11-2008

Irrelevance, Polymorphism, and Erasure in Type Theory

Richard Nathan Mishra-Linger
Portland State University

DISSERTATION APPROVAL

The abstract and dissertation of Richard Nathan Mishra-Linger for the Doctor of Philosophy in Computer Science were presented on November 7, 2008, and accepted by the dissertation committee and the doctoral program.

COMMITTEE APPROVALS:

Timothy Sheard, Chair

James Hook

Mark Jones

Andrew Black

Sava Krstić

Malgorzata Chrzanowska-Jeske
Representative of the Office of Graduate Studies

DOCTORAL PROGRAM APPROVAL:

Wu-Chi Feng, Director
Computer Science Ph.D. Program

ABSTRACT

An abstract of the dissertation of Richard Nathan Mishra-Linger for the Doctor of Philosophy in Computer Science presented November 7, 2008.

Title: Irrelevance, Polymorphism, and Erasure in Type Theory

Dependent type theory is a proven technology for verified functional programming in which programs and their correctness proofs may be developed using the same rules in a single formal system. In practice, large portions of programs developed in this way have no computational relevance to the ultimate result of the program and should therefore be removed prior to program execution. In previous work on identifying and removing irrelevant portions of programs, computational irrelevance is usually treated as an intrinsic property of program expressions. We find that such an approach forces programmers to maintain two copies of commonly used datatypes: a computationally relevant one and a computationally irrelevant one.

We instead develop an extrinsic notion of computational irrelevance and find that it yields several benefits including (1) avoidance of the above mentioned code duplication problem; (2) an identification of computational irrelevance with a highly general form of parametric polymorphism; and (3) an elective (i.e., user-

directed) notion of proof irrelevance. We also develop a program analysis for identifying irrelevant expressions and show how previously studied types embodying computational irrelevance (including subset types and squash types) are expressible in the extension of type theory developed herein.

IRRELEVANCE, POLYMORPHISM, AND ERASURE IN TYPE THEORY

by

RICHARD NATHAN MISHRA-LINGER

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Portland State University
2008

DEDICATION

This work is dedicated to Jesus Christ, in whom are hidden all the treasures of wisdom and knowledge (Colossians 2:3).

ACKNOWLEDGEMENTS

I would like to thank my advisor, Tim Sheard for introducing me to many interesting problems and ideas. His insight guided me, and his enthusiasm motivated me to pursue my own ideas. May this dissertation give him pride.

I thank Mark Jones, Sava Krstić, and John Launchbury for the quality of their instruction both inside and outside the classroom. They are excellent teachers and I enjoyed learning from them.

Thanks to Emir Pašalić, Iavor Diatchki, and Tom Harke for many discussions in which they taught me so much and listened to my half-baked ideas and new discoveries.

I thank Lyle Kopnicky, Dan Brown, Tom Harke, and Amber Telfer for their friendship on and off campus. You made my time here enjoyable.

Thanks to my spiritual family at West Valley Community Church, through whose prayers and friendship God has given me strength.

Thanks to my parents and sister for their love and support. They taught me the intrinsic value of education by word and example.

I am especially thankful to and for my wife Taniya, whose love delights and comforts me and whose hard work inspires me. Meeting and marrying you was by far the best "result" I obtained in my graduate work. I love you.

Finally, I thank God for giving me the opportunity to study such interesting things with such interesting people and the ability to make significant contributions in my field.

CONTENTS

Acknowledgements . . . . . **ii**

List of Tables . . . . . **vii**

List of Figures . . . . . **viii**

1 Introduction **1**
- 1.1 Type Systems and Expressiveness 1
- 1.2 Thesis 3
- 1.3 Contributions 4
- 1.4 Computational Irrelevance 5
  - 1.4.1 Intrinsic View Leads to Code Duplication 5
  - 1.4.2 Extrinsic View 8
- 1.5 Methodology and Overview 13

2 Review of Pure Type Systems **15**
- 2.1 History 15
- 2.2 The λ-calculus 18
- 2.3 Church's Simply-Typed λ-calculus 24
- 2.4 The Girard/Reynolds Polymorphic λ-calculus 28
  - 2.4.1 Impredicative Encodings 31
  - 2.4.2 Relational Parametricity 32
- 2.5 Girard's System F^{ω} 34
- 2.6 The Edinburgh Logical Framework 38
- 2.7 Coquand and Huet's Calculus of Constructions 42
- 2.8 Barendregt's λ-cube 45
- 2.9 Pure Type Systems 49
  - 2.9.1 Specifications 49
  - 2.9.2 Syntax 52
  - 2.9.3 Typing Rules 53

2.10 Pure Type System Examples . . . 55
2.10.1 Systems in the λ-cube . . . 55
2.10.2 Hindley-Milner Polymorphism . . . 56
2.10.3 Extended Calculus of Constructions . . . 58

3 Erasure Semantics . . . 59
3.1 Erasure Pure Type Systems . . . 59
3.1.1 Syntax . . . 60
3.1.2 Type System . . . 61
3.1.3 Semantics . . . 68
3.1.4 Meta-theory . . . 69
3.2 Implicit Pure Type Systems . . . 72
3.3 The Erasure Translation . . . 75
3.3.1 Meta-theory . . . 76
3.3.2 Erasure Semantics . . . 81
3.4 Implementation . . . 82
3.5 Conclusions . . . 86

4 Erasability Analysis . . . 87
4.1 An Example . . . 87
4.2 Constraint Generation . . . 89
4.2.1 Syntax of Annotations and Constraints in EPTS$^{\mathcal{C}}$ . . . 89
4.2.2 Constraint-Generating Typing Rules . . . 92
4.2.3 Proof of Correctness . . . 93
4.2.4 Logical Structure of Generated Constraints . . . 98
4.2.5 Implementation . . . 99
4.3 Constraint Solving . . . 101
4.3.1 Terminology . . . 101
4.3.2 The TOP-SAT Problem . . . 101
4.3.3 An Algorithm for our Special Case . . . 102
4.3.4 Partial Annotation . . . 105
4.3.5 Implementation . . . 105
4.4 Conclusions . . . 107

5 Inductive Types . . . 109
5.1 Introduction . . . 110
5.1.1 Parameterized Inductive Types . . . 114

5.1.2 Indexed Inductive Types . . . 115
5.1.3 Parameterized Indexed Inductive Types . . . 116
5.2 Opportunities for Erasure . . . 118
5.2.1 Eliminator Argument Erasure . . . 119
5.2.2 Constructor Argument Erasure . . . 124
5.2.3 Eliminator Target Erasure . . . 128
5.3 A Paradigmatic Example: Various Sum Types . . . 136
5.3.1 Strong Sums . . . 137
5.3.2 Weak Sums . . . 138
5.3.3 Subset Types . . . 140
5.4 Squash Types . . . 142
5.4.1 Relating $\bigcirc$ and $\Rightarrow$ . . . 143
5.4.2 Correspondence with Nuprl's Squash Type . . . 145
5.4.3 Usage of the Squash Type . . . 147
5.4.4 Laws Concerning $\bigcirc$ . . . 148
5.5 Comparison with Nuprl and Coq . . . 150
5.6 Conclusions . . . 153

6 Proof Irrelevance . . . 156
6.1 Extra Expressiveness of CONV$^\bullet$ . . . 157
6.1.1 Elective Proof Irrelevance . . . 157
6.1.2 Uniformity Principle . . . 159
6.1.3 Streicher's K "Axiom" is Provable . . . 159
6.2 Non-computational Axioms . . . 163
6.2.1 Axioms for Classical Reasoning . . . 165
6.2.2 The $\bigcirc$-flattening Axiom . . . 166
6.3 EPTS$^\bullet$ . . . 168
6.3.1 Meta-theory of Erasure . . . 168
6.3.2 Equivalence with IPTS . . . 169
6.4 Erasability Analysis . . . 170

7 Related Work . . . 173
7.1 Useless Variable Elimination . . . 173
7.2 Subset and Squash Types . . . 180
7.3 Program Extraction . . . 181
7.3.1 Realizability Interpretations . . . 182

7.3.2 The Theory of Specifications . . . 186
7.3.3 Pruning Methods . . . 188
7.4 Miscellaneous . . . 192

8 Conclusion . . . 194
8.1 Summary . . . 194
8.2 Significance . . . 200
8.3 Limitations and Future Work . . . 201

References . . . 203

A Proofs . . . 214
A.1 Meta-theory of EPTS . . . 214
A.2 Meta-theory of Erasure . . . 242
A.3 Implementation of Erasure Contexts . . . 270
A.4 Meta-theory of EPTS$^{\mathcal{C}}$ . . . 276
A.5 Meta-theory of EPTS$^{\bullet}$ . . . 302

LIST OF TABLES

5.1 Concrete syntax for erasure annotations 119

LIST OF FIGURES

1.1 Computationally relevant and irrelevant naturals in Coq 9
1.2 Illustration of the relativity of computational relevance. 10
1.3 Graph induced by the "may-flow-to" relation of a simple program . 12
1.4 Two-phase approach to erasure semantics 13

2.1 The λ-calculus 19
2.2 Simply typed λ-calculus 25
2.3 System F 29
2.4 System F^{ω} 35
2.5 The Edinburgh Logical Framework 39
2.6 The Calculus of Constructions (1/2) 43
2.7 The Calculus of Constructions (2/2) 44
2.8 The λ-cube 47
2.9 Typing rules for PTS 53

3.1 Typing rules for EPTS 62
3.2 Simple typing derivation in EPTS 65
3.3 Typing rules for IPTS 73
3.4 Meta-theory of EPTS 77
3.5 Clever Implementation of Typing Contexts 83

4.1 Sketch of erasability analysis and erasure 88
4.2 How a typical constraint arises. 93
4.3 Constraint generating typing rules for $\text{EPTS}^{\mathcal{C}}$ 94
4.4 Constraint generating conversion rules for $\text{EPTS}^{\mathcal{C}}$ 95

5.1 Example showing shortcomings of token type target erasure 135
5.2 An isomorphism between $A \Rightarrow B$ and $\bigcirc A \to B$ 144
5.3 An isomorphism between two squash type representations 146

5.4 Division of computational and non-computational entities into two type universes 151
5.5 Division of computational and non-computational entities inside a single type universe using squash types 152

6.1 Lemmas in Altenkirch's proof that a behavioral uniformity principle implies uniqueness of identity proofs 161
6.2 Proof of Uniqueness of (Reflexive) Identity Proofs 162
6.3 Two consequences of Streicher's K eliminator and Uniqueness of Reflexive Identity Proofs 164
6.4 Relationships between four PTS variants 168

7.1 A Coq term and its corresponding extracted looping program . . . 190

Chapter 1

INTRODUCTION

1.1 TYPE SYSTEMS AND EXPRESSIVENESS

In the past half century, type theory has emerged as a unifying principle in programming language design. In the 1960s, a deep connection between constructive logic and functional programming was discovered, the so-called *propositions-as-types* correspondence, enabling significant subsequent cross-fertilization between these two fields [30, 89, 42]. At the heart of this correspondence lies the promise of a unified language for both programming and proving programs correct. We briefly outline the history of type theory in Section 2.1.

In a programming language, types categorize the values in a program. A type system provides a structured way to assign meaning to program values. By the propositions-as-types correspondence, we may also think of a type as a logical formula or proposition stating some correctness property of program values. By this view, a static type system for a programming language corresponds to an internal logic of program correctness. Such a logic provides the formal specification for a *type checker* – a program analyzer which automatically identifies certain program errors before a program is even run, at which point they are relatively inexpensive to fix. For this reason, type systems are an important approach to taming the error-prone process of software development.

Some type systems are stronger than others – their internal logic of program

correctness is more expressive in terms of what program properties it can state. The more program properties a programmer can state in the type system, the more bugs a type checker can catch. For this reason, programming languages have evolved ever more expressive type systems. This drive towards increased expressiveness inevitably leads to *dependent types*, a proven technology for verifying strong correctness properties of real programs [69, 52, 11, 50]. For this reason, researchers have long sought practical ways to include dependent types in programming languages. As regards expressiveness, the internal logic of a typical dependently typed language is strong enough to formalize all of mathematics.

However, as the strength of a type system increases, so does the amount of extra bookkeeping information that a programmer must insert into programs to ensure that type-checking remains decidable. In the limit, programs must contain *proofs* of otherwise undecidable program properties. As the strength of the internal logic increases, a type checker thus transitions from an automatic theorem prover (for a relatively weak logic) to a proof checker (for a relatively strong logic).

For this reason, heavy use of dependent types often involves embedding proofs of program properties into the program text itself. Often, such proofs play no essential part in the execution of the program. They are necessary only for "convincing" the type-checker that various program properties hold. We would like to *erase* these computationally irrelevant portions of our program prior to run-time, so as to avoid the needless cost of evaluating them and storing their values. This notion of an erasure phase prior to run-time is called an *erasure semantics*. Because embedded proofs can be quite large, an erasure semantics is critical for practical implementation of a dependently typed programming language.

Proofs are not the only erasable parts of a program. In general, all dead code is erasable. Our erasure semantics will in some sense approximate dead code elimination. Another way of thinking of erasure is in terms of polymorphism. Any time a program exhibits *parametric polymorphism*, whether over proofs, types,

numbers, or any other kind of value, there are portions of the program that can be erased. In fact, parametric polymorphism can be understood entirely in terms of erasure. This claim is a major component of my thesis.

1.2 THESIS

The thesis of this dissertation is:

> An extrinsic view of computational irrelevance results in (1) a flexible erasure semantics for dependently typed languages; (2) a generic form of parametric polymorphism; and (3) an elective notion of proof irrelevance.

The remainder of this dissertation is spent explaining and defending this statement. For now, we give a basic overview of terminology in the thesis.

Intrinsic vs. extrinsic views of computational irrelevance. A property of an entity is *intrinsic* if it is an essential or inherent to that entity. Examples include mass and sex. In contrast, an *extrinsic* property of an entity depends on the relationship between that entity and its context or environment. Examples include weight (dependent on position with respect to other masses) and marital status (dependent on social context).

In Section 1.4, we discuss common approaches to identifying computationally irrelevant portions of a program – those parts which do not affect the execution of the program one way or the other. We identify an intrinsic view of computational irrelevance in all these approaches that leads to a problem of forced code duplication. In contrast, we advocate an extrinsic view of computational irrelevance.

Flexible erasure semantics. By flexible, we mean that the same program expression may be erased if it appears in some program contexts but not in others.

This is essentially the extrinsic view of erasure. The consequence of this view is that we may write subprograms without any concern about their erasure behavior and then add erasure annotations later. In fact, erasure annotations may be introduced completely automatically.

Generic parametric polymorphism. The polymorphic λ-calculus (System F) is the paradigmatic language exhibiting parametric polymorphism. In this language, we may parameterize a program by the types at which it operates. Such type-parameterized programs behave uniformly at all types to which they are instantiated. In our development, we generalize this notion of parametric polymorphism over types to a generic notion of parametric polymorphism over any kind of program entity (types, numbers, proofs, etc.).

Elective proof irrelevance. Proof irrelevance is a principle whereby two proofs of the same proposition are considered equal. When proofs are embedded in datatypes, this principle is often needed to reflect the user's intentions about what objects are considered equal. Rather than decide once and for all that all proofs are considered equal, an elective notion of proof irrelevance allows the programmer to determine where to use the principle of proof irrelevance.

1.3 CONTRIBUTIONS

The contributions of this dissertation are:

1. Identification of the intrinsic view of erasure as the root of the code duplication problem exhibited by all previous attempts to combine dependent types and erasure semantics (Section 1.4.1)

2. An operationally motivated extrinsic notion of erasure that solves the duplication problem (Section 1.4.2)

3. Formal development of an erasure semantics for Pure Type Systems. (Chapter 3)

4. Illumination of the relationship between erasure and parametric polymorphism as exhibited in Miquel's Implicit Pure Type Systems (Chapter 3)

5. Formal development of a program analysis that determines all erasable portions of a program. (Chapter 4)

6. Expression of previously known type-based information hiding methods in terms of inductive types with erasure annotations (Chapter 5)

7. Development of a form of elective proof irrelevance in terms of erasure. (Chapter 6)

1.4 COMPUTATIONAL IRRELEVANCE

Existing languages combining dependent types and erasure semantics have the common shortcoming of *forced code duplication*. In this section I explain this problem and diagnose its cause: an intrinsic view of computational irrelevance.

1.4.1 Intrinsic View of Erasure Leads to Code Duplication

Current languages combining dependent types and erasure semantics may be divided into two categories: erasure first and dependent types first.

Erasure first

Languages in this category start with a commitment to erasure semantics in the form of a syntactic phase distinction whereby types and program values may not depend on each other computationally. Singleton types are then used to simulate dependently typed programming. Examples of this approach include Dependent

ML [97], Ωmega [83, 82], Applied Type Systems [17], and Haskell with generalized algebraic datatypes (GADTs) [72].

Singleton types are type constructors $T : I \to \mathsf{Type}$ for which each type index $i : I$ uniquely determines the one value of type $T(i)$. For example, the declarations

$$\begin{array}{lllll} \underline{\mathsf{datakind}} & \mathsf{nat}\uparrow & : & \mathsf{kind} \\ \quad \underline{\mathsf{where}} & \mathsf{zero}\uparrow & : & \mathsf{nat}\uparrow \\ & \mathsf{succ}\uparrow & : & \mathsf{nat}\uparrow \to \mathsf{nat}\uparrow \end{array} \qquad \begin{array}{lllll} \underline{\mathsf{datatype}} & \mathsf{nat!} & : & \mathsf{nat}\uparrow \to \mathsf{Type} \\ \quad \underline{\mathsf{where}} & \mathsf{zero!} & : & \mathsf{nat!\ zero}\uparrow \\ & \mathsf{succ!} & : & \mathsf{nat!}\ n \to \mathsf{nat!}\ (\mathsf{succ}\uparrow\ n) \end{array}$$

introduce a singleton type family for the naturals. The datakind declaration defines a *copy* of the natural numbers at the type-level. The singleton type nat! then connects the type-level version nat↑ to the level of run-time expressions. To see the one-to-one correspondence, consider the following terms and their types.

$$\begin{array}{rcl} \mathsf{zero!} & : & \mathsf{nat!\ zero}\uparrow \\ \mathsf{succ!\ zero!} & : & \mathsf{nat!\ (succ}\uparrow\ \mathsf{zero}\uparrow) \\ \mathsf{succ!\ (succ!\ zero!)} & : & \mathsf{nat!\ (succ}\uparrow\ (\mathsf{succ}\uparrow\ \mathsf{zero}\uparrow)) \\ \mathsf{succ!\ (succ!\ (succ!\ zero!))} & : & \mathsf{nat!\ (succ}\uparrow\ (\mathsf{succ}\uparrow\ (\mathsf{succ}\uparrow\ \mathsf{zero}\uparrow))) \end{array}$$

Notice the singleton property: The only term of type nat! n is a term n' with exactly the same structure as n. However n' and n are not the same thing. One (n) is a type expression (a compile-time entity) and the other (n') is a term expression (a run-time entity).

A singleton type acts as a proxy between run-time and compile-time notions of the same datatype: natural numbers in this case. Whenever a program does case analysis on the value of a singleton type, the type-checker benefits from the same case analysis at the type-level. In this way, dependence of types on values is

simulated. The following program exhibits this behavior.

$$\begin{array}{rlcl} \underline{\mathsf{datatype}} & \mathsf{boollist} & : & \mathsf{nat}{\uparrow} \to \mathsf{Type} \\ \underline{\mathsf{where}} & \mathsf{nil} & : & \mathsf{boollist\ zero}{\uparrow} \\ & \mathsf{cons} & : & \mathsf{bool} \to \mathsf{boollist}\ n \to \mathsf{boollist}\ (\mathsf{succ}{\uparrow}\ n) \end{array}$$

$$\begin{aligned} &\mathit{fill} : \forall n : \mathsf{nat}{\uparrow}.\ \mathsf{bool} \to \mathsf{nat!}\ n \to \mathsf{boollist}\ n \\ &\mathit{fill}\ x\ (\mathsf{zero!}) = \mathsf{nil} \\ &\mathit{fill}\ x\ (\mathsf{succ!}\ m) = \mathsf{cons}\ x\ (\mathit{fill}\ x\ m) \end{aligned}$$

The type $\mathsf{boollist}\ n$ is the type of lists of length n containing natural numbers for elements. The function call $\mathit{fill}\ x\ n$ returns a list of length n in which every element is a copy of x. In each equation defining *fill*, the type-checker obtains more specific information about the type variable n due to the pattern matching on the argument of singleton type $\mathsf{nat!}\ n$.

Dependent types first

Languages in this category start with full dependent types. An erasure phase then strips out parts of the program that are irrelevant to its run-time execution. Examples of this approach include Cayenne [1], Coq [24], and Epigram [60, 13].

In Cayenne and Coq, the erasability of a subterm depends on its type. All types, subterms of type Type, are erased in Cayenne and Epigram[1]. Coq's program extraction mechanism supports erasure of *proofs* as well as types. A proof is distinguished by having a proposition as its type, and propositions are distinguished as terms of type Prop. In contrast to the universe Prop, Coq has another universe Set that is the type of the types of all non-erasable program terms.

This distinction between proofs and programs is concisely captured by the

[1]Some work on representations of inductive types in Epigram [14] notes that values of type families need not store certain indices that, regardless of their type, are uniquely determined by the value's data constructor.

statements

$$\begin{array}{lcll} \langle proof \rangle & : & \langle proposition \rangle & : \ \mathsf{Prop} \\ \langle program \rangle & : & \langle program\ type \rangle & : \ \mathsf{Set} \end{array}$$

where the relation $A : B$ means "A has type B". This triplet pattern ending in a special constant (here Prop or Set) is one we will see again when we review Pure Type Systems in Chapter 2.

Code duplication

Because languages in both categories treat erasability as an intrinsic property of an expression, usually determined by its type, users of these languages are sometimes forced to duplicate definitions of datatypes and functions over them in order to achieve a desired erasure behavior. In the erasure-first approach, programming with singleton types requires duplication of datatype definitions at the "type" and "kind" levels of the type hierarchy, as well as duplication of functions that operate on them. In the dependent-types-first approach, duplication of datatypes is also required if we want values of a particular type to be erased in one part of a program but not in another.

For example, it is likely that one would want natural numbers to be erased in some parts of a Coq program, but preserved in other parts. To get this behavior, one must define copies in Prop and Set of the same type. Any needed functions over naturals, such as addition, would need to be duplicated as well. See Figure 1.1 for these definitions.

1.4.2 Extrinsic View of Erasure

The computationally irrelevant parts of a program[2] are those that may be erased in an erasure semantics. Our investigation of erasure semantics is grounded in a

[2]For our purposes here, a program is a term in a typed λ-calculus. We will review λ-calculus in Chapter 2.

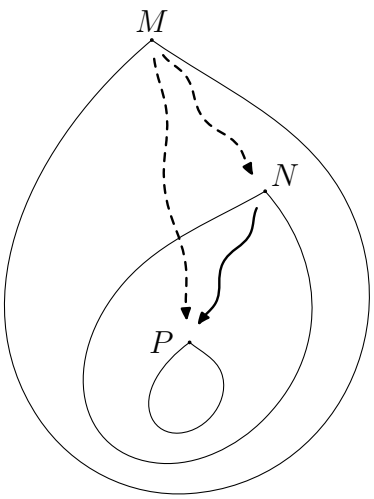
```
Inductive SNat : Set :=
    SZero : SNat
  | SSucc : SNat -> SNat.

Fixpoint splus (a b : SNat) {struct a} : SNat :=
  match a with
  | SZero => b
  | SSucc a' => SSucc (splus a' b)
  end.

Inductive PNat : Prop :=
    PZero : PNat
  | PSucc : PNat -> PNat.

Fixpoint pplus (a b : PNat) {struct a} : PNat :=
  match a with
  | PZero => b
  | PSucc a' => PSucc (pplus a' b)
  end.
```

Figure 1.1: Computationally relevant and irrelevant naturals in Coq



Consider terms $M > N > P$ (where $\leq$ is the subterm ordering) such that N depends computationally on P but M does not depend computationally on N. Then M does not depend computationally on P since it may do so only via N.

Therefore P is both computationally relevant (with respect to N) and irrelevant (with respect to M).

Figure 1.2: Illustration of the relativity of computational relevance.

simple observation: computational irrelevance of a program expression P is not a property of P itself but rather a property of the context in which we find it. Irrelevance of P is determined not by what it *is*, but by how it is *used*. In other words, computational irrelevance is an *extrinsic* rather than an *intrinsic* property. Figure 1.2 illustrates this fact in the abstract. We give several examples demonstrating this principle in the remainder of this section.

Type annotations

The domain annotation A in the β rule,

$$(\lambda x{:}A.\, M)\ N \rightarrow_\beta M[N/x]$$

is simply discarded during reduction. For this reason, we may safely erase the domain annotations of all λ-abstractions in a program without changing their computational behavior. In this case, the context in which A appears determines its erasability.

Dummy λ-binders

The erasure of domain annotations may cause some λ-binders to become superfluous. Consider the term $(\lambda\alpha{:}\mathsf{Type}.\,\lambda x{:}\alpha.\,x)\ Nat\ 5$. After erasing type annotations, we are left with $(\underline{\lambda\alpha.}\,\lambda x.\,x)\ \underline{Nat}\ 5$, in which the binder $\lambda\alpha$ is superfluous because α no longer appears anywhere in its scope. For any such dummy binder λx, the resulting specialized β rule

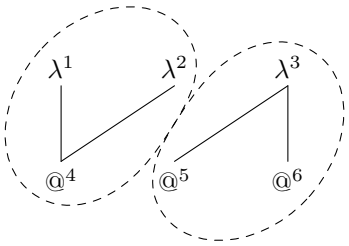
$$(\lambda x.\,M)\ N \rightarrow_\beta M \qquad\qquad \text{if } x \notin FV(M)$$

discards both the dummy binder λx and the argument N to which it would otherwise bind x. Therefore we may erase both the binding site λx and any argument N at an application site to which this λ-abstraction may flow during program execution. By this reasoning, we may erase the underlined portions of our previous example term, resulting in $(\lambda x.\,x)\ 5$.

However, during the execution of a program, other λ-abstractions may flow to some of those same application sites. We should not erase the argument N at an application site[3] $M@N$ unless every λ-abstraction that may flow to be the value of M has a dummy binder that also ends up being erased. Similarly, we should not erase a dummy binder unless we also erase every argument to whose application site it may otherwise flow. In general, the "may-flow-to" relation induces a bipartite graph (see Figure 1.3). We call this graph the $\lambda/@$ graph of a term. In order to decide if a given λ-binder or @-argument may be safely erased, we must analyze all λ-binders in its connected component (CC) in the $\lambda/@$ graph. If they are all dummies, then every λ-binder and @-argument in the CC may be safely erased. In this case, we call the CC *erasable*.

In this type of erasure step, the *usage* of a term determines its erasability. The (local) erasability of a binder λx depends on how x is (or is not) used in its scope.

[3]We sometimes write @ for application in order to have a more tangible notation than mere juxtaposition.



$$\begin{array}{l} \underline{\mathsf{let}}\ f = \lambda^1 x.\, 5\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ g = \lambda^2 y.\, y\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ h = \lambda^3 z.\, z@^4 7\ \underline{\mathsf{in}} \\ (h@^5 f, h@^6 g) \end{array} \not\Rightarrow \begin{array}{l} \underline{\mathsf{let}}\ f = 5\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ g = \lambda y.\, y\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ h = \lambda z.\, z\ \underline{\mathsf{in}} \\ (h\ f, h\ g) \end{array}$$

Figure 1.3: The $\lambda/@$ graph induced by the "may-flow-to" relation of a simple program. The fact that λ^2 is non-dummy prevents erasure of both the $@^4$-argument and the λ^1-binder. The interdependence of λ^1 and λ^2 is reflected in the $\lambda/@$ graph.

The erasability of an argument N depends on its context — whether the function that is applied to it always ends up being a λ-abstraction with a dummy binder.

Cascading Erasure

Erasure of @-arguments may make other λ-binders into dummies, thereby enabling erasure in other CCs of the $\lambda/@$ graph. Consider the following program that defines a family of identity functions.

$$\begin{array}{l} \underline{\mathsf{let}}\ id_0 = \lambda a{:}s.\, \lambda x{:}a.\, x\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ id_1 = \lambda a{:}s.\, \lambda x{:}a.\, id_0\ a\ x\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ id_2 = \lambda a{:}s.\, \lambda x{:}a.\, id_1\ a\ x\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ id_3 = \lambda a{:}s.\, \lambda x{:}a.\, id_2\ a\ x\ \underline{\mathsf{in}} \\ \ldots \end{array} \Rightarrow \begin{array}{l} \underline{\mathsf{let}}\ id_0 = \lambda a.\, \lambda x.\, x\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ id_1 = \lambda a.\, \lambda x.\, id_0\ a\ x\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ id_2 = \lambda a.\, \lambda x.\, id_1\ a\ x\ \underline{\mathsf{in}} \\ \underline{\mathsf{let}}\ id_3 = \lambda a.\, \lambda x.\, id_2\ a\ x\ \underline{\mathsf{in}} \\ \ldots \end{array}$$

After the initial erasure of domain annotations, a cascading sequence of $\lambda/@$ erasure steps is possible in this program. (Consider the λa binders).

In summary, we have explored several situations in which erasure of some part of a program was justified. In each instance, the rationale for erasing part of a program had to do with the *context* in which a term (e.g., a type annotation or a variable) appeared.

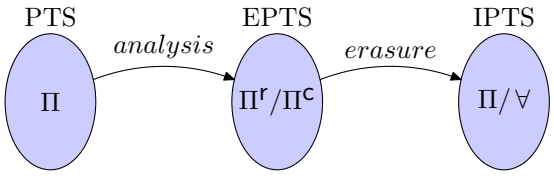


Figure 1.4: Two-phase approach to erasure semantics for Pure Type Systems in terms of two PTS variants EPTS and IPTS with support for erasure annotations and implicit polymorphism, respectively.

1.5 METHODOLOGY AND OVERVIEW

We treat computational irrelevance as a property not of a term itself, but of the context in which it is used. In λ-calculus, functions reify contexts of use, so we track relevance as a property of functions by distinguishing between functions that do not depend computationally on their arguments (of type $\forall x{:}A.\, B$) and those that might (of type $\Pi x{:}A.\, B$).

Note that the same A is used in both cases, because erasability is no longer an intrinsic property of x, but rather a property of the (functional) contexts making use of x. In this way we avoid the code duplication problem. We have *one* type A and therefore functions over A can be written *once*.

