Koch. Quoted staged rewriting: A practical approach to library-defined optimizations. *SIGPLAN Not.*, 52(12):131–145, Oct. 2017. ISSN 0362-1340. doi: 10.1145/3170492.3136043. URL <https://doi.org/10.1145/3170492.3136043>.
- [58] L. Parreaux, A. Voizard, A. Shaikhha, and C. E. Koch. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158101. URL <https://doi.org/10.1145/3158101>.
- [59] D. Petrashko. *Design and implementation of an optimizing type-centric compiler for a high-level language*. PhD thesis, EPFL, Lausanne, 2017. URL <https://infoscience.epfl.ch/record/232671>.
- [60] F. Pfenning and C. Elliott. Higher-Order Abstract Syntax. In *Proc. of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208. ACM, 1988. ISBN 0-89791-269-1.
- [61] W. Radosław. A mechanized theory of quoted code patterns. Technical report, EPFL, 2020. URL <https://infoscience.epfl.ch/record/278147>. M.Sc. Semester Project.
- [62] T. Rompf. Reflections on lms: Exploring front-end alternatives. In *Proc. of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pages 41–50, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4648-1.
- [63] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proc. of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1.
- [64] D. Shabalin. Hygiene for Scala. Technical report, EPFL, 2014. URL <https://infoscience.epfl.ch/record/215109>.
- [65] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala. Technical report, EPFL, 2013. URL <https://infoscience.epfl.ch/record/185242>.
- [66] T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proc. of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6.
- [67] Y. Smaragdakis, A. Biboudis, and G. Fourtounis. Structured Program Generation Techniques. In J. Cunha, J. P. Fernandes, R. Lämmel, J. Saraiva, and V. Zaytsev, editors, *Grand Timely Topics in Software Engineering*, pages 154–178, Cham, 2017. Springer International Publishing. ISBN 978-3-319-60074-1.

Bibliography

- [68] L. Stadler, G. Duboscq, H. Mössenböck, T. Würthinger, and D. Simon. An experimental study of the influence of dynamic compiler optimizations on Scala performance. In *Proceedings of the 4th Workshop on Scala, SCALA '13*, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320641. doi: 10.1145/2489837.2489846. URL <https://doi.org/10.1145/2489837.2489846>.
- [69] B. Stroustrup. *The C++ programming language*. Addison-Wesley Professional, 2000. ISBN 9780201700732.
- [70] N. Stucki, A. Biboudis, and M. Odersky. 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*, pages 14–27, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360456. doi: 10.1145/3278122.3278139. URL <https://doi.org/10.1145/3278122.3278139>.
- [71] N. Stucki, P. G. Giarrusso, and M. Odersky. Truly Abstract Interfaces for Algebraic Data Types: The Extractor typing problem. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, Scala 2018*, pages 56–60, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358361. doi: 10.1145/3241653.3241658. URL <https://doi.org/10.1145/3241653.3241658>.
- [72] N. Stucki, A. Biboudis, S. Doeraene, and M. Odersky. Semantics-preserving Inlining for Metaprogramming. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala, SCALA 2020*, pages 14–24, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381772. doi: 10.1145/3426426.3428486. URL <https://doi.org/10.1145/3426426.3428486>.
- [73] N. Stucki, J. I. Brachthäuser, and M. Odersky. Multi-Stage Programming with Generative and Analytical Macros. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 110–122, 2021.
- [74] N. Stucki, J. I. Brachthäuser, and M. Odersky. Virtual ADTs for Portable Metaprogramming. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 36–44, 2021.
- [75] N. Stucki, J. I. Brachthäuser, and M. Odersky. Proof of Multi-Stage Programming with Generative and Analytical Macros. Technical report, EPFL, Sept. 2021. URL <https://infoscience.epfl.ch/record/288718?ln=en>.
- [76] N. A. Stucki, F. Liu, and A. Biboudis. Report on theory of quoted code patterns. Technical report, EPFL, 2020. URL <https://infoscience.epfl.ch/record/277946>.
- [77] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00*, pages 264–280, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 158113200X. doi: 10.1145/353171.353189. URL <https://doi.org/10.1145/353171.353189>.
- [78] D. Syme. The F# 3.0 Language Specification. <https://fsharp.org/specs/language-spec/3.0/FSharpSpec-3.0-final.pdf>, Sept. 2012.
- [79] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