Our study of erasure semantics for dependently typed languages takes place in the context of three different families of typed λ-calculi, shown in Figure 1.4. The first family is *Pure Type Systems* (PTS), a well-known formalism that encompasses and organizes a wide variety of type systems [4]. Most dependently typed languages have a PTS at their core, therefore PTS is a good setting for studying features of dependently types languages. Chapter 2 briefly reviews the basics of PTS.

The next two language families are Erasure Pure Type Systems (EPTS) and Implicit Pure Type Systems (IPTS). Each is a conservative extension of PTS. EPTS extends PTS with support for erasure annotations indicating the computa-

tional relevance of various parts of a program. IPTS extends PTS with support for implicit parametric polymorphism over values of any type. IPTS is closely related to Miquel's Implicit Calculus of Constructions [64], but EPTS is a novel contribution of this thesis.

Two translations connect these three languages: (1) A program analysis that introduces optimal erasure annotations into a program and (2) an erasure phase guided by these annotations that removes the portions of a program marked as computationally irrelevant.

The erasure operation is the basis for the erasure semantics (Section 3.3.2). We prove that erasure exhibits properties one would expect: It respects the static and dynamic semantics of programs and *eliminates portions of the source program that do not affect its final value.*

The division of labor in Figure 1.4 is reminiscent of off-line partial evaluation, which consists of two tasks: a binding time analysis phase that annotates a program for specialization and a program specialization phase that simply obeys these annotations. This separation of concerns allows us to see the issues involved more clearly and allows the possibility of programming directly in the annotated language. In partial evaluation, MetaML[4] and its successor MetaOCaml[5] are each examples of such an annotated intermediate language.

[4] `http://www.cse.ogi.edu/PacSoft/projects/metaml/`
[5] `http://www.metaocaml.org/`

Chapter 2

REVIEW OF PURE TYPE SYSTEMS

The framework of Pure Type Systems (PTS) organizes type theory by illuminating the common structure in several notions previously thought to be unrelated: functions, parametric polymorphism, type constructors, and dependent types. This chapter introduces several type systems of increasing strength, culminating in the definition of PTS. It is hoped that seeing the range of type systems that may be cast as a Pure Type System will demonstrate the expressiveness of the formalism.

2.1 HISTORY

We first place Pure Type Systems in their historical context by briefly outlining the history of type theory in computer science and its application to proof assistants and programming languages. The remaining sections of this chapter will formally present several of the type theories discussed in this section in uniform, modern notation.

In 1932 and 1933, Alonzo Church developed the λ-calculus with the goal of formalizing mathematics [19, 20]. The λ-calculus turned out to be a universal model of computation. In 1940, Church applied the simple theory of types to his calculus and showed how to represent well-formed logical formulae with typed λ-terms [21].

In 1934, Gerhard Gentzen ushered in a new era of proof theory by introducing the notions of *natural deduction*, *sequent calculus*, and *cut elimination* [35, 36]. These constructions formalize proofs as mathematical objects whose structure can

be rigorously analyzed. Natural deduction formalizes the rules of logic as they are used by mathematicians. Sequent calculus is another way of looking at proofs that highlights the symmetries of classical logic. Cut elimination is a rule for normalizing proofs so that they use no intermediate lemmas.

In the late 1950s and 1960s, Church's λ-calculus began to impact actual programming languages. In 1958, John McCarthy created LISP, the first functional programming language [62]. It supported the λ notation for anonymous functions. In 1965, Peter Landin identified the λ-calculus as the core of ALGOL [47, 48] and outlined the design of ISWIM [49], a hypothetical functional language that had a large influence on later functional languages including ML, Miranda, and Haskell. Landin also developed the SECD machine, an abstract machine for evaluating λ-terms [46].

Also in the late 1950s and 1960s, the so-called *propositions as types* correspondence between various constructive logics and typed λ-calculi was discovered. In 1958, Haskell Curry observed "a close correspondence between axioms of positive implicational propositional logic, on the one hand, and basic combinators on the other hand" [30]. In 1967, William Tait discovered a correspondence between cut elimination and reduction in the λ-calculus [89]. In 1969, William Howard circulated a manuscript clarifying the correspondence [42]. The "Curry-Howard correspondence" was born.

The main idea of the correspondence is that a typed λ-calculus can be interpreted as a proof system in natural deduction style for a constructive logic. In this interpretation, types are read as logical propositions and a term inhabiting a type is read as a proof for the corresponding proposition. Furthermore, β-reducing a λ-term translates to eliminating a cut step in the corresponding proof. Howard's manuscript demonstrates this correspondence for two logics: propositional logic and Heyting arithmetic.

Also during the 1960s, Nicolaas G. de Bruijn's AUTOMATH project pioneered

the area of computer-verified mathematics [31]. AUTOMATH was a series of languages for writing mathematical proofs that were then checked for validity by a computer. These languages were built on the propositions-as-types principle and were the first computer implementation of this idea.

The 1970s saw the extension of the Curry-Howard correspondence to more expressive logics. In 1971, Jean-Yves Girard extended the correspondence to first-order (and higher-order) propositional logic with the introduction of System F, a calculus in which terms may be parameterized by types (and type constructors) [37, 38]. In his study of polymorphism in programming languages, John Reynolds independently developed essentially the same language in 1974 [76, 75]. In the 1970s, Per Martin-Löf introduced his type theory in several iterations [55, 56, 57]. Martin-Löf's type theory extended the correspondence to higher-order predicate logic with the introduction of general product and sum types that correspond to universal and existential quantifiers.

In 1984, Thierry Coquand and Gérard Huet developed the Calculus of Constructions [27, 25, 28], effectively combining the impredicative System F of Girard and the predicative type theory of Martin-Löf into a single calculus. The Calculus of Constructions is an expressive logic as well as a programming language. Christine Paulin-Mohring later extended this pure λ-calculus with inductive datatypes, thereby making it more practical as a programming language.

In 1987, Robert Harper, Furio Honsell, and Gordon Plotkin developed LF, the Edinburgh Logical Framework [39]. LF allows one to encode the formulas and typing derivations of many diverse logical systems as terms in a dependently typed λ-calculus. The LF type checker is then able to check the well-formedness of logical formulae as well as the validity of proofs encoded as LF terms. LF has similar goals to AUTOMATH, but with an emphasis on a methodology for encoding a wide variety of logics.

In 1989, Henk Barendregt analyzed the fine structure of the Calculus of Con-

structions by categorizing all its possible dependencies into four kinds: (1) terms depending on terms (regular functions), (2) terms depending on types (polymorphism), (3) types depending on types (type constructors), and (4) types depending on terms (dependent types). While all λ-calculi support dependencies of kind 1, Barendregt showed that any combination of the remaining choices 2–4 leads to a meaningful calculus [3, 4]. For example, previously known calculi such as the simply-typed λ-calculus, System F, System F$^\omega$, and LF fall out as instances of this general scheme. Shortly thereafter, Stefano Berardi and Jan Terlouw independently introduced Pure Type Systems as a generalization of Barendregt's cube that captures even more known calculi [4].

The arrival of the λ-cube and Pure Type Systems gave structure to what had become a diverse jungle of typed λ-calculi. It clarified relationships between various type theories and provided a general setting in which results about type theories may be proved once and instantiated at any particular PTS.

2.2 THE λ-CALCULUS

Church's λ-calculus is defined in modern notation in Figure 2.1. It is a pure calculus of anonymous functions. The function f defined as $f(x) = M$ (where M is the body of the function being defined) is expressed as $\lambda x.\, M$. Such a term is called a λ-*abstraction* and represents an anonymous function. We call M the *body* and x the *parameter* of the λ-abstraction. Successive λ-abstractions $\lambda x_1.\, \lambda x_2. \cdots \lambda x_n.\, M$ may be abbreviated as $\lambda x_1, x_2, \cdots, x_n.\, M$. Application of a function M to an argument N is written simply as the juxtaposition $M\ N$, instead of the more standard mathematical notation $M(N)$ with parentheses.

We call the calculus pure because functions are the only things one can directly express in the language. The only three forms of terms are variables (x), anonymous functions ($\lambda x.\, M$), and function applications ($M\ N$). The meaning of a function is embodied in the BETA reduction rule $(\lambda x.\, M)\ N \to_\beta M[N/x]$. This

Syntax

$$(terms) \quad M, N \quad ::= \quad x \mid \lambda x.\, M \mid M\ N$$

Reduction $\boxed{M \to_\beta M'}$

$$\textsc{Beta}\ \frac{}{(\lambda x.\, M)\ N \to_\beta M[N/x]} \qquad \textsc{LamCong}\ \frac{M \to_\beta M'}{\lambda x.\, M \to_\beta \lambda x.\, M'} \qquad \textsc{AppCong1}\ \frac{M \to_\beta M'}{M\ N \to_\beta M'\ N}$$

$$\textsc{AppCong2}\ \frac{N \to_\beta N'}{M\ N \to_\beta M\ N'}$$

Conversion $\boxed{M =_\beta M'}$

$$\textsc{Step}\ \frac{M \to_\beta N}{M =_\beta N} \qquad \textsc{Refl}\ \frac{}{M =_\beta M} \qquad \textsc{Symm}\ \frac{M =_\beta M'}{M' =_\beta M} \qquad \textsc{Trans}\ \frac{M =_\beta M' \qquad M' =_\beta M''}{M =_\beta M''}$$

Figure 2.1: The λ-calculus

rule says how to compute the application of a known function to an argument: by substitution of the argument N for the function parameter x in the function body M.

Before defining substitution $M[N/x]$, we must introduce the notions of bound and free occurrences of variables in a term. In a λ-abstraction $\lambda x.\, M$, the λ serves to *introduce* the variable x inside a particular *scope*, namely the body M. We say that the λ-abstraction is a *binding construct* (also λ is a *binder*) and that the λ *binds* x in M. However, there may be other variables in M that are not bound by any enclosing λ-binder in M. These variables are said to be *free* in M (and also in $\lambda x.\, M$). These notions are formalized as follows:

Definition 2.2.1 *Free and bound variables in a term.* $\boxed{FV(M)}$ *and* $\boxed{BV(M)}$

$$\begin{array}{rclcrcl} FV(x) & = & \{x\} & & BV(x) & = & \emptyset \\ FV(\lambda x.\, M) & = & FV(M) \;/\; \{x\} & & BV(\lambda x.\, M) & = & BV(M) \cup \{x\} \\ FV(M\ N) & = & FV(M) \cup FV(N) & & BV(M\ N) & = & BV(M) \cup BV(N) \end{array}$$

Note that a variable can occur both free and bound in a term. For example, if $M = (\lambda x.\, x)\ x$, then $FV(M) = BV(M) = \{x\}$. The meaning of bound variable occurrences in a term M is determined by the enclosing λ-binder. But the meaning of the free variables in M are determined by the context in which we find M.

Now we can define substitution. The definition is surprisingly involved because we want to respect the relationship between a variable occurrence and its binder. For instance, when substituting y for x in $\lambda x.\, x$, we should not return $\lambda x.\, y$ because that would break the relationship between the occurrence of x in the body of $\lambda x.\, x$ with the binder λx. Conversely, when substituting x for y in $\lambda x.\, y$, we should not return $\lambda x.\, x$ because that would introduces a binding relationship for x where there was not one before. The latter issue is called *variable capture* and is more subtle than the former.

Definition 2.2.2 *Substitution of N for free occurrences of x in M.* $\boxed{M[N/x]}$

$$(y)[N/x] = \begin{cases} N & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \qquad (M\ M')[N/x] = (M[N/x])\ (M'[N/x])$$

$$(\lambda y.\, M)[N/x] = \begin{cases} \lambda y.\, M & \text{if } x = y \\ \lambda y.\, M[N/x] & \text{if } x \neq y \text{ and } y \notin FV(N) \\ \lambda z.\, (M[z/y])[N/x] & \text{if } x \neq y \text{ and } y \in FV(N) \\ & \text{where } z \notin FV(M) \cup FV(N) \cup \{x\} \end{cases}$$

Bound variables can always be renamed consistently within their scope without changing the meaning of the term. Consider the definitions $f(x) = x + 1$ and $f(y) = y + 1$. They define the exact same function because renaming the variable x doesn't change the function being defined. This notion of equivalence up to renaming of bound variables is known as α-equivalence or α-conversion.

Definition 2.2.3 *α-conversion* $\boxed{M =_\alpha N}$

$$\frac{z \notin FV(M)}{\lambda y.\, M =_\alpha \lambda z.\, M[z/y]}$$

As is common, we consider α-equivalent terms to be syntactically identical.

Another equality that is sometimes considered in λ-calculus is η-conversion.

Definition 2.2.4 *η-conversion* $\boxed{M =_\eta N}$

$$\frac{x \notin FV(M)}{\lambda x.\, M\ x =_\eta M}$$

η-conversion is a weak extensionality principle. These two functions are in some sense equivalent, because they have the same result when applied to any argument.

In this language, all functions take a single argument. Functions of two arguments are represented as functions of one argument whose value is another function

of one argument. For example, the function f defined as $f(x, y) = M$ is represented as $\lambda x.\, \lambda y.\, M$. The application of this function f to arguments N_1 and N_2 is written as $(f\ N_1)\ N_2$ rather than $f(N_1, N_2)$. The application operation is left-associative, so we may write $f\ N_1\ N_2$ instead of $(f\ N_1)\ N_2$. This representation technique for multi-argument functions is knows as *currying*, after Haskell Curry, and it readily generalizes to n-argument functions for $n \geq 2$.

The λ-calculus comes equipped with a term-rewriting semantics given by the reduction relation $M \rightarrow_\beta N$, indicating that M transitions to N in a single step of computation. Such a step happens when the BETA rule applies somewhere inside the term M. Such a reducible subterm in M is always of the form $(\lambda x.\, M)\ N$ and is known as a *β-redex*.

We will sometimes refer to various closures of this single-step relation. The relation $\rightarrow_\beta^+$ is the transitive closure of $\rightarrow_\beta$ and the relation $\rightarrow_\beta^*$ is the reflexive transitive closure. The most basic notion of computational equality for the λ-calculus is that of β-conversion ($=_\beta$), which is defined as the reflexive, symmetric, transitive closure of the β-reduction relation $\rightarrow_\beta$ (see Figure 2.2).

One of the most important properties of the $\rightarrow_\beta$ relation is confluence. The confluence property of $\rightarrow_\beta$ states that any time a term M may step to two different terms N_1 and N_2, then there is a common term M' to which both N_1 and N_2 step (in zero or more steps). In symbols:

Theorem 2.2.5 (Confluence of $\rightarrow_\beta$ / Church-Rosser)
If $M \rightarrow_\beta N_1$ and $M \rightarrow_\beta N_2$, then there is some M' such that $N_1 \rightarrow_\beta^* M'$ and $N_2 \rightarrow_\beta^* M'$.

Though the λ-calculus allows one to speak directly only of functions, other mathematical entities may be encoded as λ-calculus terms as well. For example, the natural numbers may be encoded as λ-terms in the following way:

Definition 2.2.6 *The Church encoding* $\boxed{\overline{n}}$ *of the natural number* n

$$\overline{n} \;=\; \lambda s.\, \lambda z.\, s^n(z)$$

This definition relies on an auxiliary definition of iterated application

Definition 2.2.7 *Iterated application* $\boxed{M^n(N)}$

$$\begin{aligned} M^0(N) &= N \\ M^{n+1}(N) &= M^n(M\ N) \end{aligned}$$

The representation of n is, therefore, a two-argument iteration function that applies its first argument n times to its second argument. For example

$$\overline{4} = \lambda s.\, \lambda z.\, s\ (s\ (s\ (s\ z))).$$

This is but one example of a more general scheme of *Church encodings*, a general scheme whereby any algebraic datatype may be encoded in λ-calculus.

We may also encode the operations of addition (*plus*) and multiplication (*times*) in such a way that $\overline{n+m} =_\beta plus\ \overline{n}\ \overline{m}$ and $\overline{n \times m} =_\beta times\ \overline{n}\ \overline{m}$.

Definition 2.2.8 *Encodings of addition and multiplication*

$$\begin{aligned} plus &= \lambda n.\, \lambda m.\, \lambda s.\, \lambda z.\, n\ s\ (m\ s\ z) \\ times &= \lambda n.\, \lambda m.\, \lambda s.\, \lambda z.\, n\ (m\ s)\ z \end{aligned}$$

In fact, one may encode all computable functions as λ-terms, and thereby prove that the λ-calculus is a Turing-complete system of computation. As such, this calculus can be seen as the core of modern day functional programming languages.

However complete the language may be for computation, it has some problematic aspects when considered as a language for formalizing mathematics. In particular, the meaning of some terms is in unclear, because they never reduce

down to some irreducible term. For example, the term $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$ reduces in one step to itself:

$$\begin{aligned}(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) &\rightarrow_\beta (x\ x)[(\lambda x.\, x\ x)/x] \\ &= (x[(\lambda x.\, x\ x)/x])\ (x[(\lambda x.\, x\ x)/x]) \\ &= (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)\end{aligned}$$

A term like this in which we may continue making reduction steps forever is said to be *divergent*. Non-divergent terms — those which have *some* terminating sequence of reduction steps — are called *weakly normalizing*. Due to confluence of $\rightarrow_\beta$, every weakly normalizing term M is also *strongly normalizing* (i.e., *every* reduction sequence starting at M terminates).

It is impossible to distinguish normalizing and divergent terms mechanically. However, with the introduction of types one can often ensure that well-typed programs don't diverge. Several of the typed languages we will now discuss have this property. Assuming that type-checking is decidable, this means that there will always be some normalizing terms that our type system will reject as ill-typed. However, functions requiring a such term for their definition are rare in practice.

2.3 CHURCH'S SIMPLY-TYPED λ-CALCULUS

In 1940 Church applied the simple theory of types to his calculus and showed how to represent well-formed logical formulae with typed λ-terms [21]. The resulting language is known as the simply typed λ-calculus (STLC) and is presented in Figure 2.2. There are only two forms of types: function types and type variables. The function arrow $\rightarrow$ associates to the right, so that we may write the type $A_1 \rightarrow (A_2 \rightarrow B)$ of a curried two-argument function as $A_1 \rightarrow A_2 \rightarrow B$. The syntax of terms is the same as in the untyped λ-calculus, except that λ-abstractions are annotated with the domain type of the represented function. Successive λ-

Syntax

$$\begin{array}{rrcl} (\textit{types}) & A, B & ::= & \alpha \mid A \to B \\ (\textit{terms}) & M, N & ::= & x \mid \lambda x{:}A.\, M \mid M\ N \\ (\textit{contexts}) & \Gamma, \Delta & ::= & \varepsilon \mid \Gamma, x{:}A \end{array}$$

Typing Rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \to\textsc{Intro}\ \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B} \qquad \to\textsc{Elim}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

Figure 2.2: Simply typed λ-calculus

abstractions with the same domain annotation $\lambda x_1{:}A.\, \lambda x_2{:}A. \cdots \lambda x_n{:}A.\, M$ may be abbreviated as $\lambda x_1, x_2, \ldots, x_n{:}A.\, M$.

In addition to the reduction rules for the untyped λ-calculus, the simply-typed λ-calculus has a type system: an inference system for the judgment $\Gamma \vdash M : A$. This judgment means that term M has type A under assumptions Γ. The assumptions in Γ state the types of the free variables in M. (We assume that the term variables in Γ are distinct.)

The typing rules are straightforward. Rule Var says a variable has the type assigned to it by the typing context Γ. Rule $\to$Intro says a λ-abstraction has the type $A \to B$ if A matches the domain annotation on the abstraction and the body M of the abstraction has type B under the additional assumption that the formal parameter x has type A. Rule $\to$Elim says an application of a function of type $A \to B$ to an argument of type A has type B.

The inference rules for the type system can also be viewed as a minimal logical system. Under this interpretation, the types are read as *propositions* — either propositional variables α or implications $A \to B$ (read "A implies B") — and the terms are read as *proof terms*. The typing rules are then seen as inference rules.

Rules VAR and →INTRO capture hypothetical reasoning and the rule →ELIM is the well-known rule of modus ponens. The logic formalized by these rules is called minimal intuitionistic propositional logic. The first to notice this correspondence was Curry [30].

For example, if one interprets the variables α_1, α_2, and α_3 to stand for the propositions "It is raining", "I am wet", and "I am cold", respectively, then the derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \alpha_2 \to \alpha_3} \quad \dfrac{\overline{\Gamma \vdash g : \alpha_1 \to \alpha_2} \quad \overline{\Gamma \vdash x : \alpha_1}}{\Gamma \vdash g\ x : \alpha_2}}{\Gamma \vdash f\ (g\ x) : \alpha_3}}{f{:}\alpha_2 \to \alpha_3, g{:}\alpha_1 \to \alpha_2 \vdash \lambda x{:}\alpha_1.\, f\ (g\ x) : \alpha_1 \to \alpha_3}}{f{:}\alpha_2 \to \alpha_3 \vdash \lambda g{:}\alpha_1 \to \alpha_2.\, \lambda x{:}\alpha_1.\, f\ (g\ x) : (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_3}}{\vdash \lambda f{:}\alpha_2 \to \alpha_3.\, \lambda g{:}\alpha_1 \to \alpha_2.\, \lambda x{:}\alpha_1.\, f\ (g\ x) : (\alpha_2 \to \alpha_3) \to (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_3}$$

(where $\Gamma = f{:}\alpha_2 \to \alpha_3, g{:}\alpha_1 \to \alpha_2, x{:}\alpha_1$) ensures that

$$\lambda f{:}\alpha_2 \to \alpha_3.\, \lambda g{:}\alpha_1 \to \alpha_2.\, \lambda x{:}\alpha_1.\, f\ (g\ x)$$

is a well-formed proof of $(\alpha_2 \to \alpha_3) \to (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_3$, which is interpreted as the following proposition:

> If it is the case that (1) if I am wet then I am also cold, and (2) if it is raining then I am wet, and (3) it is raining, then it is also the case that I am cold.

The correspondence between the simply-typed λ-calculus and logic also extends to the reduction relation $M \to_\beta N$. This relation preserves types, so that if $\Gamma \vdash M : A$ and $M \to_\beta N$, then $\Gamma \vdash N : A$. With programming in mind, this says that if our program computes a value (reduces eventually to some irreducible term), then that value has the same type as the original term. With logic in mind, this says that we may regard term reduction as a form of proof simplification. Tait [89] was the first to note that the typing correspondence extends to β-reduction. All the

type systems reviewed in this chapter have this property that reduction preserves types, known as *subject reduction*.

One interesting property of the simply-typed λ-calculus is that every well-typed term is strongly normalizing. Because this holds of any well-typed term in the language, we say that the simply-typed λ-calculus itself is strongly normalizing. This means that the language is not Turing complete. However, the language may be used to program certain forms of iteration, as in the natural numbers of the previous section.

Church's original motivation for the simply-typed λ-calculus was to represent logical formulas. For example, one may introduce a special type variable o as the type of logical formulas and introduce the following typed constants[1].

$$\begin{array}{lcllcl} \neg & : & o \to o & \forall_A & : & (A \to o) \to o \\ \wedge & : & o \to o \to o & \exists_A & : & (A \to o) \to o \\ \Rightarrow & : & o \to o \to o & =_A & : & A \to A \to o \end{array}$$

In this way, one may represent the logical formula

$$\forall x, y, z : A.\ x =_A y \wedge y =_A z \Rightarrow x =_A z$$

as the λ-term

$$\forall_A(\lambda x{:}A.\, \forall_A(\lambda y{:}A.\, \forall_A(\lambda z{:}A.\, \Rightarrow\ (\wedge\ (=_A\ \ x\ y)\ (=_A\ \ y\ z))\ (=_A\ \ x\ z))))$$

Note that we are introducing an infinite number of constants here: because there are an infinite number of types A, there are infinitely many constants $=_A$ (similarly for $\forall_A$ and $\exists_A$).

The typing rules of the simply-typed λ-calculus ensure that logical formulas represented in this way are syntactically well-formed. The types also prevent certain paradoxes from arising. For example, let us represent sets of objects as o-valued functions in the following way:

[1] A constant is treated like a variable with global scope. It has no reduction rules, and is assigned the same type at each occurrence.

1. The set comprehension $\{x : A \mid M\}$ is represented as the λ-abstraction $\lambda x{:}A.\, M$.

2. The set membership operation $N \in M$ is represented as the application $M\ N$.

Under this encoding, Russell's paradox — that the set of all sets that do not include themselves ($S = \{X \mid X \notin X\}$) both is and is not a member of itself ($S \in S \iff S \notin S$) — is represented by the divergent λ-term $R = S\ S$ where $S = \lambda A{:}x.\, \neg\ (x\ x)$ for some type A. This leads to a paradox, because $R =_\beta \neg R$.

$$\begin{aligned} R \quad &= \quad S\ S \quad = \quad (\lambda A{:}x.\, \neg\ (x\ x))\ (\lambda A{:}x.\, \neg\ (x\ x)) \\ &\to_\beta \quad \neg\ ((\lambda A{:}x.\, \neg\ (x\ x))\ (\lambda A{:}x.\, \neg\ (x\ x))) \quad = \quad \neg\ (S\ S) \quad = \quad \neg\ R \end{aligned}$$

However, the term R is not well-formed because the parameter x in S must have both types A and $A \to o$, thereby violating the inductive nature of the syntax of types. In this way, the type system renders certain vicious circles of logic as circular types, which are themselves illegal and easy to spot.

Note that the use of simply typed λ-terms to represent logical formulas is different from the Curry-Howard correspondence. In the former, propositions are *represented as terms*, while in the latter, propositions *correspond directly to types*. In the latter, well-typed terms correspond to proofs, but in the former, we have no way of representing proofs, although in Section 2.6 will introduce a more powerful type system in which terms may represent proofs.

2.4 THE GIRARD/REYNOLDS POLYMORPHIC λ-CALCULUS

In the simply-typed λ-calculus, one can define an identity function $\lambda x{:}A.\, x$ of type $A \to A$ for each type A. However, we cannot define *one* identity function that works for *any* type A. Reynolds invented the polymorphic λ-calculus in order to program families of functions of this sort. Girard, seeking to extend

Syntax

$$
\begin{array}{rrcl}
(\textit{types}) & A, B & ::= & \alpha \mid A \to B \mid \forall \alpha.\, B \\
(\textit{terms}) & M, N & ::= & x \mid \lambda x{:}A.\, M \mid M\ N \mid \lambda \alpha.\, M \mid M\ A \\
(\textit{contexts}) & \Gamma, \Delta & ::= & \varepsilon \mid \Gamma, x{:}A \mid \Gamma, \alpha\ \mathsf{type}
\end{array}
$$

Well-formed types $\boxed{\Gamma \vdash A\ \mathsf{type}}$

$$
\textsc{TyVar}\ \frac{\alpha\ \mathsf{type} \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}
\qquad
\to\textsc{-Form}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash B\ \mathsf{type}}{\Gamma \vdash A \to B\ \mathsf{type}}
\qquad
\forall\textsc{-Form}\ \frac{\Gamma, \alpha\ \mathsf{type} \vdash B\ \mathsf{type}}{\Gamma \vdash \forall \alpha.\, B\ \mathsf{type}}
$$

Well-formed terms $\boxed{\Gamma \vdash M : A}$

$$
\textsc{Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}
\qquad
\to\textsc{-Intro}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B}
\qquad
\to\textsc{-Elim}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}
$$

$$
\forall\textsc{-Intro}\ \frac{\Gamma, \alpha\ \mathsf{type} \vdash M : B}{\Gamma \vdash \lambda \alpha.\, M : \forall \alpha.\, B}
\qquad
\forall\textsc{-Elim}\ \frac{\Gamma \vdash M : \forall \alpha.\, B \qquad \Gamma \vdash A\ \mathsf{type}}{\Gamma \vdash M\ A : B[A/\alpha]}
$$

Figure 2.3: System F / The polymorphic λ-calculus

the Curry-Howard isomorphism to (intuitionistic) second-order propositional logic, independently developed the same calculus and gave it the name System F.

In this language, one can write $I = \lambda\alpha.\,\lambda x{:}\alpha.\,x$ which has type $\forall\alpha.\,\alpha \to \alpha$. In fact, our original identity function at type A is equivalent in this language to $I\ A$ of type $A \to A$. The upshot is that programmers can define such *polymorphic* functions once and reuse them at many different types. System F has inspired the use of polymorphism in the type systems of many functional languages.

Figure 2.3 contains the syntax and typing rules for System F. There is an additional type former $\forall\alpha.\,B$ indicating the type of a polymorphic entity that may take on the type $B[A/\alpha]$ for any type A. At the level of terms, we have two new constructs for introducing $(\lambda\alpha.\,M)$ and eliminating $(M\ A)$ polymorphic entities. The typing rules $\forall$-Intro and $\forall$-Elim show how these terms are typed. We also have a new form of context entry α type and a new typing judgment $\Gamma \vdash A$ type. The purpose of the new typing judgment is basically to enforce the scoping rules for type variables. As before, successive λ-binders $\lambda\alpha_1.\,\lambda\alpha_2.\cdots\lambda\alpha_n.\,M$ may be abbreviated as $\lambda\alpha_1, \alpha_2, \ldots, \alpha_n.\,M$. Similarly, $\forall\alpha_1, \alpha_2, \ldots, \alpha_n.\,B$ abbreviates $\forall\alpha_1.\,\forall\alpha_2.\cdots\forall\alpha_n.\,B$.

This form of polymorphism is called *parametric polymorphism* as opposed to *ad hoc polymorphism*[2]. Parametric polymorphism occurs when a value like I has several different types, but at each type it behaves in exactly the same way. Ad hoc polymorphism occurs when a value like *plus* has different types (in this case, $int \to int \to int$, $float \to float \to float$, etc.) and behaves in different ways at each of those types.

[2]The distinction between parametric and ad hoc polymorphism is due to Strachey [86].

2.4.1 Impredicative Encodings

The encodings of datatypes that we discussed in the setting of untyped λ-calculus can be typed in simply-typed λ-calculus, but they can be given even more precise types in System F. For example, the natural numbers can be encoded using the type $Nat = \forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$. The previous encodings for zero, successor, etc. become:

$$\begin{aligned}
zero &= \lambda\alpha.\,\lambda s{:}\alpha \to \alpha.\,\lambda z{:}\alpha.\,z && : Nat \\
succ &= \lambda n{:}Nat.\,\lambda\alpha.\,\lambda s{:}\alpha \to \alpha.\,\lambda z{:}\alpha.\,s\ (n\ \alpha\ s\ z) && : Nat \to Nat \\
plus &= \lambda n, m{:}Nat.\,\lambda\alpha.\,\lambda s{:}\alpha \to \alpha.\,\lambda z{:}\alpha.\,n\ \alpha\ s\ (m\ \alpha\ s\ z) && : Nat \to Nat \to Nat \\
times &= \lambda n, m{:}Nat.\,\lambda\alpha.\,\lambda s{:}\alpha \to \alpha.\,\lambda z{:}\alpha.\,n\ \alpha\ (m\ \alpha\ s)\ z && : Nat \to Nat \to Nat
\end{aligned}$$

We can then encode additional types in terms of those already encoded. For example, the type of lists of naturals may be defined as

$$NatList = \forall\alpha.\,(Nat \to \alpha \to \alpha) \to \alpha \to \alpha.$$

Then we may define several operations on lists, for example:

$$\begin{aligned}
&nil\ :\ NatList \\
&nil = \lambda\alpha.\,\lambda c{:}Nat \to \alpha \to \alpha.\,\lambda n{:}\alpha.\,n
\end{aligned}$$

$$\begin{aligned}
&cons\ :\ Nat \to NatList \to NatList \\
&cons = \lambda x{:}Nat.\,\lambda xs{:}NatList. \\
&\qquad \lambda\alpha.\,\lambda c{:}Nat \to \alpha \to \alpha.\,\lambda n{:}\alpha. \\
&\qquad\quad c\ x\ (xs\ \alpha\ c\ n)
\end{aligned}$$

$$\begin{aligned}
&map\ :\ (Nat \to Nat) \to NatList \to NatList \\
&map = \lambda f{:}Nat \to Nat.\,\lambda xs{:}NatList. \\
&\qquad \lambda\alpha.\,\lambda c{:}Nat \to \alpha \to \alpha.\,\lambda n{:}\alpha. \\
&\qquad\quad xs\ \alpha\ (\lambda h{:}Nat.\,\lambda t{:}\alpha.\,c\ (f\ h)\ t)\ n
\end{aligned}$$

$$
\begin{aligned}
&append \ : \ NatList \rightarrow NatList \rightarrow NatList \\
&append = \lambda xs, ys{:}NatList. \\
&\qquad \lambda\alpha.\, \lambda c{:}Nat \rightarrow \alpha \rightarrow \alpha.\, \lambda n{:}\alpha. \\
&\qquad\quad xs\ \alpha\ c\ (ys\ \alpha\ c\ n)
\end{aligned}
$$

There is a certain type of circularity inherent in the System F rules for type formation. For example, in the type $Nat = \forall\alpha.\,(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$, the type variable α ranges over all types, including the type Nat itself. This sort of circularity — where one quantifies over some class of objects in order to define a member of that class — is called *impredicativity*. Remarkably, impredicativity does not lead to any paradox in System F. Girard proved that System F is strongly normalizing and therefore sound as a logic.

2.4.2 Relational Parametricity

Reynolds, in his celebrated *abstraction theorem*, proved that the type at which a polymorphic entity is instantiated does not affect its subsequent behavior [77]. He characterized this behavioral invariance by interpreting type variables as abstract types which can be implemented in several different ways. He then proved that the equivalent implementations yield equivalent behavior in clients of the abstract type. We try to explain his result here.

Consider an abstract type α of natural numbers supporting the operations $zero : \alpha$, $succ : \alpha \rightarrow \alpha$, and $even : \alpha \rightarrow boolean$. This type and its associated operations may be implemented in multiple ways. Say we have two implementations, namely $\{A_1, zero_1, succ_1, even_1\}$ and $\{A_2, zero_2, succ_2, even_2\}$. Now we want to say that these two implementations are observationally equivalent. How can we formalize that? First of all we say that there is a relation $R : A_1 \leftrightarrow A_2$ relating A_1 objects and A_2 objects that represent the same (abstract) natural number. In other words,

for $a_1 : A_1$ and $a_2 : A_2$, it is the case that $R(a_1, a_2)$ holds iff a_1 and a_2 represent the same natural number. Furthermore, we want the exported operations *zero*, *succ*, and *even* to respect this relation. The meaning of "respecting the relation" varies according to the type of each operation. Specifically, we require

- $R(zero_1, zero_2)$ — $zero_1$ and $zero_2$ represent the same number;
- For all $x_1 : A_1$ and $x_2 : A_2$ such that $R(x_1, x_2)$, we have $R(succ_1\ x_1, succ_2\ x_2)$ — parallel applications of the two successor operations to (two representations of) the same input yield (two representations of) the same output; and
- For all $x_1 : A_1$ and $x_2 : A_2$ such that $R(x_1, x_2)$, we have $even_1\ x_1 = even_2\ x_2$ — any two observations of two representations of the same natural number have the same outcome.

The property of being R-respecting for the various operations in the abstract interface may be systematically derived from their types α, $\alpha \to \alpha$, and $\alpha \to boolean$.

Now consider a client N of the abstract type α. Without loss of generality, we assume N has the type $\forall\alpha.\, \alpha \to (\alpha \to \alpha) \to (\alpha \to boolean) \to B$ for some type B. We may "link" this code to either of our two implementations of natural numbers by forming the applications $M_1 = N\ A_1\ zero_1\ succ_1\ even_1$ and $M_2 = N\ A_2\ zero_2\ succ_2\ even_2$. What we want to say is that these two terms behave in the same way *up to the pseudo-equivalence* R whenever R is a relation between A_1 and A_2 that is respected by the two implementations. This is exactly what Reynolds abstraction theorem says (in a more general way, of course) about the polymorphic λ-calculus.

We have been explaining Reynolds' abstraction theorem from the perspective of the implementation side of an abstraction barrier. When viewed from the client side, the abstraction theorem also says something about the behavior of polymorphic terms, such as our client code N. Because N must behave the same way when

"linked" against related implementations of α, it cannot be the case that N is able to inspect the structure of the particular α to which it is applied and determine its behavior based on the outcome of the inspection. Otherwise, it might behave differently when linked against different implementations of the same type, in contradiction to the abstraction theorem. Therefore, the notion of polymorphism in System F is parametric, rather than ad-hoc.

Wadler showed that this client-side view of the abstraction theorem can be used to prove useful properties about polymorphic functions [93]. He used the term *parametricity*[3] to refer to the constraint that Reynolds' abstraction theorem places on the behavior of polymorphic programs. Wadler demonstrated that several known and useful properties of polymorphic functions commonly used in functional programming follow by parametricity simply by virtue of the type of the function. These so-called "theorems for free" are beloved of functional programmers. The principle behind this class of properties is referred to as *relational parametricity* because it illuminates the nature of parametric polymorphism by interpreting types as relations.

2.5 GIRARD'S SYSTEM F^ω

Girard further developed System F into System F^ω, by generalizing to intuitionistic *higher-order* propositional logic. Figure 2.4 presents this language.

From a programming perspective, moving from System F to System F^ω means adding the language feature of *type constructors* (a.k.a. *type operators*). For example, in the previous section we defined a type for encoding lists of natural numbers, namely $\forall\alpha.\,(Nat \to \alpha \to \alpha) \to \alpha \to \alpha$. However, one usually wants to program with lists of various element types. In System F, we must define a new list type for each new element type. But in System F^ω, we can write the single type con-

[3]Wadler credits Bainbridge, Freyd, Girard, Scedrov, and Scott with coining the term "parametricity".

Syntax

$$
\begin{array}{rrcl}
(\textit{kinds}) & K & ::= & * \mid K \to K' \\
(\textit{types}) & A, B & ::= & \alpha \mid A \to B \mid \forall \alpha{:}K.\, A \mid \lambda \alpha{:}K.\, A \mid A\ B \\
(\textit{terms}) & M, N & ::= & x \mid \lambda x{:}A.\, M \mid M\ N \mid \lambda \alpha{:}K.\, M \mid M\ A \\
(\textit{contexts}) & \Gamma, \Delta & ::= & \varepsilon \mid \Gamma, x{:}A \mid \Gamma, \alpha{:}K
\end{array}
$$

Well-formed types $\boxed{\Gamma \vdash A : K}$

$$\textsc{TyVar}\ \frac{\alpha{:}K \in \Gamma}{\Gamma \vdash \alpha : K} \qquad \to\textsc{-Form}\ \frac{\Gamma \vdash A : \star \qquad \Gamma \vdash B : \star}{\Gamma \vdash A \to B : \star} \qquad \forall\textsc{-Form}\ \frac{\Gamma, \alpha{:}K \vdash A : \star}{\Gamma \vdash \forall \alpha{:}K.\, A : \star}$$

$$\to\textsc{-Intro}\ \frac{\Gamma, \alpha{:}K \vdash A : K'}{\Gamma \vdash \lambda \alpha{:}K.\, A : K \to K'} \qquad \to\textsc{-Elim}\ \frac{\Gamma \vdash A : K \to K' \qquad \Gamma \vdash B : K}{\Gamma \vdash A\ B : K'}$$

Well-formed terms $\boxed{\Gamma \vdash M : A}$

$$\textsc{Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \to\textsc{-Intro}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B} \qquad \to\textsc{-Elim}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\forall\textsc{-Intro}\ \frac{\Gamma, \alpha{:}K \vdash M : B}{\Gamma \vdash \lambda \alpha{:}K.\, M : \forall \alpha.\, KB} \qquad \forall\textsc{-Elim}\ \frac{\Gamma \vdash M : \forall \alpha{:}K.\, B \qquad \Gamma \vdash A : K}{\Gamma \vdash M\ A : B[A/\alpha]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \star \qquad A =_\beta B}{\Gamma \vdash M : B}$$

Figure 2.4: System F^ω

structor $List = \lambda\beta{:}\star.\, \forall\alpha{:}\star.\, (\beta \to \alpha \to \alpha) \to \alpha \to \alpha$. Then *List Nat* recovers the original type of lists of naturals, and we also have *List Bool*, *List* (*List Nat*), etc.

The presentation of System F^ω extends that of System F in four ways. First, there are λ-abstractions and applications at the level of types. We have already seen a type-level λ-abstraction in the definition $List = \lambda\beta{:}\star.\cdots$, and we have already seen type-level application in the examples of various concrete list types (e.g., *List Nat*).

Secondly, there is a new syntactic category called *kinds*. Kinds are to types as types are to terms. In other words, kinds are the types of type expressions. The well-formedness rules for types use kinds to prevent meaningless type expressions such as $(\lambda\alpha{:}\star.\, \alpha) \to (\lambda\alpha{:}\star.\, \alpha)$. Kinds take one of two forms: the base kind $\star$ (pronounced "star") or a function kind $K \to K'$. $\star$ is the kind of all proper types, that is types A of which it is meaningful to ask whether $\Gamma \vdash M : A$ for some M and Γ. Function kinds are the kinds of type constructors. Notice that there are type constructors with more complicated kinds than simply $\star \to \star$. For example, we may have type constructors with two (proper) type arguments (of kind $\star \to \star \to \star$) or type constructors which take a type constructor as an argument (of kind $(\star \to \star) \to \star$, for instance).

Thirdly, now that type variables may have kinds other than $\star$, we annotate each type-variable binding construct with the kind of that type variable.

Fourthly, because types may now contain beta-redices such as $(\lambda\alpha{:}K.\, A)\ B$, we need a more general notion of what it means for two types to be the same. In the type systems we have described previously, two types were considered to be the same whenever they were syntactically equal (up to α-conversion). But for type constructors to be at all useful, the type system must consider *List Nat*, which is simply an abbreviation for

$$(\lambda\beta{:}\star.\, \forall\alpha.\, (\beta \to \alpha \to \alpha) \to \alpha \to \alpha)\ (\forall\gamma.\, (\gamma \to \gamma) \to \gamma \to \gamma),$$

to be equal to $\forall\alpha.\,(Nat \to \alpha \to \alpha) \to \alpha \to \alpha$, which is simply an abbreviation for

$$\forall\alpha.\,((\forall\gamma.\,(\gamma \to \gamma) \to \gamma \to \gamma) \to \alpha \to \alpha) \to \alpha \to \alpha.$$

This is accomplished in the typing rules for System F^{ω} by the so-called *conversion rule* (named Conv in Figure 2.4). That rule says that if M has type A and if B is another type that is β-convertible with A, then M also has type B. The notion of equality specified in the conversion rule is also known as the *definitional equality* for the system because it provides the relationship between types that are considered to be equivalent *by definition*.

To complete the generic list type example, we generalize the definitions previously given for operations on lists of naturals.

$$\begin{aligned}
&nil \;:\; \forall\beta{:}\star.\, List\ \beta\\
&nil = \lambda\beta{:}\star.\,\lambda\alpha{:}\star.\,\lambda c{:}\beta \to \alpha \to \alpha.\,\lambda n{:}\alpha.\,n
\end{aligned}$$

$$\begin{aligned}
&cons \;:\; \forall\beta{:}\star.\,\beta \to List\ \beta \to List\ \beta\\
&cons = \lambda\beta{:}\star.\,\lambda x{:}\beta.\,\lambda xs{:}List\ \beta.\\
&\qquad\qquad \lambda\alpha{:}\star.\,\lambda c{:}\beta \to \alpha \to \alpha.\,\lambda n{:}\alpha.\\
&\qquad\qquad\quad c\ x\ (xs\ \alpha\ c\ n)
\end{aligned}$$

$$\begin{aligned}
&map \;:\; \forall\beta,\gamma{:}\star.\,(\beta \to \gamma) \to List\ \beta \to List\ \gamma\\
&map = \lambda\beta,\gamma{:}\star.\,\lambda f{:}\beta \to \gamma.\,\lambda xs{:}List\ \beta.\\
&\qquad\qquad \lambda\alpha{:}\star.\,\lambda c{:}\gamma \to \alpha \to \alpha.\,\lambda n{:}\alpha.\\
&\qquad\qquad\quad xs\ \alpha\ (\lambda h{:}\beta.\,\lambda t{:}\alpha.\,c\ (f\ h)\ t)\ n
\end{aligned}$$

$$\begin{aligned}
&append \;:\; \forall\beta{:}\star.\, List\ \beta \to List\ \beta \to List\ \beta\\
&append = \lambda\beta{:}\star.\,\lambda xs, ys{:}List\ \beta.\\
&\qquad\qquad \lambda\alpha{:}\star.\,\lambda c{:}\beta \to \alpha \to \alpha.\,\lambda n{:}\alpha.\,xs\ \alpha\ c\ (ys\ \alpha\ c\ n)
\end{aligned}$$

These polymorphic operations can be instantiated for use on lists of any element type. Notice how the type of *map* has been generalized so as to work with two different types of list.

2.6 THE EDINBURGH LOGICAL FRAMEWORK

The languages we have considered so far all correspond to various propositional logics, where all variables appearing in types range over propositions or (possibly higher-order) propositional functions. We now consider a type-theoretic rendering of predicate logic, where logical formulas can quantify over individuals in some domain of discourse.

The type system feature corresponding to predicate logic is called *dependent types*. Dependent types are types that depend on non-types. The simplest λ-calculus exhibiting dependent types is that of the Edinburgh Logical Framework (LF), defined in Figure 2.5. This language corresponds to intuitionistic first-order predicate logic.

The Π type-former has replaced the $\rightarrow$ of the simply-typed λ-calculus. The meaning of $\Pi x{:}A.\, B$ is similar to that of $A \rightarrow B$, except that the former names the eventual argument to which a function of this type will be applied. This name, x, may appear inside B, the return type of the function. In this way, the type of a function application may depend on the actual parameter passed to the function. The typing rule Π-Elim2 in Figure 2.5 shows how this happens. We continue to write $A \rightarrow B$ as an abbreviation for $\Pi x{:}A.\, B$ in the special case that x does not appear free in B.

Furthermore, we have an additional Π kind-former, as well as term-abstractions and term-applications at the type level. Using these, we may work with types dependent on terms. The typing rules for the Π at the kind level are similar to those for the Π at the type level. Again, we write $A \rightarrow K$ as an abbreviation for $\Pi x{:}A.\, K$ when x does not appear free in K.

Syntax

$$\begin{array}{rrcl} (\textit{kinds}) & K & ::= & \star \mid \Pi x{:}A.\, K \\ (\textit{types}) & A, B & ::= & \alpha \mid \Pi x{:}A.\, B \mid \lambda x{:}A.\, B \mid A\ N \\ (\textit{terms}) & M, N & ::= & x \mid \lambda x{:}A.\, M \mid M\ N \\ (\textit{contexts}) & \Gamma, \Delta & ::= & \varepsilon \mid \Gamma, x{:}A \end{array}$$

Well-formed kinds $\boxed{\Gamma \vdash K\ \mathsf{kind}}$

$$\textsc{Star}\ \frac{}{\Gamma \vdash \star\ \mathsf{kind}} \qquad \Pi\textsc{-Form1}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x{:}A \vdash K\ \mathsf{kind}}{\Gamma \vdash \Pi x{:}A.\, K\ \mathsf{kind}}$$

Well-formed types $\boxed{\Gamma \vdash A : K}$

$$\textsc{TyVar}\ \frac{\alpha{:}K \in \Gamma}{\Gamma \vdash \alpha : K} \qquad \Pi\textsc{-Form2}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x{:}A \vdash B : \star}{\Gamma \vdash \Pi x{:}A.\, B : \star} \qquad \Pi\textsc{-Intro1}\ \frac{\Gamma, x{:}A \vdash B : K}{\Gamma \vdash \lambda x{:}A.\, B : \Pi x{:}A.\, K}$$

$$\Pi\textsc{-Elim1}\ \frac{\Gamma \vdash A : \Pi x{:}A.\, K \qquad \Gamma \vdash N : A}{\Gamma \vdash A\ N : K[N/x]} \qquad \textsc{Conv1}\ \frac{\Gamma \vdash A : K \qquad \Gamma \vdash K'\ \mathsf{kind} \qquad K =_\beta K'}{\Gamma \vdash A : K'}$$

Well-formed terms $\boxed{\Gamma \vdash M : A}$

$$\textsc{Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \Pi\textsc{-Intro2}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : \Pi x{:}A.\, B} \qquad \Pi\textsc{-Elim2}\ \frac{\Gamma \vdash M : \Pi x{:}A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}$$

$$\textsc{Conv2}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \star \qquad A =_\beta B}{\Gamma \vdash M : B}$$

Figure 2.5: The Edinburgh Logical Framework (LF)

In Section 2.3, we saw that simply-typed λ-calculus can be used to represent logical formulas in such a way that type-correctness of the λ-term representing a logical formula ϕ guarantees the syntactic well-formedness of ϕ. In LF, we can go further and represent logical inference rules as types in such a way that type-correctness of certain "proof" terms ensures correctness of the proof so represented.

Consider the following fragment of propositional logic:

$$\phi \quad ::= \quad P \mid \phi \wedge \phi' \mid \phi \Rightarrow \phi' \mid \bot$$

where P indicates a propositional variable, and $\bot$ is the formula denoting a constant falsehood. Using the following simple LF signature (i.e., typing context), we may represent formulas of this logic as LF terms of type o.

$$o : \star \qquad \mathit{and} : o \to o \to o \qquad \mathit{imp} : o \to o \to o \qquad \mathit{false} : o$$

Often, intuitionistic negation is presented as a derived notion, defined as $\neg\, \phi = \phi \Rightarrow \bot$. We can also define this logical connective as a derived notion.

$$\mathit{not} \quad = \quad \lambda p{:}o.\, \mathit{imp}\ p\ \mathit{false} \quad : \quad o \to o$$

This much was possible in the simply-typed λ-calculus. However, in LF we can also declare a type of *proofs* of a particular proposition.

$$\mathit{proof} \quad : \quad o \to \star$$

The expression $o \to \star$ is a kind. The unique thing about this kind is that it is built up from a type (o) and a kind ($\star$) rather than from two kinds. A functional kind with a type for a domain and a kind for a codomain indicates a dependent type, because it classifies type expressions (of kind $\star$) parameterized over terms (of type o). We now declare constructors for the *proof* type with which we may build LF terms representing proofs in the same way that we can build LF terms of type o representing logical formulas.

$$
\begin{array}{rcl}
\mathit{and\_intro} & : & \Pi a{:}o.\, \mathit{proof}\ a \rightarrow \mathit{proof}\ b \rightarrow \mathit{proof}\ (\mathit{and}\ a\ b) \\
\mathit{and\_elim}_1 & : & \Pi a{:}o.\, \mathit{proof}\ (\mathit{and}\ a\ b) \rightarrow \mathit{proof}\ a \\
\mathit{and\_elim}_2 & : & \Pi a{:}o.\, \mathit{proof}\ (\mathit{and}\ a\ b) \rightarrow \mathit{proof}\ b \\
\mathit{false\_elim} & : & \Pi a{:}o.\, \mathit{proof}\ \mathit{false} \rightarrow \mathit{proof}\ a \\
\mathit{imp\_intro} & : & \Pi a{:}o.\, (\mathit{proof}\ a \rightarrow \mathit{proof}\ b) \rightarrow \mathit{proof}\ (\mathit{imp}\ a\ b) \\
\mathit{imp\_elim} & : & \Pi a{:}o.\, \mathit{proof}\ (\mathit{imp}\ a\ b) \rightarrow \mathit{proof}\ a \rightarrow \mathit{proof}\ b
\end{array}
$$

These declarations correspond to the following inference rules for our logic.

$$
\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B} \qquad
\frac{\vdash A \wedge B}{\vdash A} \qquad
\frac{\vdash A \wedge B}{\vdash B} \qquad
\frac{\vdash \bot}{\vdash A} \qquad
\frac{\begin{array}{c}[A] \\ \vdots \\ \vdash B\end{array}}{\vdash A \Rightarrow B} \qquad
\frac{\vdash A \Rightarrow B \qquad \vdash A}{\vdash B}
$$

So far, we have merely declared and applied constants with dependent types. Using the new form of λ-abstraction at the level of types, we may also define new terms with dependent types. For example, we may define inference rules for $\neg$ in terms of those for $\bot$ and $\Rightarrow$.

$$
\begin{array}{rcl}
\mathit{not\_elim} & : & \Pi a{:}o.\, \mathit{proof}\ a \rightarrow \mathit{proof}\,(\mathit{not}\ a) \rightarrow \mathit{proof}\ b \\
\mathit{not\_elim} & = & \lambda a{:}o.\, \lambda p{:}\mathit{proof}\ a.\, \lambda q{:}\mathit{proof}\,(\mathit{not}\ a). \\
 & & \quad \mathit{false\_elim}\ b\ (\mathit{imp\_elim}\ a\ \mathit{false}\ q\ p)
\end{array}
$$

This LF definition corresponds to the following derived rule in our logic.

$$
\frac{\vdash A \qquad \vdash \neg A}{\vdash B}
\quad \mapsto \quad
\dfrac{\dfrac{\vdash \neg A}{\vdash A \Rightarrow \bot} \qquad \vdash A}{\vdash \bot}
\Bigg/ \vdash B
$$

By representing a formula as a term M of type o and representing a proof of that formula as a term N of type $\mathit{proof}\ M$, we can apply the typing rules for LF to

check syntactic well-formedness of the formula M as well as validity of the proof N. This is the purpose of a logical framework — a general purpose system for defining and implementing logics. An implementation of (a type-checker for) LF can be used to check proofs in whatever logic we can encode as a LF signature.

2.7 COQUAND AND HUET'S CALCULUS OF CONSTRUCTIONS

All the previous languages may be combined into a single language: Coquand and Huet's Calculus of Constructions [28]. This language has the following syntax

$$\begin{array}{rrcl}
(\textit{kinds}) & K & ::= & * \mid \Pi\alpha{:}K.\, K' \mid \Pi x{:}A.\, K \\
(\textit{types}) & A, B & ::= & \alpha \mid \Pi x{:}A.\, B \mid \forall\alpha{:}K.\, A \mid \lambda\alpha{:}K.\, A \mid A\ B \mid \lambda x{:}A.\, B \mid A\ N \\
(\textit{terms}) & M, N & ::= & x \mid \lambda x{:}A.\, M \mid M\ N \mid \lambda\alpha{:}K.\, M \mid M\ A \\
(\textit{contexts}) & \Gamma, \Delta & ::= & \varepsilon \mid \Gamma, x{:}A \mid \Gamma, \alpha{:}K
\end{array}$$

and Figures 2.6 and 2.7 presents the type system.

The Calculus of Constructions (CC) corresponds to intuitionistic higher-order logic. As such, it is extremely expressive, including as features, impredicative polymorphism, type constructors, and dependent types. Due to this assortment of features, the system as presented in Figures 2.6 and 2.7 is quite large. In the next section we will see that the system can be much more compactly presented by identifying a common pattern that occurs several times over in the language.

The Calculus of Constructions provides the extra expressiveness necessary to extend our generic list type so that it is indexed by the list length. After doing so, the following typing relationships should hold (after desugaring the syntax for lists and naturals):

$$\begin{array}{rcl}
[\,] & : & \textit{List Nat } 0 \\
[4] & : & \textit{List Nat } 1 \\
[5, 3] & : & \textit{List Nat } 2
\end{array}$$

Well-formed kinds $\boxed{\Gamma \vdash K \text{ kind}}$

$$\textsc{Star}\ \frac{}{\Gamma \vdash \star \text{ kind}} \qquad \Pi\text{-}\textsc{Form}\ \frac{\Gamma \vdash K \text{ kind} \qquad \Gamma, \alpha{:}K \vdash K' \text{ kind}}{\Gamma \vdash \Pi\alpha{:}K.\, K' \text{ kind}} \qquad \Pi\text{-}\textsc{Form1}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x{:}A \vdash K \text{ kind}}{\Gamma \vdash \Pi x{:}A.\, K \text{ kind}}$$

Well-formed types $\boxed{\Gamma \vdash A : K}$

$$\textsc{TyVar}\ \frac{\alpha{:}K \in \Gamma}{\Gamma \vdash \alpha : K} \qquad \Pi\text{-}\textsc{Form2}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x{:}A \vdash B : \star}{\Gamma \vdash \Pi x{:}A.\, B : \star} \qquad \forall\text{-}\textsc{Form}\ \frac{\Gamma \vdash K \text{ kind} \qquad \Gamma, \alpha{:}K \vdash A : \star}{\Gamma \vdash \forall\alpha{:}K.\, A : \star}$$

$$\Pi\text{-}\textsc{Intro}\ \frac{\Gamma \vdash K \text{ kind} \qquad \Gamma, \alpha{:}K \vdash A : K'}{\Gamma \vdash \lambda\alpha{:}K.\, A : \Pi\alpha{:}K.\, K'} \qquad \Pi\text{-}\textsc{Elim}\ \frac{\Gamma \vdash A : \Pi\alpha{:}K.\, K' \qquad \Gamma \vdash B : K}{\Gamma \vdash A\ B : K'[B/\alpha]}$$

$$\Pi\text{-}\textsc{Intro1}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x{:}A \vdash B : K}{\Gamma \vdash \lambda x{:}A.\, B : \Pi x{:}A.\, K} \qquad \Pi\text{-}\textsc{Elim1}\ \frac{\Gamma \vdash A : \Pi x{:}A.\, K \qquad \Gamma \vdash N : A}{\Gamma \vdash A\ N : K[N/x]}$$

$$\textsc{Conv1}\ \frac{\Gamma \vdash A : K \qquad \Gamma \vdash K' \text{ kind} \qquad K =_\beta K'}{\Gamma \vdash A : K'}$$

Figure 2.6: The Calculus of Constructions. Rules for well-formed kinds and types

Well-formed terms $\boxed{\Gamma \vdash M : A}$

$$\textsc{Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \Pi\textsc{-Intro2}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : \Pi x{:}A.\, B} \qquad \Pi\textsc{-Elim2}\ \frac{\Gamma \vdash M : \Pi x{:}A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}$$

$$\forall\textsc{-Intro}\ \frac{\Gamma \vdash K\ \mathsf{kind} \qquad \Gamma, \alpha{:}K \vdash M : B}{\Gamma \vdash \lambda \alpha{:}K.\, M : \forall \alpha{:}K.\, B} \qquad \forall\textsc{-Elim}\ \frac{\Gamma \vdash M : \forall \alpha{:}K.\, B \qquad \Gamma \vdash A : K}{\Gamma \vdash M\ A : B[A/\alpha]}$$

$$\textsc{Conv2}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \star \qquad A =_\beta B}{\Gamma \vdash M : B}$$

Figure 2.7: The Calculus of Constructions. Rules for Well-formed terms.