-
- [80] W. Taha. *A Gentle Introduction to Multi-stage Programming*, pages 30–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-25935-0. doi: 10.1007/978-3-540-25935-0_3. URL https://doi.org/10.1007/978-3-540-25935-0_3.
- [81] W. Taha and M. F. Nielsen. Environment classifiers. In *In Proc. of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5.
- [82] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12): 203–217, dec 1997. ISSN 0362-1340. doi: 10.1145/258994.259019. URL <https://doi.org/10.1145/258994.259019>.
- [83] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 132–141, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993514. URL <https://doi.org/10.1145/1993498.1993514>.
- [84] S. Tobin-Hochstadt, V. St-Amour, E. Dobson, and A. Takikawa. The Typed Racket Guide - caveats and limitations, 2021. URL <https://docs.racket-lang.org/ts-guide/caveats.html>.
- [85] L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6), oct 2008. ISSN 0164-0925. doi: 10.1145/1391956.1391958. URL <https://doi.org/10.1145/1391956.1391958>.
- [86] T. L. Veldhuizen and D. Gannon. Active Libraries: Rethinking the roles of compilers and libraries, 1998. URL <https://arxiv.org/abs/math/9810022>.
- [87] N. Xie, M. Pickering, A. Löh, N. Wu, J. Yallop, and M. Wang. Staging with class: A specification for typed template haskell. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi: 10.1145/3498723. URL <https://doi.org/10.1145/3498723>.
- [88] J. Yallop and L. White. Modular macros. *OCaml Users and Developers Workshop*, 2015.

NICOLAS STUCKI

Language Designer
Computer Scientist
Software Engineer

 [linkedin.com/in/nicolas-stucki](https://www.linkedin.com/in/nicolas-stucki)
 github.com/nicolasstucki
 Lausanne, Switzerland
 Swiss



HIGHLIGHTS

Core contributor to Scala language design and implementation. Extensively worked on Scala 3 and Scalajs compilers and libraries. A decade and a half of JVM-languages experience. Contributed: 3'700+ commits, 500'000+ LOC, 2'000+ PRs to Scala open source projects.

INTERESTS

- Programming language design, implementation and theory
- Algorithms and data structures design and implementation
- Modern language features
- Maintainable and clean code

EDUCATION

| | | |
|-------------|--|---|
| 2016 – 2022 | PhD in Computer Science supervised by Martin Odersky | LAMP at EPFL, <i>Lausanne, CH</i> |
| 2012 – 2015 | Master in Computer Science with Specialization in Foundations of Software | EPFL, <i>Lausanne, CH</i> |
| 2006 – 2012 | Bachelor in Systems and Computer Engineering with Minor in Computational Mathematics | Universidad de los Andes, <i>Bogotá, CO</i> |

WORK EXPERIENCE

| | | |
|-----------------|---|---------------------|
| 9/2016 – 8/2022 | Doctoral Assistant (LAMP at EPFL) | <i>Lausanne, CH</i> |
| 4/2015 – 8/2016 | Software Developer (LAMP at EPFL) | <i>Lausanne, CH</i> |
| 2/2014 – 8/2014 | Software Developer (Akselos SA internship at EPFL Innovation Park) | <i>Lausanne, CH</i> |
| 6/2010 – 7/2010 | Software Developer (Panalpina internship) | <i>Basel, CH</i> |

TEACHING EXPERIENCE

| | | |
|-------------------|--|---|
| 2018 – 2021 | (Head) TA of Parallel Programming Class | EPFL, <i>Lausanne, CH</i> |
| 2014, 2017 – 2021 | (Head) TA of Functional Programming Class | EPFL, <i>Lausanne, CH</i> |
| 2010 – 2012 | TA of Algorithm Design Class | Universidad de los Andes, <i>Bogotá, CO</i> |
| 2008 | TA of OOP and Algorithms Class | Universidad de los Andes, <i>Bogotá, CO</i> |
| 2008 | TA of Differential Equations Class | Universidad de los Andes, <i>Bogotá, CO</i> |