With this goal in mind, we redefine the *List* type as follows:

$$\begin{aligned} List \;=\; & \lambda \beta{:}\star.\, \lambda n{:}Nat. \\ & \forall \alpha{:}Nat \to \star.\, (\Pi m{:}Nat.\, \beta \to \alpha\, m \to \alpha\, (succ\, m)) \to \alpha\, zero \to \alpha\, n \end{aligned}$$

Now we may define length-aware versions of the previous operations.

$$nil \;:\; \forall \beta{:}\star.\, List\ \beta\ zero$$
$$nil = \lambda \beta{:}\star.\, \lambda \alpha{:}Nat \to \star.\, \lambda c{:}(\Pi m{:}Nat.\, \beta \to \alpha\, m \to \alpha\, (succ\, m)).\, \lambda e{:}\alpha\ zero.\, e$$

$$cons \;:\; \forall \beta{:}\star.\, \Pi n{:}Nat.\, \beta \to List\ \beta\ n \to List\ \beta\ (succ\ n)$$
$$\begin{aligned} cons = \lambda \beta{:}\star.\, & \lambda n{:}Nat.\, \lambda x{:}\beta.\, \lambda xs{:}List\ \beta\ n. \\ & \lambda \alpha{:}Nat \to \star.\, \lambda c{:}(\Pi m{:}Nat.\, \beta \to \alpha\, m \to \alpha\, (succ\, m)).\, \lambda e{:}\alpha\ zero. \\ & \quad c\ n\ x\ (xs\ \alpha\ c\ e) \end{aligned}$$

$$
\begin{aligned}
&\mathit{map} \;:\; \forall\beta,\gamma{:}\star.\,\Pi n{:}\mathit{Nat}.\,(\beta \to \gamma) \to \mathit{List}\;\beta\;n \to \mathit{List}\;\gamma\;n\\
&\mathit{map} = \lambda\beta,\gamma{:}\star.\,\lambda n{:}\mathit{Nat}.\,\lambda f{:}\beta \to \gamma.\,\lambda xs{:}\mathit{List}\;\beta\;n.\\
&\qquad\qquad \lambda\alpha{:}\mathit{Nat} \to \star.\,\lambda c{:}(\Pi m{:}\mathit{Nat}.\,\gamma \to \alpha\;m \to \alpha\;(\mathit{succ}\;m)).\,\lambda e{:}\alpha\;\mathit{zero}.\\
&\qquad\qquad\quad xs\;\alpha\;(\lambda m{:}\mathit{Nat}.\,\lambda h{:}\beta.\,\lambda t{:}\alpha.\,c\;m\;(f\;h)\;t)\;e
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{append} \;:\; \forall\beta{:}\star.\,\Pi n,m{:}\mathit{Nat}.\,\mathit{List}\;\beta\;n \to \mathit{List}\;\beta\;m \to \mathit{List}\;\beta\;(\mathit{plus}\;n\;m)\\
&\mathit{append} = \lambda\beta{:}\star.\,\lambda n,m{:}\mathit{Nat}.\,\lambda xs{:}\mathit{List}\;\beta\;n.\,\lambda ys{:}\mathit{List}\;\beta\;m.\\
&\qquad\qquad \lambda\alpha{:}\mathit{Nat} \to \star.\,\lambda c{:}(\Pi l{:}\mathit{Nat}.\,\beta \to \alpha\;l \to \alpha\;(\mathit{succ}\;l)).\,\lambda e{:}\alpha\;\mathit{zero}.\\
&\qquad\qquad\quad xs\;(\lambda n{:}\mathit{Nat}.\,\alpha\;(\mathit{plus}\;n\;m))\\
&\qquad\qquad\qquad (\lambda l{:}\mathit{Nat}.\,\lambda x{:}\beta.\,\lambda xs{:}\alpha\;(\mathit{plus}\;l\;m).\,c\;(\mathit{plus}\;l\;m)\;x\;xs)\\
&\qquad\qquad\qquad (ys\;\alpha\;c\;e)
\end{aligned}
$$

Note: In order to type-check *append*, it must be the case that *plus zero m* is definitionally equal to m and that *plus* (*succ l*) m is definitionally equal to *succ* (*plus l m*). The former condition requires that the notion of definitional equality include η-conversion.

2.8 BARENDREGT'S λ-CUBE

The typing rules for the Calculus of Constructions can can be organized in two different ways. In Figures 2.6 and 2.7 they are organized by syntactic categories: where they fit into the typing hierarchy. However, we might also organize them so that rules with similar structures are grouped together.

STAR	VAR, TYVAR	CONV1, CONV2
Π-FORM, Π-FORM1, Π-FORM2, $\forall$-FORM	Π-INTRO, Π-INTRO1, Π-INTRO2, $\forall$-INTRO	Π-ELIM, Π-ELIM1, Π-ELIM2, $\forall$-ELIM

The following changes serve to highlight the similarities of typing rules within each of these boxes.

1. Merge the syntactic categories of terms, types, and kinds into a single syntactic category (called terms). Also merge the syntactic categories of term and type variables.

2. Write Π to indicate parametric polymorphism instead of $\forall$.

3. Introduce a new symbol $\Box$ to name the type of all kinds and accordingly replace the judgment $\Gamma \vdash K$ kind with $\Gamma \vdash K : \Box$

4. Introduce a special syntactic category of *sorts*, for special symbols that act as the "type" of an entire class of "types". Include $\star$ and $\Box$ in this category.

After making these changes, each of the above mentioned boxes of similar typing rules collapses down to a single rule (shown in Figure 2.8). In the case of the box of Π-Intro rules, we index the resulting rule by the set

$$\mathcal{R} = \{(\star, \star), (\Box, \star), (\Box, \Box), (\star, \Box)\}.$$

Each element of $\mathcal{R}$ allows a particular form of parameterization and corresponds to a distinct feature of the Calculus of Constructions: The element $(\star, \star) \in \mathcal{R}$ allows us to parameterize terms over terms (i.e., to form functions)[4]; The element $(\Box, \star) \in \mathcal{R}$ allows us to parameterize terms over types as in System F (i.e., the feature of parametric polymorphism); The element $(\Box, \Box) \in \mathcal{R}$ allows us to parameterize types over types as in System F^{ω} (i.e., the feature of type constructors); The element $(\star, \Box) \in \mathcal{R}$ allows us to parameterize types over terms as in LF. (i.e., the feature of dependent types).

[4]Here, we mean terms in the original Calculus of Constructions sense, as opposed to types and kinds.

Syntax

$$\begin{array}{rrcl} (\textit{sorts}) & s & ::= & * \mid \Box \\ (\textit{terms}) & M, N, A, B, K & ::= & x \mid \lambda x{:}A.\, M \mid M\ N \mid \Pi x{:}A.\, B \mid s \\ (\textit{contexts}) & \Gamma, \Delta & ::= & \varepsilon \mid \Gamma, x{:}A \end{array}$$

Typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{Star}\quad \frac{}{\Gamma \vdash \star : \Box} \qquad \textsc{Var}\quad \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \Pi\textsc{-Form}\quad \frac{(s, s') \in \mathcal{R} \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s'}{\Gamma \vdash \Pi x{:}A.\, B : s'}$$

$$\Pi\textsc{-Intro}\quad \frac{\Gamma \vdash A : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : \Pi x{:}A.\, B} \qquad \Pi\textsc{-Elim}\quad \frac{\Gamma \vdash M : \Pi x{:}A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}$$

$$\textsc{Conv}\quad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash M : B}$$

Possible rules: $\mathcal{R} \subseteq \{(\star, \star), (\Box, \star), (\Box, \Box), (\star, \Box)\}$ and $(\star, \star) \in \mathcal{R}$

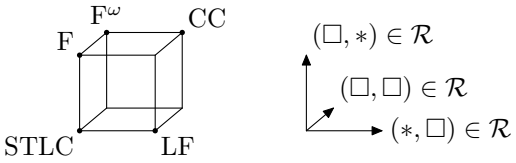


Figure 2.8: The λ-cube. A family of eight typed λ-calculi parameterized by $\mathcal{R}$.

With the exception of $(\star, \star)$, these elements of $\mathcal{R}$ may be removed in any combination to restrict the features of the language. In fact, each previous typed λ-calculus in this chapter results from a particular selection of these features, as shown in the following table.

	$(\star, \star)$	$(\Box, \star)$	$(\Box, \Box)$	$(\star, \Box)$
STLC	✓			
System F	✓	✓		
System F$^\omega$	✓	✓	✓	
LF	✓			✓
CC	✓	✓	✓	✓

Because the decisions to keep or drop elements $(\Box, \star)$, $(\Box, \Box)$, and $(\star, \Box)$ from $\mathcal{R}$ can be made independently, we can coordinatize the resulting languages along three dimensions and map them to the corners of a cube. For this reason, the resulting family of λ-calculi is known as the λ-cube. The full definition for all the systems of the λ-cube may be found in Figure 2.8. Each corner corresponds to a language studied in the literature (see Barendregt [4] for a full bibliography).

Though we have collapsed all syntactic categories into a single category called terms, we informally use metavariables M and N differently than A and B and K. We use metavariables A and B when we want to emphasize that a particular λ-cube term is a *type*, in the sense that it may classify other expressions. We only say "may" because it is possible that a type have no inhabitants, just as it is possible that a proposition have no proofs. For example, in an empty context, false propositions have no inhabitants. The next section will define this new notion of type much more precisely.

Note that the new notion of type is more general than before — what we previously called either types or kinds, we now call types. Unless otherwise indicated, we will now say "CC types" and "CC kinds" to indicate the prior concepts. We tend to use the metavariable K to indicate a CC kind. The word "term" has a

similar capacity for confusion. For the prior notion, we will say "CC term".

While discussing terminology, we note that the term *expression* will be (and has been) used to refer to any bit of syntax, no matter in what syntactic category it happens to belong. In the λ-cube or PTS, "expression" is coextensive with "term" because there is but one syntactic category.

2.9 PURE TYPE SYSTEMS

Pure type systems are a natural generalization of the λ-cube. Instead of parameterizing only the Π-Intro typing rule, we parameterize everything in the language having to do with sorts.

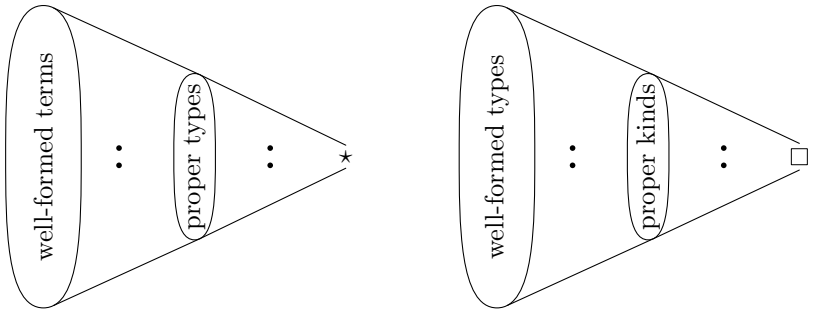
2.9.1 Specifications

Pure Type Systems are a family of typed λ-calculi. Each member of this family is identified by a *specification* consisting of a set $\mathcal{S}$ of *sorts* (a.k.a. *universes*), a set $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ of *axioms*, and a set $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ of *rules*. We discuss the role of each specification component in turn.

In the Calculus of Constructions, both (proper) types and kinds act like types in the more general sense that they classify other expressions in the language. How might we characterize this more general notion of types? In the λ-cube, CC types and CC kinds are distinguished not syntactically, but by the type system: If $\Gamma \vdash M : \star$ holds, then M is a CC type; If $\Gamma \vdash M : \Box$ holds, then M is a CC kind. In this way sorts in the λ-cube serve to name different *universes* of types. For this reason sorts are sometimes called universes. This idea is made more clear if we give sorts the same name as the universe they represent. For example, if we rename $\star$ as type and $\Box$ as kind, then the judgments $\Gamma \vdash M : \mathsf{type}$ and $\Gamma \vdash M : \mathsf{kind}$ become much clearer.

Because sorts are types of types, each sort sits atop a three-level structure in

the λ-cube. The sort $\star$ classifies all proper CC types, which in turn classify all well-formed CC terms. Similarly, the sort $\Box$ classifies all proper CC kinds[5], which in turn classify all CC types. Therefore the following pattern arises.



Every well-formed expression in the Calculus of Constructions belongs somewhere in this figure.

In PTS, we generalize from $\{\star, \Box\}$ to an arbitrary set of sorts $\mathcal{S}$. However, the following theorem of PTS shows that the pattern observed above holds for PTS as well.

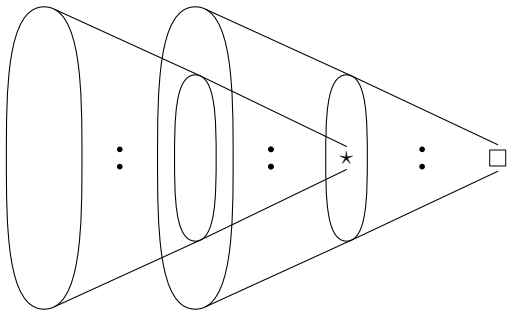
Theorem 2.9.1 (Coherence)

If $\Gamma \vdash M : A$, *then either* $A = s$ *or* $\Gamma \vdash A : s$ *for some sort* $s \in \mathcal{S}$.

This theorem says that every PTS-type is either a sort or belongs to the universe named by a sort. In fact, we take this to be the definition of a type in PTS.

The next component of a PTS specification is the set $\mathcal{A}$ of axioms. Axioms specify the typing relationship between sorts and thereby place type universes into a hierarchical relationship. For example, in the λ-cube, we have the single axiom $\star : \Box$, stating that the sort $\star$ is a kind. This axiom orders the universes of CC types and CC kinds as follows:

[5]The term "proper kinds" is redundant because there are no kind constructors in CC. However, the designation is accurate.



Note the induced overlap since proper types are a (proper) subset of well-formed types.

In PTS, we generalize from the single axiom $\star : \Box$ to a set of axioms $\mathcal{A}$. Each element $(s_1, s_2) \in \mathcal{A}$ corresponds to an axiom $s_1 : s_2$. In this way, the hierarchical structure of type universes in a particular PTS is determined by its specification.

The third component of a PTS specification is the set $\mathcal{R}$ of rules. Rules enumerate the permitted forms of dependency between various expressions in the language in terms of the universes to which their types belong. We have already seen how every possible form of dependence between CC terms and CC types corresponds to a particular rule in the λ-cube specification $\mathcal{R}$. For example, the dependence of a CC type of kind K on a CC term of type A is permitted because the rule $(\star, \Box) \in \mathcal{R}$ allows us to specialize the λ-cube typing rule Π-Form as follows:

$$\Pi\text{-Form}\quad \frac{(s, s') \in \mathcal{R} \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s'}{\Gamma \vdash \Pi x{:}A.\, B : s'} \quad \mapsto \quad \frac{\Gamma \vdash A : \star \qquad \Gamma \vdash K : \Box}{\Gamma \vdash \Pi x{:}A.\, K : \Box}$$

When we move to PTS, the Π-Form rule is similarly parameterized by a set $\mathcal{R}$ of rules, but rules are triples instead of pairs.

$$\Pi\text{-Form}\quad \frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.\, B : s_3}$$

The reason for using triples is the following. When we talk about one entity depending on another, there are actually three expressions involved: the expression

M denoting the dependent entity, the expression x denoting the entity upon which the first depends, and the expression $\lambda x{:}A.\, M$ that witnesses the dependence. In general, the types of these three expressions may belong to three different universes. Section 2.10 demonstrates languages leveraging this extra measure of flexibility. However, since many pure type systems of interest have the property that $s_2 = s_3$ for all $(s_1, s_2, s_3) \in \mathcal{R}$, it is common to abuse notation and write $(s, s') \in \mathcal{R}$ as an abbreviation for $(s, s', s') \in \mathcal{R}$ when discussing such a system.

2.9.2 Syntax

The syntax of PTS terms and typing contexts is as follows:

$$\begin{array}{lrcl} (\textit{terms}) & M, N, A, B & ::= & x \mid \lambda x{:}A.\, M \mid M\ N \mid \Pi x{:}A.\, B \mid s \\ (\textit{contexts}) & \Gamma, \Delta & ::= & \varepsilon \mid \Gamma, x{:}A \end{array}$$

The metavariable x stands for a program variable and the metavariable s stands for a sort in $\mathcal{S}$. As in the λ-cube, there is no distinct syntactic category for types.

The term $\Pi x{:}A.\, B$ is a function type with domain A and codomain $B(x)$ where x names the value to which the function is ultimately applied. In this way, the return type of a function may depend on the *value* of the actual parameter. When $x \notin FV(B)$ the type $\Pi x{:}A.\, B$ indicates a regular (non-dependent) function space and may be abbreviated as $A \to B$.

Though we haven't mentioned it previously, we should note here that typing contexts Γ are ordered *sequences* of bindings rather than *sets* of bindings. The order in such a sequence matters, because variables bound in the sequence may appear in types that occur later in the sequence (to the right). For this reason, it may not make sense to re-order the sequence of bindings lest some occurrence of a variable move to a position preceding its binding. Again, we require that all variables bound in a typing context are distinct.

$\boxed{\Gamma \vdash M : A}$

$$\textsc{Axiom}\ \frac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 : s_2} \qquad \textsc{Var}\ \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \qquad \textsc{Weak}\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash M : B}{\Gamma, x{:}A \vdash M : B}$$

$$\Pi\text{-}\textsc{Form}\ \frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.\, B : s_3} \qquad \Pi\text{-}\textsc{Intro}\ \frac{\Gamma \vdash \Pi x{:}A.\, B : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : \Pi x{:}A.\, B}$$

$$\Pi\text{-}\textsc{Elim}\ \frac{\Gamma \vdash M : \Pi x{:}A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash M : B}$$

Figure 2.9: Typing rules for PTS

2.9.3 Typing Rules

The typing rules for PTS are shown in Figure 2.9. The $\mathcal{A}$ and $\mathcal{R}$ components of the specification determine the typing relationship between sorts (rule Axiom) and the permitted forms of dependency in the language (rule Π-Form).

Because Pure Type Systems are the formalism upon which the work presented in this dissertation is based, we will spend some time describing each of its typing rules

- Axiom: This rule states the typing relationship between sorts $s \in \mathcal{S}$. Axiom is actually a family of rules, one for each pair $(s_1, s_2) \in \mathcal{A}$, where $\mathcal{A}$ is part of the PTS specification.

- VAR: This rule says that a variable has exactly the type that the typing context says it does. The premise ensures that the type of the context entry is well-formed as a type.

- WEAK: This rule says we need not use all variables in the typing context. Again, it must be the case that the type of the context entry is well-formed as a type.

- Π-FORM: This rule states the well-formedness conditions for Π-types. As Π-types classify λ-abstractions, this rule determines the allowable forms of dependence in the language (i.e., what types of things may be abstracted over what other types of things). As with AXIOM, this is actually a family of typing rules, one for each triple $(s_1, s_2, s_3) \in \mathcal{R}$ where $\mathcal{R}$ is part of the PTS specification.

- Π-INTRO: This rule says when a functional abstraction $\lambda x{:}A.\, M$ is well-formed — whenever one can show that M has type B under the additional assumption that x has type A. Furthermore, the type $\Pi x{:}A.\, B$ of the abstraction must be a well-formed type. This is how the rule Π-FORM (indirectly) determines the allowable forms of λ-abstractions.

- Π-ELIM: This rule gives the type of a function application. The function must have a function type $\Pi x{:}A.\, B$ and the argument must have the same type as the function domain A. Note, however, that the return type of the application is not simply B, but rather $B[N/x]$. This is because PTS is a family of *dependently typed* calculi by default. The return type of a function may depend on the argument to which that function is applied.

- CONV: The *conversion* rule determines the language's notion of equality between types. As we saw in rule Π-ELIM, arbitrary terms may be lifted up into types. In general, therefore, comparing two types for equality requires

comparing arbitrary terms for equality. The notion of equality used is β-conversion. For β-conversion to be decidable, it is necessary that well-typed terms are strongly normalizing. Because no evidence is required of the programmer indicating why A and B are equal, we say that they are equal *by definition*. For this reason, the notion of equality (β-equality in this case) used in the conversion rule is called *definitional equality*.

In many type systems for λ-calculi, it is assumed in the typing rules that typing contexts are well-formed. However, in PTS, this requirement is made explicit in that rules that inspect the typing context (VAR and WEAK) ensure that each type A in the typing context is well-formed as a type in the preceding portion Γ of the typing context (i.e., $\Gamma \vdash A : s$ for some s).

Another difference between these typing rules and those of the λ-cube is that the VAR rule of the λ-cube is split into the VAR and WEAK rules of PTS. This change helps emphasize the sequential nature of typing contexts.

2.10 PURE TYPE SYSTEM EXAMPLES

In this section, we discuss several examples of Pure Type Systems.

2.10.1 Systems in the λ-cube

As expected, all eight calculi in the λ-cube are examples of Pure Type Systems. Each one has a specification with $\mathcal{S} = \{\star, \Box\}$ and $\mathcal{A} = \{(\star, \Box)\}$. They differ only with respect to the $\mathcal{R}$ component of the specifications. Each one has a specification $\mathcal{R} \subseteq \{(\star,\star), (\Box,\star), (\Box,\Box), (\star,\Box)\}$ such that $(\star,\star) \in \mathcal{R}$.

That such a wide variety of typed λ-calculi appear as special cases of the PTS formalism underscores the expressiveness of the formalism.

2.10.2 Hindley-Milner Polymorphism

So far we have not discussed any PTS in which $s_2 \neq s_3$ for some $(s_1, s_2, s_3) \in \mathcal{R}$. This section introduces one such example: a PTS with the same expressiveness as Hindley-Milner polymorphism, due to Barthe and Coquand [6], who discuss several other PTS examples.

Statically typed functional languages feature a form of parametric polymorphism along the lines of the System F. However, these languages also support type inference so that the programmer need not write the type of every single variable. Polymorphism is largely implicit in such languages, meaning that explicit type-abstractions and type-applications need not be written out as they are in System F.

However, inferring types as well as type-abstractions and type-applications for System F is undecidable [95], so how can these languages provide implicit polymorphism? The answer is that polymorphic types are restricted to a particular form, namely $\forall\alpha_1 \ldots \alpha_n.\, B$ where B contains no $\forall$. This restricted form of polymorphic type is called a *scheme* and is given by the following grammar:

$$\begin{array}{rrcl} (\textit{schemes}) & \sigma & ::= & \forall\alpha.\, \sigma \mid \tau \\ (\textit{types}) & \tau & ::= & \alpha \mid \tau \to \tau' \end{array}$$

Therefore, in the Hindley-Milner PTS we have sorts for types ($\star$) and schemes ($\triangle$) as well as the usual one for kinds ($\Box$).

$$\mathcal{S} = \{\star, \triangle, \Box\}$$

Both types and schemes are classified by kinds.

$$\mathcal{A} = \{(\star, \Box), (\triangle, \Box)\}$$

In order to capture in a PTS specification the restriction on occurrences of $\forall$, we distinguish between three types of "function space" that cover all possible cases.

- $(\tau \to \tau')$ the abstraction of one monomorphic entity over another to form a third
- $(\forall\alpha.\,\tau)$ the abstraction of a monomorphic entity over a type to form a polymorphic entity
- $(\forall\alpha.\,\sigma)$ the abstraction of a polymorphic entity over a type to form a polymorphic entity

The Π-formation rules simply follow this enumeration. (As promised, $s_2 \neq s_3$ in the middle rule.)

$$\mathcal{R} = \{(\star,\star,\star),(\Box,\star,\triangle),(\Box,\triangle,\triangle)\}$$

Let $(\mathcal{S}_{HM},\mathcal{A}_{HM},\mathcal{R}_{HM})$ and $(\mathcal{S}_F,\mathcal{A}_F,\mathcal{R}_F)$ be the PTS specifications for the Hindley-Milner PTS and the System F PTS, respectively. Then the following mapping from $\mathcal{S}_{HM}$ to $\mathcal{S}_F$ extends to an embedding of the Hindley-Milner PTS into the System F PTS. (Notice how the mapping sends $\mathcal{A}_{HM}$ to $\mathcal{A}_F$ and $\mathcal{R}_{HM}$ to $\mathcal{R}_F$.)

	HM		F
$\mathcal{S}$:	$\Box$	$\mapsto$	$\Box$
	$\star,\triangle$	$\mapsto$	$\star$
$\mathcal{A}$:	$(\star,\Box),(\triangle,\Box)$	$\mapsto$	$(\star,\Box)$
$\mathcal{R}$:	$(\star,\star,\star)$	$\mapsto$	$(\star,\star)$
	$(\Box,\star,\triangle),(\Box,\triangle,\triangle)$	$\mapsto$	$(\Box,\star)$

The rules for System F use the previously-mentioned abbreviation, so that, for example, $(\star,\star) \in \mathcal{R}_F$ really means $(\star,\star,\star) \in \mathcal{R}_F$.

Perhaps as important as the rules that appear in $\mathcal{R}_{HM}$ are two rules that *do not* appear, namely $(\Box,\star,\star)$ and $(\Box,\triangle,\star)$. These rules would allow one to form a "monomorphic term" by abstracting a (monomorphic or polymorphic) term over a type, thereby directly contradicting the meaning of "monomorphic".

2.10.3 Extended Calculus of Constructions

A particularly expressive type theory, called the Extended Calculus of Constructions (ECC), was developed by Zhaohui Luo in his Ph.D. thesis [53]. It adds to the Calculus of Constructions a *predicative hierarchy* of universes (sorts).

$$\Box_0 : \Box_1 : \Box_2 : \cdots \qquad (\text{where } \Box = \Box_0)$$

In this hierarchy, we may form Π-types as long as they are not of the form $A = \Pi x{:}\Box_i.\, B$ where $A : \Box_i$, as this would indicate impredicativity.

The core of ECC can be cast as a PTS with the following specification:

$$\mathcal{S} = \{\star\} \cup \{\Box_i \mid i \in \mathbb{N}\} \qquad \mathcal{A} = \{(\star, \Box_i) \mid i \in \mathbb{N}\} \cup \{(\Box_i, \Box_j) \mid i, j \in \mathbb{N} \wedge i < j\}$$

$$\mathcal{R} = \{(s, \star, s') \mid s, s' \in \mathcal{S}\} \cup \{(\star, \Box_j, \Box_k) \mid j, k \in \mathbb{N} \wedge j \leq k\}$$
$$\cup\, \{(\Box_i, \Box_j, \Box_k) \mid i, j, k \in \mathbb{N} \wedge i, j \leq k\}$$

While this notation is suggestive of the Calculus of Constructions, a more compact presentation of the same specification is possible.

$$\mathcal{S} = \mathbb{N} \qquad \mathcal{A} = \{(i, j) \mid i < j\} \qquad \mathcal{R} = \{(i, 0, 0) \mid i \in \mathbb{N}\} \cup \{(i, j, k) \mid i, j \leq k\}$$

The PTS formalism is not expressive enough to include some features of ECC. In particular, ECC includes a notion of subtyping called *full cumulativity* whereby each type universe is a subtype of all higher universes (i.e., $\star \subseteq \Box_j$ and $\Box_i \subseteq \Box_j$ whenever $i < j$). Our PTS specification approximates full cumulativity in the $\mathcal{A}$ and $\mathcal{R}$ components. ECC also supports strong Σ-types in the predicative hierarchy (Σ is to existential quantification and pairs as Π is to universal quantification and functions).

Chapter 3

ERASURE SEMANTICS

In this chapter and the next, we develop an erasure semantics for Pure Type Systems consisting of two type-respecting translations: (1) a program analysis that introduces erasure annotations, and (2) a type-respecting erasure translation that is guided by these annotations. We prove that our program analysis is correct and optimal in the sense of marking as much of a program for erasure as possible, and that our erasure translation is meaning preserving and removes computational overhead.

Our approach is based upon an extrinsic view of computational irrelevance. Type theoretically, our approach amounts to a distinction between non-computational and computational function spaces. This approach eliminates the code duplication problem inherent in previous approaches to combining dependent types and erasure semantics that we discussed in Section 1.4.

We will see that the meaning of the non-computational function space is a highly generic form of parametric polymorphism. The target language of the erasure translation is Alexandre Miquel's Implicit Pure Type Systems, which includes a $\forall$ type-former indicating implicit parametric polymorphism. Our erasure translation maps the non-computational function space to Miquel's $\forall$.

3.1 ERASURE PURE TYPE SYSTEMS

At the heart of our approach to erasure semantics lies the framework of Erasure Pure Type Systems (EPTS), an extension of Pure Type Systems (PTS) with anno-

tations indicating computationally irrelevant parts of a program. EPTS is one of the contributions of this dissertation. The EPTS typing rules enforce a phase distinction between computationally relevant and irrelevant portions of the program, guaranteeing that the former do not depend computationally on the latter.

Later we will define an erasure translation that strips out the parts of a program marked as computationally irrelevant. The phase distinction in EPTS guarantees that the erasure translation produces meaningful programs.

3.1.1 Syntax

The syntax of EPTS is that of PTS with erasure annotations added.

$$\begin{aligned} (\textit{terms}) \quad & M, N, A, B ::= x \mid \lambda^{\tau} x{:}A.\, M \mid M @^{\tau} N \mid \Pi^{\tau} x{:}A.\, B \mid s \\ (\textit{contexts}) \quad & \Gamma, \Delta ::= \varepsilon \mid \Gamma, x{:}^{\tau} A \\ (\textit{annotations}) \quad & \tau ::= \mathsf{c} \mid \mathsf{r} \end{aligned}$$

The metavariable τ ranges over erasure annotations. The annotation r means "run-time". Syntax with this annotation behaves just as it would in PTS without any annotation. The annotation c means "compile-time" and indicates erasable portions of a program. The notions of run-time and compile-time are used because, in the erasure semantics, the c-marked portions are needed only at compile-time in order to type-check the program and will be erased afterwards. Only the r-marked portions will survive erasure and exist at run-time.

Every Π, λ, and @ is annotated with a τ. The annotation on a Π type distinguishes between two forms of function space.

- $\Pi^{\mathsf{r}} x{:}A.\, B$ is the type of functions whose body *may* depend computationally on the parameter x.
- $\Pi^{\mathsf{c}} x{:}A.\, B$ is the type of functions whose body *does not* depend computationally on the parameter x. In other words, the parameter x is computationally irrelevant in the function body.

Again, the abbreviation $A \xrightarrow{\tau} B$ stands for $\Pi^{\tau} x{:}A.\, B$ where x does not occur free in B. Similarly, the annotation τ of a λ-abstraction $\lambda^{\tau} x{:}A.\, M$ indicates the computational relevance of the formal parameter x in the body M and the annotation τ of an application $M@^{\tau} N$ indicates the computational relevance of the actual parameter N. These annotations guide the erasure translation to be defined in Section 3.3.

3.1.2 Type System

The type system for EPTS enforces two sorts of invariants.

- **Type-correctness.** The underlying PTS term, obtained by ignoring all erasure annotations, is well-formed.
- **Phase-correctness.** A phase distinction is maintained between compile-time and run-time entities in the term, whereby the latter may not depend computationally on the former.

We achieve the type-correctness invariant by simply annotating the PTS typing rules to obtain EPTS typing rules. This ensures that the underlying type structure of EPTS is the same as that of PTS. Phase-correctness is our chief concern in this chapter, though it will not be completely formalized and proved until Section 3.3.

Figure 3.1 contains the typing rules for EPTS. The typing judgment $\Gamma \vdash M :^{\tau} A$ is indexed by an erasure annotation τ indicating its *mode*. The r-mode judgment $\Gamma \vdash M :^{\mathsf{r}} A$ says that M is a well-formed run-time entity, while the c-mode judgment $\Gamma \vdash M :^{\mathsf{c}} A$ says that M is a well-formed compile-time entity. Similarly, we saw in the previous section outlining the syntax of EPTS that context entries $x{:}^{\tau} A$ are also annotated, indicating whether x is a run-time or a compile-time entity.

When discussing the components of a typing rule, the judgments above the line are called its *premises* and the judgment below the line its *conclusion*. In a

$\boxed{\Gamma \vdash M :^{\tau} A}$

$$\textsc{Axiom}\ \frac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2} \qquad \textsc{Var}\ \frac{\Gamma \vdash A :^{\mathsf{c}} s}{\Gamma, x{:}^{\mathsf{r}} A \vdash x :^{\mathsf{r}} A} \qquad \textsc{Weak}\ \frac{\Gamma \vdash A :^{\mathsf{c}} s \qquad \Gamma \vdash M :^{\mathsf{r}} B}{\Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B}$$

$$\Pi\text{-}\textsc{Form}\ \frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} s_3}$$

$$\Pi\text{-}\textsc{Intro}\ \frac{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s \qquad \Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B}{\Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B} \qquad \Pi\text{-}\textsc{Elim}\ \frac{\Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B \qquad \Gamma \vdash N :^{\tau} A}{\Gamma \vdash M @^{\tau} N :^{\mathsf{r}} B[N/x]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash M :^{\mathsf{r}} A \qquad \Gamma \vdash B :^{\mathsf{c}} s \qquad A =_{\beta} B}{\Gamma \vdash M :^{\mathsf{r}} B} \qquad \textsc{Reset}\ \frac{\Gamma^{\circ} \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\mathsf{c}} A}$$

Figure 3.1: Typing rules for EPTS

judgment of the form $\Gamma \vdash M :^{\tau} A$, we call Γ its *typing context*, τ its *mode*, M its *subject*, and A its *object*.

To enforce phase-correctness of EPTS terms, the type system must ensure that all λ-binders and @-arguments marked with c are erasable. Recall from Section 1.4.2 that erasability of λ-binders and @-arguments in the λ/@ graph must be considered one connected component at a time. The flow analysis implicit in the typing rules ensures that each λ and @ in a particular connected component is annotated with the same τ. Therefore, if every λ^{c}-binder is a dummy binder, then every $@^{\mathsf{c}}$-argument is erasable. So we need only check that for each abstraction $\lambda^{\mathsf{c}} x{:}A.\, M$ in the program, all free occurrences of x in M must appear either inside a type annotation or inside an $@^{\mathsf{c}}$-argument (i.e., inside N' in an application $N @^{\mathsf{c}} N'$).

The typing rules enforce this invariant using the following technique, which we learned from Pfenning [73], who credits Momigliano [66] with a similar idea.

1. Each λ^{c}-bound variable x is flagged as a compile-time entity when it is added to the typing context (see the Π-Intro rule when τ is instantiated to c).

2. However, we require that the x is flagged as a run-time entity whenever we reach an occurrence of x (see rule Var where the context entry must be r).

3. To overcome this mismatch for occurrences of x in positions that are computationally irrelevant with respect to the overall λ-abstraction, this flag is then locally reset (so that x is considered a run-time entity) whenever we check a type annotation or $@^{\mathsf{c}}$-argument (see both the Π-Elim rule when τ is instantiated to c and the Reset rule for typing in c-mode). The operation of locally resetting context entry annotations is defined as follows.

Definition 3.1.1 (Context Reset Operation) $\boxed{\Gamma^{\circ}}$

$$\varepsilon^{\circ} = \varepsilon \qquad\qquad (\Gamma, x{:}^{\tau} A)^{\circ} = \Gamma^{\circ}, x{:}^{r} A$$

This strategy ensures that all occurrences of λ^{c}-bound variables occur in positions marked for erasure, and therefore all λ^{c}-binders are dummy binders after we erase their bodies.

Figure 3.2 shows a simple example derivation exhibiting all the features of this strategy. Each λ-bound variable is initially annotated in the typing context with the same annotation as its λ-binder. In particular, the variable a is initially annotated in the typing context with annotation c. However, when we get down to typing the occurrences of a, rule Var requires that its context annotation be r. This tension is resolved by requiring all occurrences of a to happen *in a compile-time setting*. In this example, the λ-bound a occurs only inside the argument of a c-application (i.e., in a compile-time setting). Whenever we move into a compile-time setting, we switch the mode of the typing judgment to c. The Reset rule says how to type check in a compile-time setting: simply pretend that all compile-time assumptions in the typing context are run-time assumptions. In this example, the Reset rule changes the annotation on the context entry for a from c to r.

Principles of Computational Irrelevance The preceding discussion of the type system focused on the intended application of erasure, but more fundamental than the application of erasure is the notion by which it is justified in the first place, namely, computational irrelevance. We now introduce four principles of computational irrelevance and how they are reflected in the way that our type system handles erasure annotations.

1. *It is meaningful to compute the value of any term.* In a pure λ-calculus, computation is simply reduction, which may be carried out on any term. For this reason, every syntactic form in the language (even sorts and Π-types) appears as the subject of its own dedicated typing rule concluding in an r-mode typing judgment (Var, Π-Intro, Π-Elim, Π-Form, Axiom).

2. *A variable depends computationally on itself.* If the subterm we want to

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash f :^{\mathsf{r}} \Pi^{\mathsf{c}} a{:}*.\, a \xrightarrow{\mathsf{r}} a} \quad \dfrac{\dfrac{\vdots}{\Gamma^{\circ} \vdash a :^{\mathsf{r}} *}}{\Gamma \vdash a :^{\mathsf{c}} *}\,(\textsc{Reset})}{\Gamma \vdash f@^{\mathsf{c}} a :^{\mathsf{r}} a \xrightarrow{\mathsf{r}} a}\,(\Pi\text{-}\textsc{Elim}) \quad \dfrac{\vdots}{\Gamma \vdash x :^{\mathsf{r}} a}}{f{:}^{\mathsf{r}}\Pi^{\mathsf{c}} a{:}*.\, a \xrightarrow{\mathsf{r}} a, a{:}^{\mathsf{c}}*, x{:}^{\mathsf{r}} a \vdash f@^{\mathsf{c}} a@^{\mathsf{r}} x :^{\mathsf{r}} a}\,(\Pi\text{-}\textsc{Elim})}{f{:}^{\mathsf{r}}\Pi^{\mathsf{c}} a{:}*.\, a \xrightarrow{\mathsf{r}} a, a{:}^{\mathsf{c}}* \vdash \lambda^{\mathsf{r}} x{:}a.\, f@^{\mathsf{c}} a@^{\mathsf{r}} x :^{\mathsf{r}} a \xrightarrow{\mathsf{r}} a}\,(\Pi\text{-}\textsc{Intro})}{f{:}^{\mathsf{r}}\Pi^{\mathsf{c}} a{:}*.\, a \xrightarrow{\mathsf{r}} a \vdash \lambda^{\mathsf{c}} a{:}*.\, \lambda^{\mathsf{r}} x{:}a.\, f@^{\mathsf{c}} a@^{\mathsf{r}} x :^{\mathsf{r}} \Pi^{\mathsf{c}} a{:}*.\, a \xrightarrow{\mathsf{r}} a}\,(\Pi\text{-}\textsc{Intro})$$

where

$$\Gamma = f{:}^{\mathsf{r}}\Pi^{\mathsf{c}} a{:}*.\, a \xrightarrow{\mathsf{r}} a, a{:}^{\mathsf{c}}*, x{:}^{\mathsf{r}} a$$

and, therefore,

$$\Gamma^{\circ} = f{:}^{\mathsf{r}}\Pi^{\mathsf{c}} a{:}*.\, a \xrightarrow{\mathsf{r}} a, a{:}^{\mathsf{r}}*, x{:}^{\mathsf{r}} a$$

Figure 3.2: Fragment of a simple typing derivation in EPTS with the underlying PTS specification of System F. (To save space, we omit the first premise in all instances of the Π-Intro rule.)

compute consists solely of a variable, then we must be in a run-time context where that variable is bound either to a pre-computed value or to another expression whose value we can compute.

This principle is embodied in the typing rule VAR. In that rule, we conclude that x is a run-time entity (the concluding judgment is a r-mode judgment) under the assumption that x is a run-time entity (the context entry is annotated with r). The typing context in the typing judgment is a compile-time approximation to the ultimate run-time contexts in which x will be evaluated. Context entries annotated with r approximate actual value bindings at run-time, but those annotated with c have no run-time counterpart, because they exist only for type-checking purposes.

3. *No term depends computationally on its type.* Just as the typing context approximates the eventual run-time contexts in which a term may be evaluated, the type of a term approximates the value that it computes in one of those run-time contexts. Given this view of typing rules, the principle in question simply states that everything needed to compute the value of a term is found in the term itself and in the context in which it is evaluated. One need not foresee the value to which a term evaluates in order to compute that very same value.

   Several typing rules have a premise in c-mode rather than r-mode because of this principle. Rules VAR, WEAK, Π-INTRO, and CONV each have a premise of the form $\Gamma \vdash A :^{\mathsf{c}} s$. The purpose of each such premise is to check that A is well-formed *as a type* of some other entity in the rule. Because A occurs in the remainder of the rule only as the type of other variables or as the object (main type) of other judgments, we conclude that the subject of the rules' conclusion judgment does not depend computationally on A, and therefore we may type A in c-mode. In particular, note that the domain annotation

A of the λ-abstraction in the rule Π-Intro is considered as a compile-time entity.

4. *Computational relevance is relative.* As argued in Section 1.4, computational relevance is a relative notion. In other words, we should not ask in absolute terms whether a particular term is computationally relevant, but rather we should ask whether a subterm of a larger term M is computationally relevant with respect to (the task of computing the value of) M.

   While typing N under Γ as a subterm of M, the annotations on context entries in Γ indicate which of the variables that may appear in N will be assigned at run-time (after erasure) to values at the time computation of N begins. An entry $x{:}^{\mathsf{c}}A$ in Γ indicates that x will not be bound to a value at the time N is evaluated.

   This is the reason that Reset, the sole typing rule with a c-mode conclusion, is defined in terms of a single r-mode premise. One types N in a compile time setting by simply promoting all computationally irrelevant context entries (marked with c) to computationally relevant ones (marked with r). This change is a "promotion" because the Var rule only recognizes computationally relevant context entries.

Of all the rules, the Π-Form rule is perhaps the least intuitive. Since Π is a type former, one might expect this rule to use the c rather than r judgment form. However, in a dependently typed language, terms may evaluate (at run-time) to types, so the mode r is appropriate, as per Principle 1. Another possible surprise is that the context entry for x is marked with r rather than τ in the typing context of B. This is because the binding site of the x will never be erased: The only purpose of the context mark c is to enforce erasability of a λ^{c}-binder.

Type-Correctness. The type-correctness invariant is easily seen to hold by simply ignoring all erasure annotations in Figure 3.1. The resulting rules are exactly those of PTS from Figure 2.9, plus an additional, useless typing rule

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : A}$$

resulting from ignoring annotations in the EPTS Reset rule (if we ignore all erasure annotations, Γ and Γ° correspond to the same underlying PTS context).

Phase distinctions in type theory. Note that our notion of phase distinction is different from another notion of phase distinction found in the literature [16, 40], whereby compile-time entities are prevented from depending computationally on run-time entities. The motivation for this other form of phase distinction comes from languages that admit non-terminating run-time entities. In order to preserve decidability of type-checking, all such divergent terms must be prevented from appearing in types, where they would ruin the decidability of type equality.

In contrast, we prevent run-time entities from depending computationally on compile-time entities. Our motivation is that only run-time entities will survive the erasure phase, and therefore a run-time entity will be impossible to compute if it depends computationally on a previously erased compile-time entity.

3.1.3 Semantics

The default operational semantics of EPTS is simply β-reduction. We do not commit to any particular evaluation order, so the single-step reduction relation is non-deterministic.

Actually this is only one of *two* different operational semantics for EPTS. The remainder of this paper introduces an erasure semantics with potential for more efficient execution.

3.1.4 Meta-theory

Figure 3.4 on page 77 concisely presents all of the meta-theory of both EPTS (in the top half) and the erasure translation (in the bottom half). Each box in the figure contains a particular result of the meta-theory. As the development follows closely that of Pure Type Systems, we focus on the changes due to introducing erasure annotations. In this section we state each result, discuss its meaning, and outline a brief sketch of the proof. Full proofs of all the results mentioned here are found in Appendix A.1.

Relative Strength of Judgment Modes

We first investigate the relative strength of typing judgments and typing assumptions in c-mode and r-mode. Because the c-mode typing judgment is defined in terms of context reset, we start with properties of that operation.

The context reset operation is idempotent.

Lemma 3.1.2 (Reset Idempotence)

$$\Gamma^{\circ\circ} = \Gamma^{\circ}$$

Once all context entry annotations have been set to r, it does not accomplish anything to set them all to r again. This is easily proved by induction on Γ.

Because run-time variables may be used in places that compile-time variables may not, the assumption $x{:}^{\mathsf{r}}A$ is stronger than $x{:}^{\mathsf{c}}A$. For this reason, context reset strengthens the typing context and, contravariantly, weakens the overall judgment.

Lemma 3.1.3 (Reset Weakening)

$$\frac{\Gamma, \Delta \vdash M :^{\tau} A}{\Gamma^{\circ}, \Delta \vdash M :^{\tau} A}$$

Splitting the typing context into Γ and Δ in the statement of this lemma yields a more useful induction hypothesis. *Proof Sketch:* The proof is by structural induction on the typing derivation. The interesting cases are RESET, where we appeal to the idempotence of context reset operation, and VAR and WEAK, which proceed by cases on whether $\Delta = \varepsilon$ or not.

An immediate consequence of the reset weakening lemma is that it is easier to prove $\Gamma \vdash M :^{\mathsf{c}} A$ than $\Gamma \vdash M :^{\mathsf{r}} A$ because the former is equivalent to $\Gamma^{\circ} \vdash M :^{\mathsf{r}} A$, in which we have a stronger typing context. This observation is embodied in an admissible phase-weakening rule.

Corollary 3.1.4 (Phase Weakening)

$$\frac{\Gamma \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\mathsf{c}} A}$$

The upshot of these results is that assumptions and conclusions are stronger in r-mode than c-mode. This is because, in general, fewer resources from the original program are available at run-time due to erasure.

Substitution Lemma

Next, we prove a substitution lemma.

Lemma 3.1.5 (Substitution)

$$\frac{\Gamma, x{:}^{\tau_1} A, \Delta \vdash M :^{\tau_2} B \qquad \Gamma \vdash N :^{\tau_1} A}{\Gamma, \Delta[N/x] \vdash M[N/x] :^{\tau_2} B[N/x]}$$

The only novelty here is that the mode τ_1 of the typing judgment for the term N to be substituted must match the context entry mark of the variable x for which it will be substituted. Also, the mode τ_2 of the subject M is the same before and after the substitution.

Proof Sketch: By induction on the typing derivation. The interesting cases are RESET (requiring Phase Weakening) and VAR and WEAK (each proceeding by cases of whether $\Delta = \varepsilon$ or not).

Coherence Lemma

The Coherence Lemma says that our type system is internally coherent in the following way — If one can derive that M has type A, then one can also prove that A is a type.

Lemma 3.1.6 (Coherence)

$$\frac{\Gamma \vdash M :^{\tau} A}{(\exists s) \quad A = s \quad \vee \quad \Gamma \vdash A :^{c} s}$$

Proof Sketch: By structural induction on the typing derivation. The interesting cases are RESET, using Reset Idempotence, and Π-ELIM, which makes use of Phase Weakening and the Substitution Lemma.

Subject Reduction

Finally, we prove that reduction preserves types. This result is known as subject reduction.

Lemma 3.1.7 (Subject Reduction)

$$\frac{\Gamma \vdash M :^{\tau} A \qquad M \rightarrow_{\beta} N}{\Gamma \vdash N :^{\tau} A}$$

Note that the mode τ of the typing judgment is preserved as well as the type.

Proof Sketch: By structural induction on the typing derivation. The most interesting case is Π-ELIM in which we use the Substitution Lemma.

3.2 IMPLICIT PURE TYPE SYSTEMS

The target language of the erasure translation is Implicit Pure Type Systems (IPTS), a family of implicitly typed calculi with both explicit and implicit dependent products. This calculus is modeled after Miquel's Implicit Calculus of Constructions (ICC) [64, 65].

The syntax of IPTS is as follows:

$$\begin{array}{rl} (\textit{terms}) & M, N, A, B ::= x \mid \lambda x.\, M \mid M\ N \mid \Pi x{:}A.\, B \mid \forall x{:}A.\, B \mid s \\ (\textit{contexts}) & \Gamma, \Delta ::= \varepsilon \mid \Gamma, x{:}A \end{array}$$

Note the distinction between $\Pi x{:}A.\, B$ (explicit product) and $\forall x{:}A.\, B$ (implicit product) as well as the omission of domain labels from λ-abstractions.

The difference between explicit and implicit products shows up in the type system (Figure 3.3). Whereas the explicit product is introduced by functional abstraction (rule Π-Intro) and eliminated by function application (rule Π-Elim), no syntactic cues indicate introduction or elimination of the implicit product (rules $\forall$-Intro and $\forall$-Elim). This is what is meant by the terms "explicit" and "implicit".

Another way to think of the difference between Π and $\forall$ is that Π indicates functional abstraction (as usual) and $\forall$ indicates a *highly generic form of parametric polymorphism*. We say that $\forall$ indicates polymorphism because the $\forall$-Elim rule shows that a term M of type $\forall x{:}A.\, B$ also has the type $B[N/x]$ whenever N has type A. So M can take on many types and is therefore polymorphic. This notion of polymorphism is parametric because all instantiations $M : B[N/x]$ of a polymorphic term $M : \forall x{:}A.\, B$ behave in the same way, because they are all the same term, namely M.

This form of parametric polymorphism is highly generic because the parameter $x : A$ over which M is polymorphic can be just about anything. If A is a sort (like $\star$ in System F), then x is a type and we have the familiar notion of type-

$\boxed{\Gamma \vdash M : A}$

$$\textsc{Axiom}\ \frac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 : s_2} \qquad \textsc{Var}\ \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \qquad \textsc{Weak}\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash M : B}{\Gamma, x{:}A \vdash M : B}$$

$$\Pi\text{-}\textsc{Form}\ \frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.\, B : s_3} \qquad \forall\text{-}\textsc{Form}\ \frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.\, B : s_3}$$

$$\Pi\text{-}\textsc{Intro}\ \frac{\Gamma \vdash \Pi x{:}A.\, B : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.\, M : \Pi x{:}A.\, B} \qquad \forall\text{-}\textsc{Intro}\ \frac{x \notin FV(M) \qquad \Gamma \vdash \forall x{:}A.\, B : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash M : \forall x{:}A.\, B}$$

$$\Pi\text{-}\textsc{Elim}\ \frac{\Gamma \vdash M : \Pi x{:}A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]} \qquad \forall\text{-}\textsc{Elim}\ \frac{\Gamma \vdash M : \forall x{:}A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M : B[N/x]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash M : B}$$

Figure 3.3: Typing rules for IPTS. Note that ∀-Intro and ∀-Elim are not syntax-directed.

polymorphism. If A is a type such as Nat, then the type $\forall x{:}Nat.\, B$ expresses polymorphism over the number x. The possibilities are only limited by the rules of type-formation, embodied by the $\mathcal{R}$ component of the underlying PTS specification. Therefore this notion of polymorphism is generic in the type A of the parameter x over which one may form polymorphic entities.

IPTS is both more general and less general than Miquel's ICC. It is more general because IPTS is defined in terms of an arbitrary PTS specification whereas ICC commits to a particular specification, namely the same specification as Luo's Extended Calculus of Constructions (ECC)[1]. It is less general because (1) ICC uses $\beta\eta$-conversion instead of β-conversion in determining type equality, (2) ICC supports a notion of universe subtyping called *cumulativity* as in ECC, and (3) ICC contains extra typing rules ensuring η subject reduction.

ICC also has a rich (derived) notion of subtyping that orders the many types one may assign to a particular Church encoding in a natural way according to how precisely they characterize their inhabitants [64]. As it does not relate to our goals, we have not studied subtyping in IPTS.

Miquel's stated motivation for developing ICC was to overcome the "inherent verbosity" of programs in "PTS-based formalisms". He notes that this verbosity makes PTS programs more difficult to write than programs written using the implicit polymorphism of "ML-style languages" and "tends to hide the real computational contents of proof-terms behind a lot of 'noise' ". Though he hints at a distinction, this motivation seems to conflate two different issues:

1. Some subterms in a program are specificationally redundant (i.e., completely determined by the type system given their context) and therefore may be inferred even if omitted.

2. Some subterms in a program are computationally irrelevant (i.e., needed

[1]See Section 2.10.3 for an explanation of the Extended Calculus of Constructions.

for type-checking but cannot affect the ultimate value of the program) and therefore may be elided before evaluation.

Towards the end of the paper introducing ICC, Miquel gives an example of a function parameter that is redundant but not irrelevant, and concludes in a footnote that these two issues are largely independent. We agree. In our view, ICC addresses issue 2 but not issue 1.

For our purposes, IPTS turns out to be a perfect target language for erasure. The fact that IPTS supports parametric polymorphism will provide an insight into the meaning of the non-computational function space $\Pi^{\mathsf{c}} x{:}A.\, B$ of EPTS.

3.3 THE ERASURE TRANSLATION

The erasure translation from EPTS to IPTS strips out all the portions of a program annotated as computationally irrelevant. This translation is defined as follows:

Definition 3.3.1 (Erasure) $\boxed{\Gamma^\bullet}$ *and* $\boxed{M^\bullet}$

$$\varepsilon^\bullet = \varepsilon \qquad (\Gamma, x{:}^\tau A)^\bullet = \Gamma^\bullet, x{:}A^\bullet \qquad x^\bullet = x \qquad s^\bullet = s$$

$$(\Pi^{\mathsf{r}} x{:}A.\, B)^\bullet = \Pi x{:}A^\bullet.\, B^\bullet \quad (\lambda^{\mathsf{r}} x{:}A.\, M)^\bullet = \lambda x.\, M^\bullet \quad (M@^{\mathsf{r}} N)^\bullet = M^\bullet\ N^\bullet$$

$$(\Pi^{\mathsf{c}} x{:}A.\, B)^\bullet = \forall x{:}A^\bullet.\, B^\bullet \quad (\lambda^{\mathsf{c}} x{:}A.\, M)^\bullet = M^\bullet \qquad (M@^{\mathsf{c}} N)^\bullet = M^\bullet$$

Note that both λ^{c}-binders and $@^{\mathsf{c}}$-arguments are erased in the translation, as are domain type annotations in λ^{r}-abstractions. Note also that Π^{c} translates to $\forall$. The fact that this translation is sensible (as we prove in the remainder of this section) shows that Π^{c} actually indicates parametric polymorphism in EPTS. This supports our claim that parametric polymorphism can be understood entirely in terms of erasure.

3.3.1 Meta-theory

The bottom half of Figure 3.4 sketches out the meta-theory of erasure. We now discuss the significance of the results listed there.

Post-erasure Variable Occurrences

A key lemma characterizes which variables' occurrences in a term may survive erasure: they are all r-annotated in the typing context.

Definition 3.3.2 (Context variables and run-time variables)

$$\boxed{CV(\Gamma)}$$

$$\begin{array}{rcl} CV(\varepsilon) & = & \emptyset \\ CV(\Gamma, x{:}^{\tau} A) & = & CV(\Gamma) \cup \{x\} \end{array}$$

$$\boxed{RV(\Gamma)}$$

$$\begin{array}{rcl} RV(\varepsilon) & = & \emptyset \\ RV(\Gamma, x{:}^{r} A) & = & RV(\Gamma) \cup \{x\} \\ RV(\Gamma, x{:}^{c} A) & = & RV(\Gamma) \end{array}$$

Lemma 3.3.3 (Variable Survival)

$$\frac{\Gamma \vdash M :^{\mathsf{r}} A}{FV(M^{\bullet}) \subseteq RV(\Gamma)}$$

The proof is by a straightforward induction on the typing derivation.

Preservation of Reductions

Since computation happens by substitution, we first show that erasure commutes with substitution.

Lemma 3.3.4 (Erasure/Substitution Commutativity)

$$(M[N/x])^{\bullet} = M^{\bullet}[N^{\bullet}/x]$$

Proof: Erasure/Substitution Commutativity is proved by straightforward induction on M.

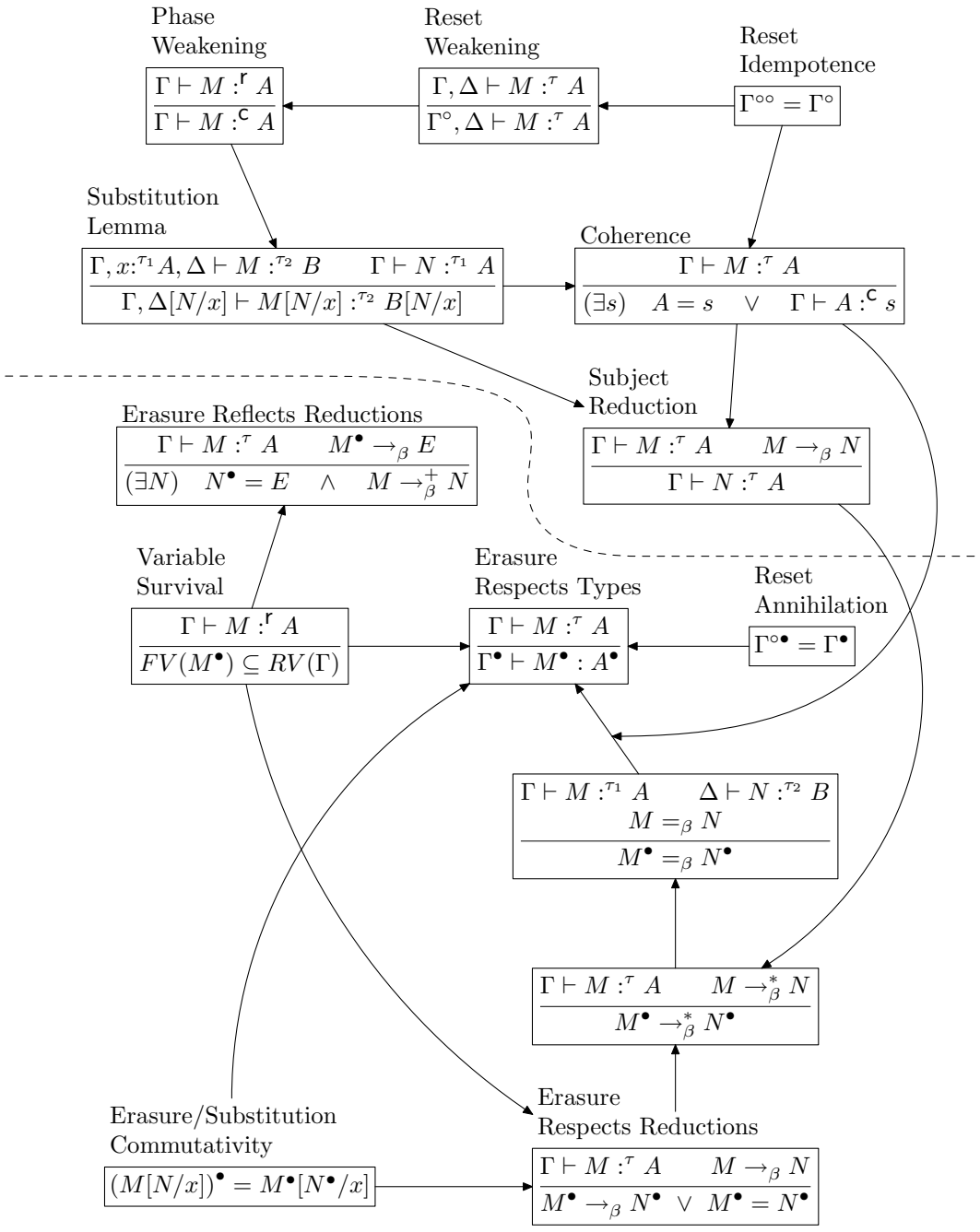


Figure 3.4: Identities and admissible rules in the meta-theory of EPTS (above the dotted line) and erasure (below it). Arrows indicate proof dependencies.

We then show that erasure respects reduction in the following sense: Each reduction step of a well-formed term in EPTS maps to either one *or zero* reduction steps in IPTS.

Theorem 3.3.5 (Erasure Respects Reduction)

$$\frac{\Gamma \vdash M :^{\tau} A \qquad M \rightarrow_{\beta} N}{M^{\bullet} \rightarrow_{\beta} N^{\bullet} \ \vee \ M^{\bullet} = N^{\bullet}}$$

Proof: The proof that erasure respects reductions proceeds by straightforward induction over the typing derivation. The interesting cases are Π-Intro and Π-Elim, where we proceed by cases on τ. In the Π-Elim case when the reduction step is β, the proof depends on Erasure/Substitution Commutativity when $\tau = \mathsf{r}$ and on Variable Survival when $\tau = \mathsf{c}$.

The proof that erasure respects reduction shows that some EPTS reductions in fact do no work when viewed through the lens of erasure. This is precisely why we want an erasure semantics — to eliminate the work associated with run-time-irrelevant portions of a program. Examination of the proof shows *where* erasure eliminates work. As expected, the eliminated work includes erased redices (terms of the form $(\lambda^{\mathsf{c}} x{:}A.\, M)@^{\mathsf{c}} N$, which erase to just $M^{\bullet}$) as well as unnecessary reduction steps inside domain-annotations and erased arguments.

The following corollaries follow immediately.

Corollary 3.3.6

$$\frac{\Gamma \vdash M :^{\tau} A \qquad M \rightarrow_{\beta}^{*} N}{M^{\bullet} \rightarrow_{\beta}^{*} N^{\bullet}} \qquad\qquad \frac{\Gamma \vdash M :^{\tau_1} A \qquad \Delta \vdash N :^{\tau_2} B \qquad M =_{\beta} N}{M^{\bullet} =_{\beta} N^{\bullet}}$$

Note: the proof of the latter requires the Church-Rosser theorem for EPTS.

Preservation of Typing

Again, we first investigate the properties of the context reset operation. The erasure operation annihilates it.

Lemma 3.3.7 (Reset Annihilation)

$$\Gamma^{\circ\bullet} = \Gamma^{\bullet}$$

Proof: By straightforward induction on Γ.

Then we prove that erasure respects types.

Theorem 3.3.8 (Erasure Respects Types)

$$\frac{\Gamma \vdash M :^{\tau} A}{\Gamma^{\bullet} \vdash M^{\bullet} : A^{\bullet}}$$

Proof: We prove this theorem by structural induction on the typing derivation. The interesting cases are:

- Reset, in which Reset Annihilation is used to simplify $\Gamma^{\circ\bullet}$;
- Π-Intro, in which Variable Survival is used to ensure the premise $x \notin FV(M^{\bullet})$ of the $\forall$-Intro rule of IPTS;
- Π-Elim, in which Erasure/Substitution Commutativity is used to simplify the type of the application; and
- Conv, in which Coherence and the fact that erasure preserves conversion are used to establish the premise $A^{\bullet} =_{\beta} B^{\bullet}$ of the IPTS Conv rule.