SCIENTIFIC PUBLICATIONS

| | | |
|------|---|-----------------|
| 2021 | Multi-Stage Programming with Generative and Analytical Macros Stucki, Nicolas / Brachthäuser, Jonathan Immanuel / Odersky, Martin ★ Best paper award | GPCE |
| 2021 | Virtual ADTs for Portable Metaprogramming Stucki, Nicolas / Brachthäuser, Jonathan Immanuel / Odersky, Martin | MPLR |
| 2020 | Semantics-Preserving Inlining for Metaprogramming Stucki, Nicolas / Biboudis, Aggelos / Doeraene, Sébastien / Odersky, Martin | Scala Symposium |
| 2018 | Truly Abstract Interfaces for Algebraic Data Types Stucki, Nicolas / Giarrusso, Paolo Giosuè / Odersky, Martin | Scala Symposium |
| 2018 | A Practical Unification of Multi-stage Programming and Macros Stucki, Nicolas / Biboudis, Aggelos / Odersky, Martin | GPCE |
| 2016 | Semantics-Driven Interoperability between Scala.js and JavaScript Doeraene, Sébastien / Schlatter, Tobias / Stucki, Nicolas | Scala Symposium |
| 2015 | Improving the Interoperation between Generics Translations Ureche, Vlad / Stojanovic, Milos / Beguet, Romain Michel / Stucki, Nicolas / Odersky, Martin | PPPJ |
| 2015 | RRB Vector: A Practical General Purpose Immutable Sequence Stucki, Nicolas / Rompf, Tiark / Bagwell, Phil / Ureche, Vlad / Odersky, Martin | ICFP |
| 2013 | Bridging Islands of Specialized Code using Macros and Reified Types Stucki, Nicolas / Ureche, Vlad | Scala Workshop |

CONFERENCES

| | | |
|------|--|--------------|
| 2022 | Program Committee at GPCE | Auckland, NZ |
| 2021 | Selected Talk at ScalaCon Scala 3 Macros | Online |
| 2019 | Selected Talk at Scala Days Metaprogramming in Dotty | Lausanne, CH |

SUPERVISED STUDENT PROJECTS

| | | |
|------|--|--------------|
| 2022 | M.Sc. Thesis: Macro Annotations for Scala 3 Ang Zhendong | LAMP at EPFL |
| 2021 | M.Sc. Project: Customizable generation of wrapper code fo Scala Programs Andres Timothée | LAMP at EPFL |
| 2020 | M.Sc. Project: A Mechanized Theory of Quoted Code Patterns Wasko Radoslaw | LAMP at EPFL |
| 2019 | B.Sc. Project: Evaluating and Improving Performance of Generic Tuple Antoine Brunner | LAMP at EPFL |
| 2019 | M.Sc. Project: XML String Interpolator for Dotty Yassin Kammoun | LAMP at EPFL |
| 2019 | B.Sc. Project: Implementing the f string interpolator using Dotty macros Sara Alemanno | LAMP at EPFL |
| 2018 | M.Sc. Project: Implementation of Decompiler for the new Scala compiler Tobias Bordenca | LAMP at EPFL |
| 2018 | B.Sc. Project: Staged Tagless Interpreters in Dotty Benoit Louis Knuchel | LAMP at EPFL |
| 2017 | M.Sc. Project: Call-graph-based Optimizations in Scala Romain Beguet | LAMP at EPFL |

OPEN SOURCE PROJECTS

-  github.com/lampepfl/dotty
Scala 3 compiler and standard library. Contributed: 3'400+ commits, 335'000+ LOC, 1'800+ PRs.
-  github.com/scala/vscode-scala-syntax
Visual Studio Code plugin for Scala 2 and Scala 3 syntax highlighting. Also the reference syntax for GitHub syntax highlighting. Contributed: 100+ commits, 4'500+ LOC, 70+ PRs.
-  github.com/scala-js/scala-js
Scala.js compiler, library and tools. Contributed: 169 commits, 178'000+ LOC, 125 PRs.

LANGUAGES

English - proficient
Spanish - native
French - mother tongue
(Swiss) German - rudimentary

PROGRAMMING LANGUAGES

Scala - expert
Java - expert
Rust - intermediate
JavaScript - intermediate
Python - intermediate
C/C++/CUDA - intermediate
C# - rudimentary

OTHER SKILLS

CI/CD - GitHub Actions, Jenkins
Versioning - Git, Subversion
Markup - \LaTeX , HTML, XML, Mark-down, Graphviz Dot
Cloud Computing - Google Cloud