Reflection of Reductions

Next we show that a reduction of a post-erasure IPTS term can be reflected back into one *or more* EPTS reductions.

Theorem 3.3.9

$$\frac{\Gamma \vdash M :^{\tau} A \qquad M^{\bullet} \rightarrow_{\beta} E}{(\exists N) \quad N^{\bullet} = E \quad \wedge \quad M \rightarrow_{\beta}^{+} N}$$

Proof: By structural induction on the typing derivation. The interesting case is Π-ELIM when the @-annotation is $\tau = \mathsf{r}$ and the reduction is a β-step $(\lambda x.\, P^{\bullet})\ N_0{}^{\bullet} \rightarrow_{\beta} P^{\bullet}[N_0{}^{\bullet}/x]$. In this case, $M = M_0 @^{\mathsf{r}} N_0$ and $M_0{}^{\bullet} = \lambda x.\, P^{\bullet}$ and $E = P^{\bullet}[N_0{}^{\bullet}/x]$. The only way $M_0{}^{\bullet}$ can be $\lambda x.\, P^{\bullet}$ is if M_0 is a $\lambda^{\mathsf{r}} x{:}B.\, P$ nested under some (perhaps zero) "frames" of the form $\lambda^{\mathsf{c}} y{:}C.\, [\,]$ or $[\,] @^{\mathsf{c}} N$. Because the type of M_0 is $\Pi^{\mathsf{r}} x{:}A.\, B$, we know the top-most (outer-most) frame cannot be a λ^{c}. Similarly, for typing reasons, the bottom-most (inner-most) frame cannot be a $@^{\mathsf{c}}$, because it would be applied to a λ^{r}. Therefore, if there are any frames at all on top of $\lambda^{\mathsf{r}} x{:}B.\, P$, then there are at least two, and at some point there is a λ^{c} frame just underneath a $@^{\mathsf{c}}$ one, forming a redex. If we reduce this redex, the rest of the frame structure remains intact, and the number of frames decreases by two. We may repeat this process until no intermediate frames are left. Then $M_0 \rightarrow_{\beta}^{*} \lambda^{\mathsf{r}} x{:}B[\theta].\, P[\theta]$ where θ is the sequence of substitutions effected by the sequence of reductions. Because θ is comprised solely of substitutions for λ^{c}-bound variables, Variable Survival tells us there will be no occurrences of these variables inside $P^{\bullet}$. Therefore $P[\theta]^{\bullet} = P^{\bullet}[\theta^{\bullet}] = P^{\bullet}$. Let $N = P[\theta][N_0/x]$. Then

$$N^{\bullet} = P[\theta][N_0/x]^{\bullet} = P[\theta]^{\bullet}[N_0{}^{\bullet}/x] = P^{\bullet}[N_0{}^{\bullet}/x] = E$$

and $M \rightarrow_{\beta}^{+} N$ because

$$M = M_0 @^{\mathsf{r}} N_0 \rightarrow_{\beta}^{*} (\lambda^{\mathsf{r}} x{:}B[\theta].\, P[\theta]) @^{\mathsf{r}} N_0 \rightarrow_{\beta} P[\theta][N_0/x] = M',$$

thereby completing this case of the proof. □

This proof shows that certain reduction steps in IPTS (of post-erasure EPTS terms) require additional reductions in the original EPTS term before an EPTS reduction corresponding to the IPTS reduction can take place. This means that some of the work that erasure avoids is unavoidable, in general, without erasure.

This theorem says that any post-erasure reduction corresponds to a potential pre-erasure reduction. In other words, the erasure of a well-formed EPTS term cannot reduce in IPTS in a strange way that was not possible in EPTS.

3.3.2 Erasure Semantics

The erasure semantics for EPTS is simply this: First erase and then execute in IPTS. The meta-theory supports the claim that this is a good erasure semantics.

Theorem 3.3.5 : erasure eliminates some old work

Theorem 3.3.8 : erasure does not introduce any new work

Theorem 3.3.9 : erasure preserves the meanings (types) of programs

One final result supports the validity of our erasure semantics for EPTS. We would not want a PTS program to compute to a value while some annotation of it diverges under the erasure semantics. Thankfully, this cannot happen.

Theorem 3.3.10 (Erasure Preserves Strong Normalization)
For a strongly normalizing PTS, any well-typed term in the corresponding EPTS erases to a strongly normalizing IPTS term.

Proof: Suppose there is an infinite reduction sequence in IPTS starting with the erasure of a well-typed term M in EPTS. Because erasure reflects reductions and we have EPTS Subject Reduction, this reflects back onto an infinite reduction sequence in EPTS starting with M. Because $\flat$ (the erasure-annotation-forgetting map from EPTS to PTS) preserves both reduction steps and typing judgments, we obtain an infinite reduction sequence in the underlying PTS starting with the well-typed term $M^{\flat}$. But this contradicts our assumption that the underlying PTS is strong-normalizing. □

3.4 IMPLEMENTATION

The similarity between the typing rules for PTS and EPTS indicates that using types to track computational irrelevance does not require a radical restructuring of the typing rules. One would hope, then, that it is similarly straightforward to extend an existing type-checker to handle erasure annotations. We have indeed found this to be the case in a prototype implementation of a simple dependently typed language.

One must add τ annotations to the abstract syntax and some extra logic to the type-checker to handle these annotations properly. As for efficiency of type-checking, the only potential increase in the time complexity comes from the context reset operation. The naive implementation of this operation, as defined in Definition 3.1.1, takes time proportional to the length of the typing context.

However, our implementation uses a clever representation of typing contexts that renders context reset a constant-time operation. The new representation of typing contexts is as follows (where i denotes an integer):

$$\begin{array}{rrcl} (\textit{typing contexts}) & \Gamma & ::= & [\![\hat{\Gamma}]\!]_i \\ (\textit{internal contexts}) & \hat{\Gamma} & ::= & \hat{\varepsilon} \mid \hat{\Gamma}, x{:}^i A \end{array}$$

This representation of typing contexts consists of a pair of an integer i and an underlying context $\hat{\Gamma}$ annotated with integers rather than erasure annotations. The top-level integer is called the reset count because it counts how many times prefixes of Γ have been reset. Every integer annotation on context entries is either (a) less than or equal to the reset count (representing the annotation r) or (b) equal to the reset count plus one (representing the annotation c). More concisely, for every cleverly represented typing context $[\![\hat{\Gamma}]\!]_i$, it must be the case that $j \leq i + 1$ for each each integer annotation j in $\hat{\Gamma}$. This invariant must be maintained at all times.

Given this representation, the original context operations are re-implemented

Representation of typing contexts

$$\begin{array}{rrcl} (\textit{typing contexts}) & \Gamma & ::= & [\![\hat{\Gamma}]\!]_i \\ (\textit{internal contexts}) & \hat{\Gamma} & ::= & \hat{\varepsilon} \mid \hat{\Gamma}, x{:}^i A \end{array}$$

Core operations

$$\boxed{\varepsilon} \qquad \varepsilon = [\![\hat{\varepsilon}]\!]_0$$

$$\boxed{\Gamma, x{:}^\tau A} \qquad \begin{array}{rcl} [\![\hat{\Gamma}]\!]_i, x{:}^{\mathsf{c}} A & = & [\![\hat{\Gamma}, x{:}^{i+1} A]\!]_i \\ [\![\hat{\Gamma}]\!]_i, x{:}^{\mathsf{r}} A & = & [\![\hat{\Gamma}, x{:}^i A]\!]_i \end{array}$$

$$\boxed{\Gamma^\circ} \qquad [\![\hat{\Gamma}]\!]_i^\circ = [\![\hat{\Gamma}]\!]_{i+1}$$

$$\boxed{x{:}^{\mathsf{r}} A \in \Gamma} \qquad x{:}^{\mathsf{r}} A \in [\![\hat{\Gamma}]\!]_i \quad \text{iff} \quad x{:}^j A \in \hat{\Gamma} \text{ and } j \leq i$$

Figure 3.5: Clever Implementation of Typing Contexts

as shown in Figure 3.5. Clearly the context reset operation is a constant time operation in this representation. Inspection reveals that each operation preserves the above-mentioned invariant.

We must show that this representation is equivalent to the naive representation introduced in Section 3.1.1 with the definition of context reset (Definition 3.1.1). We will state this equivalence in terms of two mappings going back and forth between the two different representations.

The mapping from the naive representation to the clever implementation is effectively given by the previously stated re-implementations of ε and $\Gamma, x{:}^\tau A$ in this section. We write this mapping as $\Gamma^\sharp$.

Definition 3.4.1 $\boxed{\Gamma^\sharp}$

$$\varepsilon^\sharp = [\![\hat{\varepsilon}]\!]_0 \qquad (\Gamma, x{:}^{\mathsf{c}} A)^\sharp = [\![\hat{\Gamma}, x{:}^{i+1} A]\!]_i \text{ where } \Gamma^\sharp = [\![\hat{\Gamma}]\!]_i \qquad (\Gamma, x{:}^{\mathsf{r}} A)^\sharp = [\![\hat{\Gamma}, x{:}^i A]\!]_i \text{ where } \Gamma^\sharp = [\![\hat{\Gamma}]\!]_i$$

We define the mapping from the clever to the naive implementation as follows:

Definition 3.4.2 $\boxed{\Gamma^\flat}$

$$[\![\hat{\varepsilon}]\!]^\flat_i = \varepsilon \qquad\qquad [\![\hat{\Gamma}, x{:}^j A]\!]^\flat_i = \begin{cases} [\![\hat{\Gamma}]\!]^\flat_i, x{:}^{\mathsf{r}} A & \text{if } j \leq i \\ [\![\hat{\Gamma}]\!]^\flat_i, x{:}^{\mathsf{c}} A & \text{otherwise} \end{cases}$$

We think of the $\flat$ mapping as defining the *meaning* of a cleverly represented context. As there may be multiple cleverly represented contexts with the same meaning, the appropriate notion of equality for cleverly represented contexts is equality of their meanings (i.e., their corresponding naive representations).

Definition 3.4.3 $\boxed{\Gamma \cong \Delta}$

$$\Gamma \cong \Delta \quad \textit{iff} \quad \Gamma^\flat = \Delta^\flat \qquad\qquad (\textit{i.e.,}\ [\![\hat{\Gamma}]\!]_i \cong [\![\hat{\Delta}]\!]_j \quad \textit{iff} \quad [\![\hat{\Gamma}]\!]^\flat_i = [\![\hat{\Delta}]\!]^\flat_j)$$

We are abusing notation somewhat in using the metavariable Γ to stand for typing contexts in both the naive and clever representations. However, it should always be apparent from context which representation is meant.

We prove that the naive representation of typing contexts is isomorphic to the clever representation quotiented by the $\cong$ relation. The term "isomorphic" means that there is a bijection between the two sets that respects each of the core operations on typing contexts.

First, we must show that the basic typing context operations are well-defined on $\cong$-equivalence classes of cleverly represented typing contexts.

Lemma 3.4.4 *If* $\Gamma \cong \Delta$ *then* $\Gamma, x{:}^\tau A \cong \Delta, x{:}^\tau A$.

Lemma 3.4.5 *If* $\Gamma \cong \Delta$ *then* $\Gamma^\circ \cong \Delta^\circ$

Lemma 3.4.6 *If* $\Gamma \cong \Delta$ *then* $x{:}^{\mathsf{r}} A \in \Gamma$ *iff* $x{:}^{\mathsf{r}} A \in \Delta$

This means that out of all the cleverly represented Γs with the same meaning (i.e., corresponding to the same naively represented Δ), it doesn't matter which Γ

we pick to represent Δ because all the operations on cleverly represented typing contexts are meaning preserving.

Next, we must show that both mappings between representations respect the structure of typing contexts.

Theorem 3.4.7 (Soundness) *The following identities hold:*

$$\varepsilon = \varepsilon^\flat \qquad \Gamma^\flat, x{:}^\tau A = (\Gamma, x{:}^\tau A)^\flat \qquad (\Gamma^\flat)^\circ = (\Gamma^\circ)^\flat \qquad x{:}^\mathsf{r} A \in \Gamma \text{ } \textit{iff } x{:}^\mathsf{r} A \in \Gamma^\flat$$

Theorem 3.4.8 (Completeness) *The following identities hold:*

$$\varepsilon \cong \varepsilon^\sharp \qquad \Gamma^\sharp, x{:}^\tau A \cong (\Gamma, x{:}^\tau A)^\sharp \qquad (\Gamma^\sharp)^\circ \cong (\Gamma^\circ)^\sharp \qquad x{:}^\mathsf{r} A \in \Gamma \text{ } \textit{iff } x{:}^\mathsf{r} A \in \Gamma^\sharp$$

In mathematical parlance, the $\sharp$ and $\flat$ mappings are *homomorphisms* in the abstract algebra of typing contexts. This means that they are meaningful as mappings between algebras as opposed to merely being meaningful as mappings between sets.

Finally, we must show that the mappings $\sharp$ and $\flat$ are inverses of each other.

Lemma 3.4.9 ($\flat$ undoes $\sharp$)

$$(\Gamma^\sharp)^\flat = \Gamma$$

Corollary 3.4.10 ($\sharp$ undoes $\flat$)

$$\Gamma \cong \Delta \Longrightarrow (\Gamma^\flat)^\sharp \cong \Delta$$

This means that the $\flat$ and $\sharp$ homomorphisms witness an isomorphism between the naive and clever algebras of typing contexts.

In summary, these results (which are all proved in Appendix A.3) show that the naive and clever implementations of typing contexts are functionally equivalent. However, the clever version is more efficient, so it is the one we prefer to implement.

3.5 CONCLUSIONS

Languages combining dependent types with erasure semantics sometimes require users to maintain more than one copy of a datatype to ensure erasure of some of its values but not others. This problem stems from the treatment of computational irrelevance as an intrinsic property of data, rather than a property of the way that data is used.

By treating computational irrelevance extrinsically — in particular, by distinguishing functions that may not depend computationally on their arguments from those that may — we arrive at a flexible notion of erasure semantics that generalizes both type erasure and proof erasure (i.e., program extraction) and overcomes the code duplication problem. The meta-theory of the erasure translation shows that the resulting erasure semantics is both sound and useful for eliminating extra work.

The erasure translation also exposes the fact our notion of erasure corresponds to a highly generic form of parametric polymorphism over arbitrary sorts of entities (types, proofs, numbers, etcetera). Because parametric polymorphism is a familiar concept from typed functional programming languages, we hope that programming in an EPTS-like language will be somewhat natural for ML and Haskell programmers.

Chapter 4

ERASABILITY ANALYSIS

The previous chapter showed how to equip Pure Type Systems with an erasure semantics. This erasure semantics is entirely guided by annotations in EPTS terms. However, manual program annotation may be undesirable or infeasible in some situations (e.g., for large legacy programs). For this reason we would also like to support programs written in an erasure-oblivious style.

In this chapter, we develop an automatic program analysis that determines which portions of a program should be erased. The output of this analysis is a well-annotated (i.e., phase correct) EPTS term. We prove that our analysis decorates well-typed PTS terms with erasure annotations that mark as much of a program for erasure as possible.

4.1 AN EXAMPLE

Figure 4.1 shows in greater detail how the following example program goes through the various stages of analysis and erasure depicted in Figure 1.4. The erasure phase is straightforward, but the analysis phase is more involved.

$$
\begin{array}{lcl}
(*\ \mathit{in\ PTS}\ *) & & (*\ \mathit{in\ IPTS}\ *) \\
\underline{\mathsf{let}}\ f = \lambda x{:}\mathbb{N}.\,5\ \underline{\mathsf{in}} & & \underline{\mathsf{let}}\ f = 5\ \underline{\mathsf{in}} \\
\underline{\mathsf{let}}\ g = \lambda y{:}\mathbb{N}.\,9\ \underline{\mathsf{in}} & \Longrightarrow & \underline{\mathsf{let}}\ g = 9\ \underline{\mathsf{in}} \\
\underline{\mathsf{let}}\ h = \lambda z{:}\mathbb{N} \to \mathbb{N}.\,z\ 7\ \underline{\mathsf{in}} & & \underline{\mathsf{let}}\ h = \lambda z.\,z\ \underline{\mathsf{in}} \\
(h\ f, h\ g) & & (h\ f, h\ g)
\end{array}
$$

$$
\begin{array}{lll}
 & \underline{\mathsf{let}}\ f = \lambda^{\alpha_1} x{:}\mathbb{N}.\, 5\ \underline{\mathsf{in}} & \\
(A) & \underline{\mathsf{let}}\ g = \lambda^{\alpha_2} y{:}\mathbb{N}.\, 9\ \underline{\mathsf{in}} & (B) \\
\Longrightarrow & \underline{\mathsf{let}}\ h = \lambda^{\alpha_3} z{:}\mathbb{N} \xrightarrow{\alpha_7} \mathbb{N}.\, z@^{\alpha_4} 7\ \underline{\mathsf{in}} & \Longrightarrow \\
 & (h@^{\alpha_5} f, h@^{\alpha_6} g) &
\end{array}
\quad
\begin{array}{l}
(\alpha_7 = \alpha_4) \wedge (\alpha_3 = \alpha_5) \wedge \\
(\alpha_1 = \alpha_7) \wedge (\alpha_3 = \alpha_6) \wedge \\
(\alpha_2 = \alpha_7) \wedge (\neg\, \alpha_3)
\end{array}
$$

$$
(C)\ \begin{cases}
(\alpha_7 = \alpha_4) \wedge (\alpha_3 = \alpha_5) \wedge (\alpha_1 = \alpha_7) \wedge (\alpha_3 = \alpha_6) \wedge (\alpha_2 = \alpha_7) \wedge (\neg\, \alpha_3) \\
\hookrightarrow \quad (\alpha_7 = \alpha_4) \wedge (\mathsf{false} = \alpha_5) \wedge (\alpha_1 = \alpha_7) \wedge (\mathsf{false} = \alpha_6) \wedge (\alpha_2 = \alpha_7) \\
\hookrightarrow \quad (\alpha_7 = \alpha_4) \wedge (\alpha_1 = \alpha_7) \wedge (\alpha_2 = \alpha_7) \\
\hookrightarrow \quad \alpha_3, \alpha_5, \alpha_6 := \mathsf{false}; \quad \alpha_7, \alpha_4, \alpha_1, \alpha_2 := \mathsf{true};
\end{cases}
$$

$$
\begin{array}{lll}
 & \underline{\mathsf{let}}\ f = \lambda^{\mathsf{c}} x{:}\mathbb{N}.\, 5\ \underline{\mathsf{in}} & \\
(D) & \underline{\mathsf{let}}\ g = \lambda^{\mathsf{c}} y{:}\mathbb{N}.\, 9\ \underline{\mathsf{in}} & (E) \\
\Longrightarrow & \underline{\mathsf{let}}\ h = \lambda^{\mathsf{r}} z{:}\mathbb{N} \xrightarrow{\mathsf{c}} \mathbb{N}.\, z@^{\mathsf{c}} 7\ \underline{\mathsf{in}} & \Longrightarrow \\
 & (h@^{\mathsf{r}} f, h@^{\mathsf{r}} g) &
\end{array}
\quad
\begin{array}{l}
\underline{\mathsf{let}}\ f = 5\ \underline{\mathsf{in}} \\
\underline{\mathsf{let}}\ g = 9\ \underline{\mathsf{in}} \\
\underline{\mathsf{let}}\ h = \lambda z.\, z\ \underline{\mathsf{in}} \\
(h\ f, h\ g)
\end{array}
$$

Figure 4.1: Sketch of erasability analysis and erasure for an example program. Erasability analysis consists of (A) annotation with annotation variables; (B) constraint generation; (C) optimal constraint solution; and (D) solution-determined erasure annotation. Erasure consists of (E) an annotation-guided erasure phase.

In this example, the analysis identifies f and g as syntactically constant functions and identifies the argument 7 to which they are applied. These parts of the program are then marked for erasure. In a more realistic program, these erasable portions can be quite large.

The first step of analysis is to annotate a program with *annotation variables* that will later be assigned to concrete erasure annotations (A). In this step, every Π, λ, and @ is annotated with a distinct variable. Then we generate constraints in propositional logic whose solutions correspond to well-formed annotations of the underlying PTS term (B, Section 4.2). Then we find an optimal solution to the generated constraints corresponding to erasure annotations that mark as much of the program as possible for erasure (C, Section 4.3). Finally, this optimal solution is applied to the original program, decorating it with concrete erasure annotations (D) that guide the erasure phase (E).

4.2 CONSTRAINT GENERATION

In this section, we augment the syntax and typing rules of EPTS (explained in Section 3.1) to generate a constraint stating the phase-correctness of a program in terms of its annotation variables. The result is a variant of EPTS called EPTS$^{\mathcal{C}}$. We then prove that solutions to the generated constraint correspond to legal erasure annotations of the original program.

The generated constraints are formulas of propositional logic with annotation variables doubling as propositional variables. In order to identify erasure annotations and boolean values, we interpret c as true and r as false.

4.2.1 Syntax of Annotations and Constraints in EPTS$^{\mathcal{C}}$

We now describe the syntactic form of erasure annotations and generated constraints in EPTS$^{\mathcal{C}}$ and how they follow naturally from a careful study of the typing

rules of EPTS. While reading this section, it may be useful to refer back to the EPTS typing rules in Figure 3.1 and even to look ahead to the EPTS$^{\mathcal{C}}$ typing rules in Figure 4.3

The input to the constraint generation phase is an arbitrary PTS term, in which each λ, @, and Π is annotated with a distinct annotation *variable*. The constraint generation phase then generates a constraint in terms of these annotation variables. Therefore the syntax of EPTS$^{\mathcal{C}}$ terms is as follows:

$$(\textit{term}) \quad M, N, A, B ::= x \mid \lambda^{\alpha} x{:}A.\, M \mid M@^{\alpha} N \mid \Pi^{\alpha} x{:}A.\, B \mid s$$

Where α is an annotation *variable* rather than a concrete annotation $\tau \in \{\mathsf{r}, \mathsf{c}\}$. Because we interpret annotations as booleans, α is also a *propositional* variable.

What of the erasure annotation on the EPTS typing judgment? Along with the usual judgment forms $\Gamma \vdash N :^{\mathsf{r}} A$ and $\Gamma \vdash N :^{\mathsf{c}} A$, we now need an additional judgment form $\Gamma \vdash N :^{\alpha} A$. This is because applications are annotated with variables, so the Π-Elim typing rule of EPTS$^{\mathcal{C}}$ will have a premise of the form $\Gamma \vdash N :^{\alpha} A$. For this reason, we introduce a new syntactic category ρ of judgment modes.

$$(\textit{judgment mode}) \quad \rho ::= \alpha \mid \mathsf{r} \mid \mathsf{c}$$

The next question is what form an erasure annotation on a context entry may take. Each context entry starts out marked with either an r from a Π-binder (as in the rule Π-Form) or an α from a λ^{α}-binder (as in the rule Π-Intro). However, the full form of context entries is as follows:

$$(\textit{typing context}) \quad \Gamma ::= \varepsilon \mid \Gamma, x{:}^{\gamma} A$$
$$(\textit{context entry annotation}) \quad \gamma ::= \alpha \mid \mathsf{r} \mid \neg\rho \wedge \gamma$$

How does the syntax rule for context entry annotations $\gamma ::= \neg\rho \wedge \gamma$ arise? It results from the EPTS$^{\mathcal{C}}$ version of the context-reset operation. We will discuss the EPTS$^{\mathcal{C}}$ version of this operation presently.

As in EPTS, we will see that most EPTS$^{\mathcal{C}}$ typing rules have mode $\rho = \mathsf{r}$. The mode $\rho = \mathsf{c}$ is as easy to handle as in EPTS. However, to handle the mode $\rho = \alpha$, we need to generalize the RESET rule as follows:

$$\frac{\Gamma^{\circ} \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\mathsf{c}} A} \quad \mapsto \quad \frac{\Gamma^{\circ(\rho)} \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\rho} A}$$

where $\Gamma^{\circ(\rho)}$ is some generalization of Γ°, the operation that sets each context entry annotation in Γ to r in the EPTS typing rule RESET.

How should we define $\Gamma^{\circ(\rho)}$? To properly generalize the context reset operation of EPTS, we require that $\Gamma^{\circ(\mathsf{c})} = \Gamma^{\circ}$ so that the new RESET rule for EPTS$^{\mathcal{C}}$ instantiates to the old RESET rule for EPTS when $\rho = \mathsf{c}$. Similarly, when $\rho = \mathsf{r}$, we require that $\Gamma^{\circ(\mathsf{r})} = \Gamma$, because the premise of the RESET rule is already in r-mode. Therefore we define

$$(\Gamma, x{:}^{\gamma} A)^{\circ(\rho)} \quad = \quad \Gamma^{\circ(\rho)}, x{:}^{\gamma \circ \rho} A$$

where

$$\gamma \circ \rho \quad = \quad \text{if } \rho = \mathsf{r} \text{ then } \gamma \text{ else } \mathsf{r}$$

Under our boolean interpretation of erasure annotations ($\mathsf{c} = \mathsf{true}$ and $\mathsf{r} = \mathsf{false}$), we can express the conditional logic in the definition of $\gamma \circ \rho$ more succinctly:

$$\gamma \circ \rho \quad = \quad \text{if } \rho = \mathsf{r} \text{ then } \gamma \text{ else } \mathsf{r} \quad = \quad \text{if } \neg\, \rho \text{ then } \gamma \text{ else } \mathsf{false} \quad = \quad \neg\, \rho \wedge \gamma$$

This is how a context entry annotation may take the form $\neg\, \rho \wedge \gamma$.

Definition 4.2.1 (Generalized Reset Operation) $\boxed{\Gamma^{\circ(\rho)}}$

$$\varepsilon^{\circ(\rho)} = \varepsilon \qquad\qquad (\Gamma, x{:}^{\gamma} A)^{\circ(\rho)} = \Gamma^{\circ(\rho)}, x{:}^{\neg \rho \wedge \gamma} A$$

Lastly, we ask what is the form of the generated constraints? When typing occurrences of a variable x (in the VAR rule), we require its context annotation γ

to be r (false). Therefore, one form of atomic constraint is $\neg\,\gamma$. The other form of atomic constraint is $\alpha = \alpha'$, which occurs when we need to identify annotations. The overall constraint is a conjunction of atomic constraints. Therefore, we arrive at the following syntax for constraints, typing contexts, context entry annotations, and typing judgment modes.

$$\begin{aligned}
(\textit{constraint}) \quad & \mathcal{C}, \mathcal{D}, \mathcal{E} ::= \mathsf{true} \mid \mathcal{C} \wedge \mathcal{D} \mid \neg\,\gamma \mid \alpha = \alpha' \\
(\textit{typing context}) \quad & \Gamma ::= \varepsilon \mid \Gamma, x{:}^{\gamma} A \\
(\textit{context annotation}) \quad & \gamma ::= \alpha \mid \mathsf{r} \mid \neg\rho \wedge \gamma \\
(\textit{judgment mode}) \quad & \rho ::= \alpha \mid \mathsf{r} \mid \mathsf{c}
\end{aligned}$$

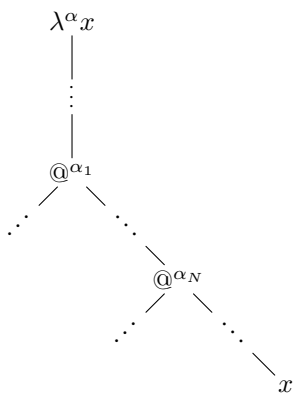
We identify both constraints and context annotations up to logical equivalence.

4.2.2 Constraint-Generating Typing Rules

Figure 4.2 shows how a typical constraint arises. It is useful to keep this typical case in mind when studying the typing rules.

Figures 4.3 and 4.4 contain the constraint-generating typing rules for EPTS$^{\mathcal{C}}$. The judgment forms are $\mathcal{C}\,;\,\Gamma \vdash M :^{\rho} A$ and $\mathcal{C} \vdash M =_{\beta} N$, the constraint-generating version of $\Gamma \vdash M :^{\tau} A$ and $M =_{\beta} N$, respectively.

The rules in Figure 4.3 follow the same pattern as the typing rules for EPTS in Figure 3.1. The only differences have to do with how constraints are gathered and how erasure annotations are represented (as described in the previous section). The constraint in the conclusion of each rule consists of the constraints of each premise that must be propagated as well as any constraint generated by the rule itself combined together into a single conjunction. In the Axiom rule this conjunction is the trivial empty conjunction true. The only typing rules that generate their own constraints are Var, in which the generated constraint $\neg\,\gamma$ corresponds to the requirement that the context entry of x be r (false), and Π-Intro, in which the generated constraint $\alpha = \alpha'$ corresponds to requirement that a λ-abstraction and its Π-type carry the same annotation.



If each $\alpha_i = \mathsf{r}$, then this occurrence of x will not be erased, and, therefore, neither can the λ binder, so α must be r. Therefore this occurrence of x gives rise to the constraint

$$(\alpha_1 = \mathsf{r} \wedge \cdots \wedge \alpha_N = \mathsf{r}) \Rightarrow \alpha = \mathsf{r},$$

which, since $\mathsf{r} = \mathsf{false}$, equals

$$(\neg\, \alpha_1 \wedge \cdots \wedge \neg\, \alpha_N) \Rightarrow \neg\, \alpha,$$

and also (by De Morgan's laws)

$$\alpha_1 \vee \cdots \vee \alpha_N \vee \neg\, \alpha.$$

Figure 4.2: How a typical constraint arises.

The rules in Figure 4.4 behave similarly to those in Figure 4.3 in terms of constraint generation. The underlying rules (ignoring constraints) are a fairly straightforward non-algorithmic presentation of β-conversion. The congruence rules each generate a constraint corresponding to the requirement that normal forms of convertible terms have matching annotations.

A final point to note is that the RESET rule uses the generalized context reset operation $\Gamma^{\circ(\rho)}$ to account for the generalized judgment mode ρ that may be either c or r or some annotation variable α.

4.2.3 Proof of Correctness

We now prove that the typing rules for EPTS$^{\mathcal{C}}$ are both sound and complete with respect to those of EPTS. In this section, the notation $\sigma \vDash \mathcal{C}$ means that the variable assignment σ satisfies the formula $\mathcal{C}$ (i.e., $\mathcal{C}$ evaluates to true under σ).

The next four lemmas concern the operation of applying an annotation variable

$\boxed{\mathcal{C}\,;\,\Gamma \vdash M :^{\rho} A}$

AXIOM
$$\frac{(s_1, s_2) \in \mathcal{A}}{\mathsf{true}\,;\,\varepsilon \vdash s_1 :^{\mathsf{r}} s_2}$$

VAR
$$\frac{\mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{c}} s}{\mathcal{C} \wedge \neg\,\gamma\,;\,\Gamma, x{:}^{\gamma} A \vdash x :^{\mathsf{r}} A}$$

WEAK
$$\frac{\mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{c}} s \qquad \mathcal{D}\,;\,\Gamma \vdash M :^{\mathsf{r}} B}{\mathcal{C} \wedge \mathcal{D}\,;\,\Gamma, x{:}^{\gamma} A \vdash M :^{\mathsf{r}} B}$$

Π-FORM
$$\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{r}} s_1 \qquad \mathcal{D}\,;\,\Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2}{\mathcal{C} \wedge \mathcal{D}\,;\,\Gamma \vdash \Pi^{\alpha} x{:}A.\, B :^{\mathsf{r}} s_3}$$

Π-INTRO
$$\frac{\mathcal{C}\,;\,\Gamma \vdash \Pi^{\alpha'} x{:}A.\, B :^{\mathsf{c}} s \qquad \mathcal{D}\,;\,\Gamma, x{:}^{\alpha} A \vdash M :^{\mathsf{r}} B}{\mathcal{C} \wedge \mathcal{D} \wedge \alpha = \alpha'\,;\,\Gamma \vdash \lambda^{\alpha} x{:}A.\, M :^{\mathsf{r}} \Pi^{\alpha'} x{:}A.\, B}$$

Π-ELIM
$$\frac{\mathcal{C}\,;\,\Gamma \vdash M :^{\mathsf{r}} \Pi^{\alpha} x{:}A.\, B \qquad \mathcal{D}\,;\,\Gamma \vdash N :^{\alpha'} A}{\mathcal{C} \wedge \mathcal{D} \wedge \alpha = \alpha'\,;\,\Gamma \vdash M @^{\alpha'} N :^{\mathsf{r}} B[N/x]}$$

CONV
$$\frac{\mathcal{C}\,;\,\Gamma \vdash M :^{\mathsf{r}} A \qquad \mathcal{D}\,;\,\Gamma \vdash B :^{\mathsf{c}} s \qquad \mathcal{E} \vdash A =_{\beta} B}{\mathcal{C} \wedge \mathcal{D} \wedge \mathcal{E}\,;\,\Gamma \vdash M :^{\mathsf{r}} B}$$

RESET
$$\frac{\mathcal{C}\,;\,\Gamma^{\circ(\rho)} \vdash M :^{\mathsf{r}} A}{\mathcal{C}\,;\,\Gamma \vdash M :^{\rho} A}$$

Figure 4.3: Constraint generating typing rules for EPTS$^{\mathcal{C}}$

$\boxed{\mathcal{C} \vdash M =_\beta N}$

$$\textsc{Refl}\ \frac{}{\mathsf{true} \vdash M =_\beta M} \qquad \textsc{Symm}\ \frac{\mathcal{C} \vdash M =_\beta N}{\mathcal{C} \vdash N =_\beta M} \qquad \textsc{Trans}\ \frac{\mathcal{C} \vdash M =_\beta M'' \qquad \mathcal{D} \vdash M'' =_\beta M'}{\mathcal{C} \wedge \mathcal{D} \vdash M =_\beta M'}$$

$$\textsc{Beta}\ \frac{}{\mathsf{true} \vdash (\lambda^\alpha x{:}A.\, M)@^{\alpha'} N =_\beta M[N/x]}$$

$$\textsc{CongPi}\ \frac{\mathcal{C} \vdash A =_\beta A' \qquad \mathcal{D} \vdash B =_\beta B'}{\alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D} \vdash \Pi^\alpha x{:}A.\, B =_\beta \Pi^{\alpha'} x{:}A'.\, B'}$$

$$\textsc{CongLam}\ \frac{\mathcal{C} \vdash A =_\beta A' \qquad \mathcal{D} \vdash M =_\beta M'}{\alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D} \vdash \lambda^\alpha x{:}A.\, M =_\beta \lambda^{\alpha'} x{:}A'.\, M'}$$

$$\textsc{CongApp}\ \frac{\mathcal{C} \vdash M =_\beta M' \qquad \mathcal{D} \vdash N =_\beta N'}{\alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D} \vdash M@^\alpha N =_\beta M'@^{\alpha'} N'}$$

Figure 4.4: Constraint generating conversion rules for EPTS$^{\mathcal{C}}$

assignment σ to a term or context and how it interacts with other operations and relations such as context reset, substitution, and reduction.

Lemma 4.2.2 (Correctness of Generalized Context Reset)

$$\sigma(\Gamma^{\circ(\rho)}) = \begin{cases} \sigma\Gamma & \text{if } \sigma(\rho) = \mathsf{r} \\ (\sigma\Gamma)^{\circ} & \text{if } \sigma(\rho) = \mathsf{c} \end{cases}$$

Proof Sketch: By an easy induction on Γ.

Lemma 4.2.3 $\sigma(M[N/x]) = \sigma M[\sigma N/x]$

Proof Sketch: Straightforward induction on M.

Lemma 4.2.4 *If $\sigma P = M[\sigma N/x]$, then $M = \sigma M'$ for some M'.*

Proof Sketch: By straightforward induction on M.

Lemma 4.2.5

$$\frac{\sigma P \to_\beta Q}{(\exists Q') \quad \sigma Q' = Q \quad \wedge \quad P \to_\beta Q'}$$

Proof Sketch: By straightforward induction on the structure of the derivation of $\sigma P \to_\beta Q$. All the congruence cases are easy. In the case where the reduction step is a single β reduction, the proof makes use of Lemma 4.2.3

The next two lemmas state that the EPTS$^{\mathcal{C}}$ conversion judgment subsumes the single-step reduction relation and that it is complete for terms with the same underlying structure.

Lemma 4.2.6 *If $M \to_\beta N$, then $\mathsf{true} \vdash M =_\beta N$.*

Proof Sketch: By induction over the structure of the derivation of $M \to_\beta N$. In the case of a simple β-reduction, use the rule Beta. In any of the congruence cases for the reduction, use the corresponding congruence rule (CongPi, CongLam, or CongApp).

Lemma 4.2.7 (Pre-Completeness of EPTS$^{\mathcal{C}}$ conversion rules)

$$\frac{\sigma M = \sigma N}{(\exists \mathcal{C}) \quad \mathcal{C} \vdash M =_\beta N \quad \wedge \quad \sigma \vDash \mathcal{C}}$$

Proof Sketch: By straightforward induction on σM.

Now we prove soundness and completeness of the EPTS$^{\mathcal{C}}$ conversion rules. The next two theorems say that two EPTS$^{\mathcal{C}}$ terms M and N are provably convertible in EPTS$^{\mathcal{C}}$ under some condition $\mathcal{C}$ satisfied by σ if, and only if, σ instantiates them to β-convertible terms in EPTS.

Theorem 4.2.8 (Soundness of EPTS$^{\mathcal{C}}$ conversion rules)

$$\frac{\mathcal{C} \vdash M =_\beta N \qquad \sigma \vDash \mathcal{C}}{\sigma M =_\beta \sigma N}$$

Proof Sketch: Straightforward induction on the derivation of $\mathcal{C} \vdash M =_\beta N$. The interesting cases are Beta, in which we use Lemma 4.2.3, and the congruence cases, in which we make use of the fact that $\sigma \vDash \alpha = \alpha'$ implies $\sigma\alpha = \sigma\alpha'$. (In fact, the two are logically equivalent).

Theorem 4.2.9 (Completeness of EPTS$^{\mathcal{C}}$ conversion rules)

$$\frac{\sigma M =_\beta \sigma N}{(\exists \mathcal{C}) \quad \mathcal{C} \vdash M =_\beta N \quad \wedge \quad \sigma \vDash \mathcal{C}}$$

Proof: Since $\sigma M =_\beta \sigma N$, there exists a term $\widehat{P}$ such that $\sigma M \rightarrow_\beta^* \widehat{P}$ and $\sigma N \rightarrow_\beta^* \widehat{P}$ (by the Church-Rosser Theorem). By repeated applications of Lemma 4.2.5, there exists P_1 and P_2 such that $\sigma P_1 = \sigma P_2 = \widehat{P}$ and $M \rightarrow_\beta^* P_1$ and $N \rightarrow_\beta^* P_2$. By Lemma 4.2.7, there is some constraint $\mathcal{C}$ such that $\mathcal{C} \vdash P_1 =_\beta P_2$ and $\sigma \vDash \mathcal{C}$. By repeated applications of Lemma 4.2.6, we have $\mathsf{true} \vdash M =_\beta P_1$ and $\mathsf{true} \vdash N =_\beta P_2$. Therefore, by some applications of Symm and Trans, we can derive $\mathcal{C} \vdash M =_\beta N$, and we already know that $\sigma \vDash \mathcal{C}$. □

Finally we prove soundness and completeness of the EPTS$^{\mathcal{C}}$ typing rules. The next two theorems say that an EPTS$^{\mathcal{C}}$ term M is typable in EPTS$^{\mathcal{C}}$ under some condition $\mathcal{C}$ satisfied by σ if and only if σ instantiates M to a well-typed EPTS term.

Theorem 4.2.10 (Soundness of EPTS$^{\mathcal{C}}$ typing rules)

$$\frac{\mathcal{C}\,;\,\Gamma \vdash M :^{\rho} A \qquad \sigma \vDash \mathcal{C}}{\sigma\Gamma \vdash \sigma M :^{\sigma\rho} \sigma A}$$

Proof Sketch: By straightforward induction on typing derivations. The interesting cases are: Var, which makes use of our boolean interpretation of formulas; Conv, which makes use of Lemma 4.2.8; and Reset, which makes use of Lemma 4.2.2.

Theorem 4.2.11 (Completeness of EPTS$^{\mathcal{C}}$ typing rules)

$$\frac{\sigma\Gamma \vdash \sigma M :^{\sigma\rho} \sigma A}{(\exists \mathcal{C}) \quad \mathcal{C}\,;\,\Gamma \vdash M :^{\rho} A \quad \wedge \quad \sigma \vDash \mathcal{C}}$$

Proof Sketch: By straightforward induction on typing derivations. The interesting cases are: Var, which makes use of our boolean interpretation of formulas; Conv, which makes use of Lemma 4.2.8; Π-Elim, which makes use of Lemma 4.2.4; and Reset, which makes use of Lemma 4.2.2.

4.2.4 Logical Structure of Generated Constraints

Now we investigate the logical structure of context annotations and atomic constraints. Recall the form of context annotations.

$$(\textit{context entry annotation}) \quad \gamma ::= \alpha \mid \mathsf{r} \mid \neg\rho \wedge \gamma$$

Each context annotation γ is a conjunction of a base annotation α or r and the negations of zero or more ρs: either $\alpha \wedge \neg\rho_1 \wedge \cdots \wedge \neg\rho_n$ or $\mathsf{r} \wedge \neg\rho_1 \wedge \cdots \wedge \neg\rho_n$.

If the base annotation is r (false) then $\gamma = \mathsf{false}$. Similarly, if any ρ_i is c (true) then $\gamma = \mathsf{false}$. In either case, the atomic constraint $\neg\,\gamma$ equals true. If any ρ_i is r, then that conjunct evaluates to true and may therefore be elided from the overall conjunction. In the remaining case, when the base annotation and each ρ_i are all variables, γ has the form

$$\alpha \wedge \neg\,\alpha_1 \wedge \cdots \wedge \neg\,\alpha_n,$$

and, by De Morgan's laws, the atomic constraint $\neg\,\gamma$ equals

$$\neg\,\alpha \vee \alpha_1 \vee \cdots \vee \alpha_n.$$

In other words, atomic constraints generated by the VAR rule are logically equivalent to either a trivially true constraint or a disjunction of one negated variable and zero or more other variables. Trivially true atomic constraints may be elided from the conjunction forming the overall constraint.

Interestingly, equations between annotation variables can also be expressed as a conjunction of atomic constraints in this form:

$$\begin{aligned} \alpha = \alpha' &= (\alpha \Rightarrow \alpha') \wedge (\alpha' \Rightarrow \alpha) \\ &= (\neg\,\alpha \vee \alpha') \wedge (\neg\,\alpha' \vee \alpha) \end{aligned}$$

We conclude that the constraints generated by the EPTS$^{\mathcal{C}}$ typing rules are logically equivalent to a conjunction of disjunctions of one negated variable with zero or more other variables.

4.2.5 Implementation

The typing rules for PTS are not syntax-directed. This means that the typing rules, when viewed as a program, express a non-deterministic algorithm. Our presentations of EPTS and EPTS$^{\mathcal{C}}$ inherit this aspect of PTS.

Type checking is not decidable for all Pure Type Systems. However, for many Pure Type Systems, there exist algorithmic presentations of the typing rules that are amenable to direct implementation [91].

We believe that a parallel situation holds for Erasure Pure Type Systems. For strongly normalizing functional Pure Type Systems, it should be completely straightforward to derive algorithmic versions of the typing rules for the corresponding EPTS that abstractly specify the behavior of a type-checker. The rules for constraint generation should fit easily into such a type-checker.

One piece missing from the formal development of EPTS$^{\mathcal{C}}$ is a coherence theorem. There may be several ways to prove (i.e., derive) that a particular term M is well-formed in a particular context Γ. Different derivations will, in general, correspond to different constraints. If these different constraints have different optimal solutions, then different portions of M will be marked for erasure in each case. We don't want the (erasure) semantics of a program to depend on the particular way in which it was type-checked. To satisfy ourselves that this cannot happen, we would like to prove something like the following coherence result.

Conjecture 4.2.12 (Coherence)

$$\frac{\mathcal{C}_1 \,;\, \Gamma \vdash M :^{\rho} A \qquad \mathcal{C}_2 \,;\, \Gamma \vdash M :^{\rho} A}{\mathcal{C}_1 \Longrightarrow \mathcal{C}_2}$$

However, this problem is somewhat theoretical, as any language implementation will fix a particular deterministic type-checking algorithm in which typing annotations are checked in a fixed manner. In this situation, coherence is not an issue because there is at most one way in which a program is type-checked, and therefore at most one possible constraint $\mathcal{C}$ that will be generated once the checking algorithm is instrumented to generate constraints on erasure annotations. We are confident that the formal development presented in this chapter will carry over naturally to the algorithmic presentation of the type system underlying such a type-checker.

4.3 CONSTRAINT SOLVING

Now we turn to the problem of solving the constraints generated in the previous section. In general, this is simply the boolean satisfiability problem (SAT) — finding a satisfying assignment σ for a formula ϕ in propositional logic. However, we prefer solutions that assign as many variables to true (c) as possible, so that more of the program is marked for erasure.

4.3.1 Terminology

Modern SAT solvers typically take their input formula in *Conjunctive Normal Form* (CNF) — as a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal is either a propositional variable (a *positive* literal) or the negation of a propositional variable (a *negative* literal). The *negation* $-L$ of a literal L has the same underlying variable but opposite *sign* (positive or negative). An occurrence of a literal L in a formula is called a *positive occurrence* of L and a *negative occurrence* of $-L$. A *unit clause* is a clause consisting of a single literal.

4.3.2 The TOP-SAT Problem

For certain applications some solutions are better than others. We consider the booleans to be totally ordered by setting true $>$ false. This ordering has a minimum element false and extends point-wise to boolean-valued functions (e.g., variable assignments) as follows:

$$\sigma \geq \sigma' \quad \Leftrightarrow \quad \forall \alpha.\ \sigma(\alpha) \geq \sigma'(\alpha)$$

The *Variable Maximizing SAT Problem* (hereafter TOP-SAT[1]) is as follows: Given a formula ϕ in propositional logic, find a solution σ that is maximal in the

[1]A more obvious name choice would be "MAX-SAT", but it already refers to the problem of maximizing the number of satisfied clauses.

point-wise ordering, that is

$$\forall \sigma'. \quad \sigma' \vDash \phi \quad \Rightarrow \quad \sigma \geq \sigma'.$$

A program solving the TOP-SAT problem for ϕ should first indicate whether a maximal solution for ϕ exists and, if so, give the solution.

In terms of erasure annotations, a maximal solution sets more annotations to c than any other, and therefore marks as much of a program for erasure as possible.

4.3.3 An Algorithm for our Special Case

The constraints generated by $\text{EPTS}^{\mathcal{C}}$ are in CNF with the special property that each clause contains exactly one negative literal. In this case, there is an efficient algorithm for TOP-SAT.

Let ϕ be a propositional logic formula in CNF with the property that each clause in ϕ contains exactly one negative literal.

$$\phi = (\neg\, \alpha_1 \vee \varphi_1) \wedge (\neg\, \alpha_2 \vee \varphi_2) \wedge \cdots \wedge (\neg\, \alpha_N \vee \varphi_N)$$

(each φ_i is a (possibly empty) disjunction of positive literals). Notice that assigning all variables to false in this situation satisfies ϕ, though (likely) not optimally.

Definition 4.3.1 (The Algorithm) 1. ***Unit Clause Propagation.*** *While ϕ contains a unit clause L, assign $L = \mathsf{true}$ and then simplify ϕ — Remove from ϕ all clauses with positive occurrences of L and remove all negative occurrences of L in other clauses.*

2. ***Completion.*** *When no unit clauses are left, assign all remaining unassigned variables to true.*

Lemma 4.3.2 (Invariant) *Each step of Unit Clause Propagation preserves the invariant that all clauses in ϕ contain exactly one negative literal.*

Proof: Assuming every clause in ϕ contains a single negative literal, the unit clause that we propagate must consist of a single negative literal $\neg\,\alpha$. We assign this literal to true (by setting α to false) and simplify. Every clause containing $\neg\,\alpha$ will be removed and every occurrence of α will be removed from its clause. Each remaining clause still contains its sole negative literal because only positive literals were removed from any (surviving) clause. □

Lemma 4.3.3 (Correctness of Step 1) *If a Unit Clause Propagation step takes ϕ to ϕ', then any TOP-SAT solution of ϕ' is uniquely extensible to a TOP-SAT solution for ϕ.*

Proof: Because ϕ is a conjunction containing a unit clause $\neg\,\alpha$, any assignment satisfying ϕ must set α to false. Let σ' be some assignment satisfying ϕ'. Then σ' satisfies every clause in ϕ that was not removed, because each such clause is logically weaker than its corresponding clause in ϕ'. The extended assignment $\sigma'[\mathsf{false}/\alpha]$ also satisfies the clauses that were removed from ϕ. Because σ' maximizes the number of non-α variables set to true in an assignment satisfying ϕ', so does $\sigma'[\mathsf{false}/\alpha]$ for ϕ, because we may not choose $\alpha = \mathsf{true}$ and still satisfy ϕ. □

Lemma 4.3.4 (Correctness of Step 2) *If ϕ contains no unit clauses and each clause in ϕ contains exactly one negative literal, then $\lambda\alpha.\,\mathsf{true}$ is the maximal satisfying assignment for ϕ.*

Proof: In this case, ϕ is of the form

$$(\neg\,\alpha_1 \vee \varphi_1) \wedge (\neg\,\alpha_2 \vee \varphi_2) \wedge \cdots \wedge (\neg\,\alpha_N \vee \varphi_N)$$

where each φ_i is a disjunction of positive literals. Because ϕ contains no unit clauses, each φ_i is non-empty. Let σ be the assignment $\lambda\alpha.\,\mathsf{true}$. Then $\sigma(\varphi_i) = \mathsf{true}$

because φ_i is non-empty and contains positive literals. Therefore σ satisfies ϕ

$$\begin{aligned}\sigma(\phi) &= \sigma\left(\bigwedge_i \neg\,\alpha_i \vee \varphi_i\right) &&= \bigwedge_i \sigma(\neg\,\alpha_i) \vee \sigma(\varphi_i)\\ &&&= \bigwedge_i \mathsf{false} \vee \mathsf{true}\\ &&&= \mathsf{true}\end{aligned}$$

and is clearly the maximum solution. □

Theorem 4.3.5 (Correctness) *If each clause in ϕ contains exactly one negative literal, then this algorithm returns the maximal assignment satisfying ϕ.*

Proof: By Lemma 4.3.2, each step of Unit Clause Propagation preserves the invariant that each clause contains exactly one negative literal. When the Unit Clause Propagation loop finishes, any remaining clauses are of size ≥ 2 and the invariant holds, so, by Lemma 4.3.4, setting all as-yet-undetermined variables to true maximally satisfies the remaining formula. By Lemma 4.3.3, this solution extends to a maximal solution of the original ϕ. □

Discussion Unit clause propagation can be explained in terms of erasure annotations, Recall the cause of a typical phase-ordering constraint $\alpha_1 \vee \cdots \vee \alpha_N \vee \neg\,\alpha$ from Figure 4.2. A unit clause $\neg\,\alpha$ corresponds to a variable occurrence for which every enclosing α_i in its scope has been determined to equal r, and therefore α must be r. The process is initiated by occurrences of λ-bound variables that do not appear inside any @-arguments (or domain annotations) in their scope (i.e., $N = 0$).

In this way, the algorithm deduces which annotations must be r. When no more annotations can be deduced to equal r, we set all remaining variables to c. The algorithm discussed here calculates a sort of greatest fixed-point. Contrast this to the informal least fixed-point algorithm outlined in Section 1.4.2.

4.3.4 Partial Annotation

It may be useful for programmers to have the ability to annotate some parts of their program without having to annotate everything. In this case, we would like to run the erasability analysis on partially annotated programs and fill in the remainder of the unspecified annotations in an optimal way. The analysis algorithm stated thus far can be extended to handle this case.

As for constraint generation, we need to add in extra equations of the form $\alpha_1 = \mathsf{r}$ and $\alpha_2 = \mathsf{c}$ for positions in the term with user-provided annotations. Under our boolean interpretation, these constraints are simply unit clauses $\neg\,\alpha_1$ and α_2, respectively. Admitting clauses of this second form violates the invariant that each clause has exactly one negative literal.

In this case, we instead maintain the invariant that each clause has *at most* one negative literal. The proofs of Lemmas 4.3.2, 4.3.3, and 4.3.4 may all be extended to this more general case. The only difference is that now the algorithm may fail to find a satisfying assignment. This is because user-provided annotations may be inconsistent. For example, the constraint $\neg\,\alpha_1 \wedge \alpha_2 \wedge (\alpha_1 = \alpha_2)$ has no solution.

In terms of unit clause propagation, this inconsistency manifests itself in a clause being simplified to the point that it becomes empty and therefore false. This could not happen before because every clause always had at least one literal, namely the negative one. In the example just given, the constraint expands to $\neg\,\alpha_1 \wedge \alpha_2 \wedge (\neg\,\alpha_1 \vee \alpha_2) \wedge (\alpha_1 \vee \neg\,\alpha_2)$, in which the fourth clause will become empty after doing unit propagation on the first two clauses.

4.3.5 Implementation

Modern SAT solvers rely heavily on unit clause propagation and use clever data structures to implement it efficiently. We have used these same techniques to implement a constraint solver.

In a naive implementation of unit propagation, we maintain for each literal L a list of all the clauses in which it appears. Any time a literal L is set to false, we visit each clause it appears in to check if that clause has become a unit clause.

The designers of the Chaff SAT solver [68] pioneered a technique called *two watched literals*. Their insight was that we need not visit a clause of original length n to check if it has become unit until it changes from size $n-2$ to $n-1$ and this can never happen as long as there are at least two unassigned literals in the clause. This insight leads to an implementation where we pick two unassigned literals in each clause to watch. Now we maintain for each literal a list of all the clauses in which it is *watched*, rather than a list of all the clauses in which it appears. When a literal L is set to false, we need only visit the clauses in which it is watched to see if that clause has become a unit clause. Any other clause C in which L appears but is not watched cannot become unit by assigning $L = \mathsf{false}$, because C still contains two unassigned watched literals. This change of implementation greatly speeds up unit clause propagation in general.

When doing erasability analysis on a partially-annotated program, contradictory constraints may arise due to inconsistent annotations. In this case, the program analyzer should give the user some feedback about which annotations are inconsistent. In our situation, each clause of the SAT formula comes from a particular variable occurrence in the program. We would like to list the variable occurrences that are to blame in case of an error. This requires some sort of conflict explanation facility in the SAT algorithm. Fortunately for us, modern SAT solvers do *clause-based learning*, a process that relies on exactly the sort of explanation facility that we require.

Whenever a SAT solver makes inconsistent guesses about propositional variables, it will derive a contradiction and then backtrack to a consistent state by "un-guessing" some previous guesses. To ensure that it doesn't end up deriving the same sorts of contradictions over and over again, the solver can analyze the

contradiction to see which of the guesses it made were contradictory. Let's say it finds that of all the current guesses it has made, only three are responsible for the contradiction: $\alpha_6 = \mathsf{true}$, $\alpha_{47} = \mathsf{false}$, and $\alpha_{99} = \mathsf{true}$. The fact that these lead to a contradiction means that the formula $\phi = \alpha_6 \wedge \neg\alpha_{47} \wedge \alpha_{99} \Longrightarrow \mathsf{false}$ is a consequence of the overall formula we're trying to satisfy. Therefore ϕ may be rewritten as the clause $\neg\alpha_6 \vee \alpha_{47} \vee \neg\alpha_{99}$ and added to the overall formula. The addition of this *learned clause* will keep the SAT solver out of this particular contradictory corner of the search space in the future.

The contradiction inspection mechanism of a SAT solver is easily adapted to find all the clauses that are to blame for a contradiction. Because each clause arises from a particular variable occurrence in the source program, this information may be used to generate intelligent error messages outlining which variable occurrences in a program caused the phase error.

We have implemented a prototype constraint solver weighing in at under 250 lines of OCaml[2]. Following the ideas of this section, our implementation uses common SAT algorithms and data structures and supports conflict explanation.

4.4 CONCLUSIONS

We have developed a two-phase constraint generation and solving strategy for determining optimal erasure annotations for PTS terms. The constraint-generation scheme is sound and complete with respect to the EPTS type system that checks (among other things) correctness of erasure annotations. Though our presentation of EPTS$^{\mathcal{C}}$ is not algorithmic, it should be straightforward to adapt to any type-checker for a particular PTS. The constraint solver we describe exploits state of the art data structures and algorithms from modern SAT solvers.

Because the erasure annotations resulting from our approach are provably op-

[2] `http://caml.inria.fr/`

timal, programmers need not bother with manual annotation in order to achieve efficient execution of dependently typed programs.

The separation of erasure semantics into two phases leaves open the possibility of programming in either an erasure-oblivious style (in PTS), an erasure-aware style (in EPTS), or anywhere in between (by partially annotating the program).

Chapter 5

INDUCTIVE TYPES

So far, we have developed an erasure semantics for a family of dependently typed λ-calculi. However, programming in such a language would be extremely tedious for any practical application. Two prominent features of modern (statically typed) functional languages that are especially suited to practical applications are *algebraic datatypes* and, conversely, function definition by *pattern matching*.

In this chapter we discuss inductively defined types, the type theorist's version of algebraic datatypes, and the interplay between this language feature and EPTS-style erasure annotations as developed in Chapter 3.

Along the way, we will see how some features that other languages have used for handling non-computational aspects of programming can be expressed using erasure annotations. These examples demonstrate the expressive power of erasure annotations.

Note the following notational conventions used in this chapter: Lower case names like *xs* and *cong* stand for program variables. Upper case names like M and A are meta-variables standing for program terms. Sans-serif names like list and zero are used for all type constructors and data constructors. Also, keywords like data and else are sans-serif and underlined.

In this chapter we follow the precedent set by Cayenne in writing $(x : A) \to B$ instead of $\Pi x{:}A.\, B$. We find this notation more palatable for programming. We omit type annotations when they are inferable from the context, that is we write $\lambda x.\, M$ for $\lambda x{:}A.\, M$ when A is obvious. The following iterated versions of the

syntax are also used: we write $(x, y : A) \to B$ for $(x : A) \to (y : A) \to B$; we write $\lambda x, y{:}A.\, M$ for $\lambda x{:}A.\, \lambda y{:}A.\, M$; and we write $\lambda x, y.\, M$ for $\lambda x{:}A.\, \lambda y{:}B.\, M$.

5.1 INTRODUCTION

Inductively defined types in type theory are similar to algebraic datatypes in the functional languages ML and Haskell. For example, the following declarations

data bool : $*$ where	data nat : $*$ where	data blist : $*$ where
true : bool	zero : nat	bnil : blist
false : bool	succ : nat $\to$ nat	bcons : bool $\to$ blist $\to$ blist

define some commonly used inductive types. Each data declaration defines a *type constructor* t : $*$ and some *data constructors* for constructing inhabitants of type t. Type definitions of this form are known as *algebraic datatypes* in functional programming because they define a free algebra with the constructors playing the role of operators. (We follow the convention of universal algebra by referring to constant constructors such as true or zero as "operations" of arity zero.) Freeness means two things: (1) the constructors are injective (e.g., succ n = succ m implies $n = m$) and (2) they construct distinct values (e.g., for all n, zero $\neq$ succ n). The type t is the smallest type closed under its constructor operations subject to these restrictions.

At run-time, the operational behavior of a constructor is allocation and copying. For example, the evaluation of the expression bcons H T proceeds as follows (assuming a call-by-value implementation):

1. H is evaluated to some value v_h and T is evaluated to some value v_t
2. a region ρ of memory is allocated
3. the values of v_h and v_t are written into ρ along with a "tag" value distinguishing bcons values from bnil values

4. the (address of) memory region ρ is returned as the value of bcons

Furthermore, each inductively defined type comes with its own induction principle. For example, the natural numbers are equipped with the following familiar induction principle:

$$\frac{\vdash n{:}\mathsf{nat} \qquad \vdash P(\mathsf{zero}) \qquad \begin{array}{c} \vdash P(m) \\ \vdots \\ \vdash P(\mathsf{succ}\ m) \end{array}}{\vdash P(n)}$$

This induction principle is made available in the form of an *eliminator* of type

$$\begin{aligned} \mathit{elim}_{\mathsf{nat}} : {} & (n : \mathsf{nat}) \to \\ & (p : \mathsf{nat} \to *) \to \\ & p\ \mathsf{zero} \to \\ & ((m : \mathsf{nat}) \to p\ m \to p\ (\mathsf{succ}\ m)) \to \\ & p\ n. \end{aligned}$$

The type of $\mathit{elim}_{\mathsf{nat}}$ says if, for some predicate p on naturals, one can prove that p holds of zero and is preserved by succ, then p holds of any particular natural n.

In general, the type of $\mathit{elim}_{\mathsf{t}}$ says that every predicate preserved by each data constructor of t holds of every inhabitant of t. The induction principle for t reflects back into the language the knowledge that every inhabitant of type t is formed by finitely many applications of the constructors, and therefore we can always replace any canonical term $M : \mathsf{t}$ with a proof having the same structure as M. For example, if Z and S are terms of types $P\ \mathsf{zero}$ and $(n : \mathsf{nat}) \to P\ n \to P\ (\mathsf{succ}\ n)$ respectively, then

$$\begin{aligned} Z &: P\ (\mathsf{zero}), \\ S\ (\mathsf{zero})\ (Z) &: P\ (\mathsf{succ}\ (\mathsf{zero})), \\ S\ (\mathsf{succ}\ \mathsf{zero})\ (S\ (\mathsf{zero})\ (Z)) &: P\ (\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{zero}))), \end{aligned}$$

and so on. If we ignore (erase) the first argument of each application of S then this sequence becomes.

$$\begin{array}{rcl} Z & : & P\ (\mathsf{zero}) \\ S\ (Z) & : & P\ (\mathsf{succ}\ (\mathsf{zero})) \\ S\ (S\ (Z)) & : & P\ (\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{zero}))) \end{array}$$

The pattern is clear. Replacing each constructor in $t : \mathsf{t}$, with the corresponding proof combinator, we end up with a proof of $P\ t$.

In functional programming, this sort of operation is called a *fold* or *catamorphism*, and is well known to embody the notion of structural recursion. In this setting, $elim_{\mathsf{t}}$ has a more general type than $fold_{\mathsf{t}}$, due to dependent types. In fact one can implement $fold_{\mathsf{t}}$ in terms of $elim_{\mathsf{t}}$ by instantiating $P : \mathsf{t} \to *$ with a constant function $\lambda\_.\, C$, for some type C into which we would like to fold. For example, we can implement $fold_{\mathsf{nat}}$ as follows:

$$\begin{array}{rcl} fold_{\mathsf{nat}} & : & \mathsf{nat} \to (c : *) \to c \to (c \to c) \to c \\ fold_{\mathsf{nat}} & = & \lambda n.\, \lambda c.\, \lambda z.\, \lambda s.\, elim_{\mathsf{nat}}\ n\ (\lambda\_.\, c)\ z\ (\lambda\_.\, \lambda x.\, s\ x) \end{array}$$

The $elim_{\mathsf{t}}$ operator is equipped with reduction rules that are incorporated into the language's notion of definitional equality. For $elim_{\mathsf{nat}}$ the reduction rules are:

$$\begin{array}{lcll} elim_{\mathsf{nat}}\ (\mathsf{zero})\quad P\ Z\ S & \to_\beta & Z & : P\ \mathsf{zero} \\ elim_{\mathsf{nat}}\ (\mathsf{succ}\ N)\ P\ Z\ S & \to_\beta & S\ N\ (elim_{\mathsf{nat}}\ N\ P\ Z\ S) & : P\ (\mathsf{succ}\ N) \end{array}$$

The $elim_{\mathsf{t}}$ operation encapsulates the notion of *pattern matching* on t-constructors as well as the notion of *recursion* over t-values.

A simple example making use of $fold_{\mathsf{nat}}$ is addition.

$$\begin{array}{rcl} plus & : & \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \\ plus & = & \lambda n.\, \lambda m.\, fold_{\mathsf{nat}}\ n\ \mathsf{nat}\ m\ \mathsf{succ} \end{array}$$

The reduction behavior of *plus* is as follows:

$$\begin{array}{lll} \mathit{plus}\ (\mathsf{zero}) & M & \rightarrow^*_\beta \quad M \\ \mathit{plus}\ (\mathsf{succ}\ N)\ M & & =_\beta \quad \mathsf{succ}\ (\mathit{plus}\ N\ M) \end{array}$$

How might one go about proving that $\mathit{plus}\ M\ \mathsf{zero} = M$ for all M? Assuming we have a type $\mathsf{equal} : (a : *) \rightarrow a \rightarrow a \rightarrow *$, we can state this property with the type

$$(n : \mathsf{nat}) \rightarrow \mathsf{equal}\ \mathsf{nat}\ (\mathit{plus}\ n\ \mathsf{zero})\ n.$$

The way to prove this is by induction (i.e., with $\mathit{elim}_{\mathsf{nat}}$). Assuming equal satisfies the following properties

$$\begin{array}{lll} \mathit{refl} & : & (a : *) \rightarrow (x : a) \rightarrow \mathsf{equal}\ a\ x\ x \\ \mathit{cong} & : & (a, b : *) \rightarrow (f : a \rightarrow b) \rightarrow (x, y : a) \rightarrow \mathsf{equal}\ a\ x\ y \rightarrow \mathsf{equal}\ b\ (f\ x)\ (f\ y) \end{array}$$

we may prove the identity as follows:

$$\begin{array}{lll} \mathit{zero\_right\_unit} & : & (n : \mathsf{nat}) \rightarrow \mathsf{equal}\ \mathsf{nat}\ (\mathit{plus}\ n\ \mathsf{zero})\ n \\ \mathit{zero\_right\_unit} & = & \lambda n.\ \mathit{elim}_{\mathsf{nat}}\ n\ (\lambda n.\ \mathsf{equal}\ \mathsf{nat}\ (\mathit{plus}\ n\ \mathsf{zero})\ n) \\ & & \quad (\mathit{refl}\ \mathsf{nat}\ \mathsf{zero}) \\ & & \quad (\lambda m, p.\ \mathit{cong}\ \mathsf{nat}\ \mathsf{nat}\ \mathsf{succ}\ (\mathit{plus}\ m\ \mathsf{zero})\ m\ p) \end{array}$$

The inductive structure of the argument can be seen in the type of the partial application $\mathit{elim}_{\mathsf{nat}}\ n\ (\lambda n.\ \mathsf{equal}\ \mathsf{nat}\ (\mathit{plus}\ n\ \mathsf{zero})\ n)$, namely,

$$\begin{array}{l} \mathsf{equal}\ \mathsf{nat}\ (\mathit{plus}\ \mathsf{zero}\ \mathsf{zero})\ \mathsf{zero} \rightarrow \\ ((m : \mathsf{nat}) \rightarrow \\ \quad \mathsf{equal}\ \mathsf{nat}\ (\mathit{plus}\ m\ \mathsf{zero})\ m \rightarrow \\ \quad \mathsf{equal}\ \mathsf{nat}\ (\mathit{plus}\ (\mathsf{succ}\ m)\ \mathsf{zero})\ (\mathsf{succ}\ m)) \rightarrow \\ \mathsf{equal}\ \mathsf{nat}\ (\mathit{plus}\ n\ \mathsf{zero})\ n \end{array}$$

The base case is proved by reflexivity, because $\mathit{plus}\ \mathsf{zero}\ \mathsf{zero} =_\beta \mathsf{zero}$. The inductive case is proved by congruence, since $\mathit{plus}\ (\mathsf{succ}\ m)\ \mathsf{zero} =_\beta \mathsf{succ}\ (\mathit{plus}\ m\ \mathsf{zero})$.

The *plus* and *zero_right_unit* examples serve to demonstrate that type theory provides a common language for both programming and reasoning about programs. In the case of inductive types, we can use eliminators for both programming primitive recursive functions and reasoning by induction.

5.1.1 Parameterized Inductive Types

The blist datatype is useful, but limited to storing elements of type bool. We may also require lists of other types – lists of naturals, lists of lists of booleans, etc. Rather than define roughly the same type over and over again, we would like to define an entire *family* of list types *parameterized* by the element type. We declare this family of types as follows:

$$\begin{aligned}&\underline{\mathsf{data}}\ \mathsf{plist}\ (a : *) : * \ \underline{\mathsf{where}}\\ &\qquad \mathsf{pnil} : \mathsf{plist}\ a\\ &\qquad \mathsf{pcons} : a \to \mathsf{plist}\ a \to \mathsf{plist}\ a\end{aligned}$$

The types assigned to pnil and pcons fall inside the scope of the parameter $a : *$. However, when we use these constructors in other locations, their types as they appear in this declaration must be parameterized by a in order to make sense. Therefore, the following types are assigned to all subsequent uses of these constructors:

$$\begin{aligned}\mathsf{pnil} &: (a : *) \to \mathsf{plist}\ a\\ \mathsf{pcons} &: (a : *) \to a \to \mathsf{plist}\ a \to \mathsf{plist}\ a\end{aligned}$$

The induction principle for plist is also parameterized by a.

$$\begin{aligned}elim_{\mathsf{plist}} : &(a : *) \to (t : \mathsf{plist}\ a) \to\\ &(p : \mathsf{plist}\ a \to *) \to\\ &p\ (\mathsf{pnil}\ a) \to\\ &((x : a) \to (xs : \mathsf{plist}\ a) \to p\ xs \to p\ (\mathsf{pcons}\ a\ x\ xs)) \to\\ &p\ t.\end{aligned}$$

Compared to the induction principle for blist,

$$
\begin{aligned}
elim_{\mathsf{blist}} : {} & (t : \mathsf{blist}) \to \\
& (p : \mathsf{blist} \to *) \to \\
& p\ \mathsf{bnil} \to \\
& ((x : \mathsf{bool}) \to (xs : \mathsf{blist}) \to p\ xs \to p\ (\mathsf{bcons}\ x\ xs)) \to \\
& p\ t
\end{aligned}
$$

there is an additional argument a to $elim_{\mathsf{plist}}$. The induction principle for $\mathsf{plist}\ a$ is simply $elim_{\mathsf{plist}}\ a$. In this way, the parameterized family of inductively defined types gives rise to a parameterized family of induction principles.

Functional languages such as ML and Haskell have featured parameterized algebraic datatypes since their inception, as this feature goes hand in hand with parametric polymorphism, the cornerstone of their type systems.

5.1.2 Indexed Inductive Types

Dependent types afford us the possibility of *indexing* a type with some data that reveals something of the structure of that type. For example, we can inductively define a type family of boolean lists indexed by their length.

$$
\begin{aligned}
& \underline{\mathsf{data}}\ \mathsf{ilist} : \mathsf{nat} \to * \ \underline{\mathsf{where}} \\
& \quad \mathsf{inil} : \mathsf{ilist}\ \mathsf{zero} \\
& \quad \mathsf{icons} : (n : \mathsf{nat}) \to \mathsf{bool} \to \mathsf{ilist}\ n \to \mathsf{ilist}\ (\mathsf{succ}\ n)
\end{aligned}
$$

The length information in the type $\mathsf{ilist}\ n$ allows us to write more expressive types for list manipulating functions. For example, we may define a function *append* of type

$$(n, m : \mathsf{nat}) \to \mathsf{ilist}\ n \to \mathsf{ilist}\ m \to \mathsf{ilist}\ (plus\ n\ m)$$

and a function *head* of type

$$(n : \mathsf{nat}) \to \mathsf{ilist}\ (\mathsf{succ}\ n) \to \mathsf{bool}.$$

This extra information in types leads to safer programs. For example, we are assured that *head* will never give rise to the run-time error of taking the head of an empty list, because such an application of *head* is ill-typed.

What induction principle do we get for an inductive family of types? The induction principle for the ilist family is

$$
\begin{aligned}
elim_{\mathsf{ilist}} : {} & (n : \mathsf{nat}) \to (t : \mathsf{ilist}\ n) \to \\
& (p : (n : \mathsf{nat}) \to \mathsf{ilist}\ n \to *) \to \\
& p\ \mathsf{zero}\ \mathsf{inil} \to \\
& ((m : \mathsf{nat}) \to (x : \mathsf{bool}) \to (xs : \mathsf{ilist}\ m) \to \\
& \qquad p\ m\ xs \to p\ (\mathsf{succ}\ m)\ (\mathsf{icons}\ m\ x\ xs)) \to \\
& p\ n\ t.
\end{aligned}
$$

The difference between this induction principle and $elim_{\mathsf{blist}}$ is that p is now a predicate over both a ilist *and its length*.

Functional programming languages have just recently begun to explore the idea of inductive families of types. The idea has been introduced by several researchers under several names: Xi's "guarded algebraic datatypes" [96], Hinze and Cheney's "first-class phantom types" [18], Sheard's "equality-qualified types" [83]. The functional programming community has converged on the term "generalized algebraic datatypes" (GADTs) for the core idea unifying these various approaches. Recently, the Glasgow Haskell Compiler (GHC), probably the most widely used Haskell compiler, added support for GADTs [72].

5.1.3 Parameterized Indexed Inductive Types

The same type can have both parameters and indices. To complete our running example, we define a list type that is both indexed by length and parameterized

by element type.

$$\begin{aligned}&\underline{\mathsf{data}}\ \mathsf{list}\ (a : *) : \mathsf{nat} \to * \ \underline{\mathsf{where}}\\ &\quad \mathsf{nil} : \mathsf{list}\ a\ \mathsf{zero}\\ &\quad \mathsf{cons} : (n : \mathsf{nat}) \to a \to \mathsf{list}\ a\ n \to \mathsf{list}\ a\ (\mathsf{succ}\ n)\end{aligned}$$

The clause $\mathsf{list}\ (a : *) : \mathsf{nat} \to *$ declares list to have type $* \to \mathsf{nat} \to *$ where the first argument is a parameter and the second an index. The mediating colon after serves to partition the arguments of list into parameters (to the left) and indices (to the right).

Note how the index argument changes from one occurrence of list to the next (becoming zero, n, and $\mathsf{succ}\ n$ in different places), but the parameter argument is used uniformly (always simply a). In fact, it is a requirement that any parameter in any inductive type t must be used uniformly in the definition across all recursive occurrences of t.

The eliminator for list is parameterized and the induction hypothesis is indexed.

$$\begin{aligned}elim_{\mathsf{list}} : {} & (a : *) \to (m : \mathsf{nat}) \to (t : \mathsf{list}\ a\ m) \to\\ & (p : (n : \mathsf{nat}) \to \mathsf{list}\ a\ n \to *) \to\\ & p\ \mathsf{zero}\ (\mathsf{nil}\ a) \to\\ & ((n : \mathsf{nat}) \to (x : a) \to (xs : \mathsf{list}\ a\ n) \to\\ & \quad p\ n\ xs \to p\ (\mathsf{succ}\ n)\ (\mathsf{cons}\ a\ x\ xs)) \to\\ & p\ m\ t\end{aligned}$$

Another common and useful example of a type that is both parameterized and indexed is the equality type (a.k.a. the identity type).

$$\begin{aligned}&\underline{\mathsf{data}}\ \mathsf{equal}\ (a : *)\ (x : a) : a \to * \ \underline{\mathsf{where}}\\ &\quad \mathsf{refl} : \mathsf{equal}\ a\ x\ x\end{aligned}$$

In this case, the parameters are a type a and one of its inhabitants x and the index is another a-object. The indexed type defined with these parameters can be

thought of as the predicate "is equal to x" on a-objects. The single constructor refl simply states reflexivity of equality: x is equal to x.

Outside this definition, the parameterization must be made explicit, so refl is assigned the following type:

$$\mathsf{refl} \quad : \quad (a : *) \to (x : a) \to \mathsf{equal}\ a\ x\ x$$

Using the eliminator for equal, one can define functions of the following types

$$\begin{aligned} \mathit{symm} \quad &: \quad (a : *) \to (x, y : a) \to \mathsf{equal}\ a\ x\ y \to \mathsf{equal}\ a\ y\ x \\ \mathit{trans} \quad &: \quad (a : *) \to (x, y, z : a) \to \mathsf{equal}\ a\ x\ y \to \mathsf{equal}\ a\ y\ z \to \mathsf{equal}\ a\ x\ z \end{aligned}$$

thereby demonstrating that equal is symmetric and transitive as well as reflexive.

5.2 OPPORTUNITIES FOR ERASURE

Now we consider a type theory with both erasure annotations (as in EPTS) and inductive types. How do these two language features interact?

First a small point on notation. The use of erasure annotations used in previous chapters is concise and close to what one would use for abstract syntax in an implementation, but it is overly cumbersome as the surface syntax for a programming language, as it involves annotations on nearly every single syntactic construct. Table 5.1 introduces the concrete syntax we will use in this chapter and relates it to (1) the abstract syntax used up to this point and (2) a possible rendering in ASCII.

Compile-time application with @ has the same precedence and associativity as run-time application by juxtaposition. For example, the application $f\ w\ @x\ y\ @z$ is correctly parsed as $(((f\ w)\ @x)\ y)\ @z$ rather than $((f\ w)\ @(x\ y))\ @z$. The @ symbol is simply an optional annotation preceding the argument in an application.

mode	abstract syntax	concrete syntax	ASCII rendering
relevant	$\Pi^{\mathsf{r}} x{:}A.\, B$	$(x{:}A) \to B$	`(x:A) -> B`
	$A \xrightarrow{\mathsf{r}} B$	$A \to B$	`A -> B`
	$\lambda^{\mathsf{r}} x{:}A.\, M$	$\lambda x{:}A.\, M$	`\x:A. M`
	$M @^{\mathsf{r}} N$	$M\ N$	`M N`
irrelevant	$\Pi^{\mathsf{c}} x{:}A.\, B$	$(x{:}A) \Rightarrow B$	`(x:A) => B`
	$A \xrightarrow{\mathsf{c}} B$	$A \Rightarrow B$	`A => B`
	$\lambda^{\mathsf{c}} x{:}A.\, M$	$\lambda\!\!\lambda x{:}A.\, M$	`\\x:A. M`
	$M @^{\mathsf{c}} N$	$M @ N$	`M @N`

Table 5.1: Concrete syntax for erasure annotations

5.2.1 Eliminator Argument Erasure

Since the reduction rules of an inductive type are tied up with applications of its eliminator, we first investigate the computational relevance of eliminator arguments.

In order to fruitfully discuss eliminator arguments, we follow the terminology of Conor McBride in naming various categories of eliminator arguments [59]. Consider the eliminator for list, the type of lists indexed by their length and parameterized by their element type.

$$
\begin{aligned}
\mathit{elim}_{\mathsf{list}} : {} & (a : *) \to (n : \mathsf{nat}) \to (t : \mathsf{list}\ a\ n) \to \\
& (p : \mathsf{list}\ a\ \mathsf{zero} \to *) \to \\
& (m_{\mathsf{nil}} : p\ \mathsf{nil}) \to \\
& (m_{\mathsf{cons}} : (m : \mathsf{nat}) \to (x : \mathsf{bool}) \to (xs : \mathsf{list}\ a) \to \\
& \qquad p\ xs \to p\ (\mathsf{cons}\ m\ x\ xs)) \to \\
& p\ t
\end{aligned}
$$

We categorize the arguments of this eliminator as follows:

- Argument a is a *parameter* of the type to be eliminated, and argument n is an *index* of the type to be eliminated.
- Argument t is the *target* as its type is the one we are eliminating. Together with any parameters and indices, the target states *what* is being eliminated.
- Argument p is the *motive* as it states the knowledge we stand to gain by the elimination. The motive states *why* the target is being eliminated.
- Arguments m_{nil} and m_{cons} are the *methods* by which each data constructor is destructed during the elimination. The methods state *how* the target may be eliminated.

We will now consider each category of eliminator arguments in turn, investigating which arguments are relevant to the computation of the eliminator.

Motives

Consider the eliminator for natural numbers discussed in Section 5.1.

$$\begin{aligned} \mathit{elim}_{\mathsf{nat}} : {} & (n : \mathsf{nat}) \to \\ & (p : \mathsf{nat} \to *) \to \\ & p\ \mathsf{zero} \to \\ & ((n : \mathsf{nat}) \to p\ n \to p\ (\mathsf{succ}\ n)) \to \\ & p\ n \end{aligned}$$

Recall that this eliminator has the following computational behavior:

$$\begin{aligned} \mathit{elim}_{\mathsf{nat}}\ (\mathsf{zero}) \quad P\ Z\ S &\to_\beta Z \\ \mathit{elim}_{\mathsf{nat}}\ (\mathsf{succ}\ N)\ P\ Z\ S &\to_\beta S\ N\ (\mathit{elim}_{\mathsf{nat}}\ N\ P\ Z\ S) \end{aligned}$$

It is apparent that the motive P plays no role in the computation of $\mathit{elim}_{\mathsf{nat}}$, as it only appears on the right-hand side in the same computationally irrelevant

argument position in which it started on the left hand side. Therefore, in a language with erasure annotations, we may assign $elim_{\mathsf{nat}}$ the following more precise type:

$$\begin{aligned} elim_{\mathsf{nat}} : {} & (n : \mathsf{nat}) \to \\ & (p : \mathsf{nat} \to *) \Rightarrow \\ & p\ \mathsf{zero} \to \\ & ((n : \mathsf{nat}) \to p\ n \to p\ (\mathsf{succ}\ n)) \to \\ & p\ n \end{aligned}$$

At run-time (i.e., after erasure), the computation rules for nat then become

$$\begin{aligned} elim_{\mathsf{nat}}\ (\mathsf{zero})\ \ Z\ S\ & \to_\beta\ Z \\ elim_{\mathsf{nat}}\ (\mathsf{succ}\ N)\ Z\ S\ & \to_\beta\ S\ N\ (elim_{\mathsf{nat}}\ N\ Z\ S) \end{aligned}$$

This observation holds in general for other types besides nat — the motive argument to an eliminator is computationally irrelevant to the execution of that eliminator and the eliminator's type should be strengthened to reflect this fact.

Parameters and Indices

Recall the parameterized family of list types.

$$\begin{aligned} \underline{\mathsf{data}}\ \mathsf{plist}\ (a : *) : * \ \underline{\mathsf{where}} \\ \mathsf{pnil} : \mathsf{plist}\ a \\ \mathsf{pcons} : a \to \mathsf{plist}\ a \to \mathsf{plist}\ a \end{aligned}$$

The eliminator for plist has the following type (given the previous improvement regarding the motive p):

$$\begin{aligned} elim_{\mathsf{plist}} : {} & (a : *) \to (t : \mathsf{plist}\ a) \to \\ & (p : \mathsf{plist}\ a \to *) \Rightarrow \\ & p\ \mathsf{pnil} \to \\ & ((x : \mathsf{bool}) \to (xs : \mathsf{plist}\ a) \to p\ xs \to p\ (\mathsf{pcons}\ x\ xs)) \to \\ & p\ t \end{aligned}$$

and the following post-erasure computational behavior:

$$\begin{array}{lll} elim_{\mathsf{plist}}\ A\ (\mathsf{pnil}\ A) & N\ C & \rightarrow_\beta\ N \\ elim_{\mathsf{plist}}\ A\ (\mathsf{pcons}\ A\ H\ T)\ N\ C & & \rightarrow_\beta\ C\ H\ T\ (elim_{\mathsf{plist}}\ A\ T\ N\ C) \end{array}$$

Note that the parameter argument A is irrelevant to the computation of $elim_{\mathsf{plist}}$. Therefore, we again increase the precision of the eliminator's type.

$$\begin{array}{l} elim_{\mathsf{plist}} : (a : *) \Rightarrow (t : \mathsf{plist}\ a) \rightarrow \\ \qquad (p : \mathsf{plist}\ a \rightarrow *) \Rightarrow \\ \qquad p\ \mathsf{pnil} \rightarrow \\ \qquad ((x : \mathsf{bool}) \rightarrow (xs : \mathsf{plist}\ a) \rightarrow p\ xs \rightarrow p\ (\mathsf{pcons}\ x\ xs)) \rightarrow \\ \qquad p\ t. \end{array}$$

The run-time behavior then becomes:

$$\begin{array}{lll} elim_{\mathsf{plist}}\ (\mathsf{pnil}\ A) & N\ C & \rightarrow_\beta\ N \\ elim_{\mathsf{plist}}\ (\mathsf{pcons}\ A\ H\ T)\ N\ C & & \rightarrow_\beta\ C\ H\ T\ (elim_{\mathsf{plist}}\ T\ N\ C) \end{array}$$

Since (1) the A component of both pnil and pcons objects is not needed by the eliminator, and (2) the eliminator is the primitive provided by the language for inspection of plist values, we conclude that these A components are not necessary to store inside the representation of plist objects. As we will see in Section 5.2.2, the way to omit certain constructor arguments from the run-time representation of objects is to update the type of the constructor so that those arguments are marked as computationally irrelevant. For example, we now update the type assignment of constructors pnil and pcons as follows:

$$\begin{array}{rcl} \mathsf{pnil} & : & (a : *) \Rightarrow \mathsf{plist}\ a \\ \mathsf{pcons} & : & (a : *) \Rightarrow a \rightarrow \mathsf{plist}\ a \rightarrow \mathsf{plist}\ a \end{array}$$

The post-erasure run-time behavior of $elim_{\mathsf{plist}}$ then becomes

$$\begin{array}{lll} elim_{\mathsf{plist}}\ (\mathsf{pnil}) & N\ C & \rightarrow_\beta\ N \\ elim_{\mathsf{plist}}\ (\mathsf{pcons}\ H\ T)\ N\ C & & \rightarrow_\beta\ C\ H\ T\ (elim_{\mathsf{plist}}\ T\ N\ C). \end{array}$$

Now let us consider indexed type families, such as ilist, the family of length-indexed list types. The eliminator for ilist has the type

$$\begin{aligned}
\mathit{elim}_{\mathsf{ilist}} : {} & (n : \mathsf{nat}) \to (t : \mathsf{ilist}\ n) \to \\
& (p : (n : \mathsf{nat}) \to \mathsf{ilist}\ n \to *) \Rightarrow \\
& p\ \mathsf{zero}\ \mathsf{inil} \to \\
& ((n : \mathsf{nat}) \to (x : \mathsf{bool}) \to (xs : \mathsf{ilist}\ n) \to \\
& \qquad p\ n\ xs \to p\ (\mathsf{succ}\ n)\ (\mathsf{icons}\ n\ x\ xs)) \to \\
& p\ n\ t.
\end{aligned}$$

The computation rules for $\mathit{elim}_{\mathsf{ilist}}$ are as follows:

$$\mathit{elim}_{\mathsf{ilist}}\ (\mathsf{zero})\ (\mathsf{inil})\ @P\ M_n\ M_c \quad \to_\beta \quad M_n$$

$$\begin{aligned}
\mathit{elim}_{\mathsf{ilist}}\ (\mathsf{succ}\ N)\ (\mathsf{icons}\ N\ H\ T)\ @P\ M_n\ M_c & \\
\to_\beta \quad M_c\ N\ H\ T\ (\mathit{elim}_{\mathsf{ilist}}\ N\ T\ @P\ M_n\ M_c) &
\end{aligned}$$

If we view these rules as pattern-matching equations, the second one that matches against the constructors succ and icons has the undesirable property of reusing the meta-variable N. However, this does not mean that we need to test the two occurrences of N for equality in order to proceed with the match. Indeed, the fact that the constructor icons is found as the top constructor of the target argument ensures that the two occurrences of N are the same (i.e., definitionally equal) whenever the left-hand side is well-typed.

Brady et al. make the same observation [14]. Furthermore, they note that the entire succ N argument is determined by the constructor icons. Similarly, the constructor inil in the first reduction rule determines the preceding argument zero, so that $\mathit{elim}_{\mathsf{ilist}}$ need not inspect its first argument at all after inspecting the target. In our terminology, the index argument to $\mathit{elim}_{\mathsf{ilist}}$ is computationally irrelevant. Therefore, the type of $\mathit{elim}_{\mathsf{ilist}}$ may be strengthened as in the case of $\mathit{elim}_{\mathsf{plist}}$ by changing the annotation on the first argument from $(n : \mathsf{nat}) \to$ to $(n : \mathsf{nat}) \Rightarrow$.

Again, these observations hold in general — parameters of families of inductive types are computationally irrelevant as arguments of both eliminators and constructors, and indices of inductive families of types are computationally irrelevant as eliminator arguments. Using erasure annotations, the types of eliminators and constructors can and should be strengthened to reflect these facts.

Methods and the Target

No matter the type, the target argument and the method arguments of the eliminator are computationally relevant. This is because each reduction rule for an eliminator inspects the top-most constructor of the target argument and, based on what constructor it finds, executes the corresponding method with the arguments of the constructor and the results of any necessary recursive calls to the eliminator.

Two apparent exceptions to this rule are discussed in Section 5.2.3.

5.2.2 Constructor Argument Erasure

What happens if we use the computationally-irrelevant function space when assigning types to constructors? For example, we might redefine length-indexed lists as follows:

$$\begin{array}{l}\underline{\mathsf{data}}\ \mathsf{ilist} : \mathsf{nat} \to * \ \underline{\mathsf{where}} \\ \qquad \mathsf{inil} : \mathsf{ilist}\ \mathsf{zero} \\ \qquad \mathsf{icons} : (n : \mathsf{nat}) \Rightarrow \mathsf{bool} \to \mathsf{ilist}\ n \to \mathsf{ilist}\ (\mathsf{succ}\ n)\end{array}$$

The difference between this definition and the original one is that icons has been assigned the type

$$(n : \mathsf{nat}) \Rightarrow \mathsf{bool} \to \mathsf{ilist}\ n \to \mathsf{ilist}\ (\mathsf{succ}\ n)$$

in which $n : \mathsf{nat}$ is declared to be irrelevant to the computation of icons.

What does it mean to say that icons does not depend computationally on one of its arguments? Recall from Section 5.1 the operational description of how

constructors evaluate: icons allocates some memory and writes some values into that memory. If icons does not depend computationally on the value of n : nat then that must mean that n is not written into the memory allocated for representing a icons object.

This interpretation is consistent with the definition of the erasure translation, in which arguments marked for erasure are simply discarded. In this case, non-computational arguments to a constructor are simply not part of the representation of data constructed by that constructor. For example, the list

$$\mathsf{icons}\ @2\ \mathsf{true}\ (\mathsf{icons}\ @1\ \mathsf{false}\ (\mathsf{icons}\ @0\ \mathsf{true}\ \mathsf{inil}))$$

(presented using syntactic sugar for natural number literals) is erased to

$$\mathsf{icons}\ \mathsf{true}\ (\mathsf{icons}\ \mathsf{false}\ (\mathsf{icons}\ \mathsf{true}\ \mathsf{inil}))$$

which corresponds to the way icons will store things in memory.

Given the optimizations discussed so far, the original eliminator for ilist has the type

$$\begin{aligned}
&\mathit{elim}_{\mathsf{ilist}} : (n : \mathsf{nat}) \Rightarrow (t : \mathsf{ilist}\ n) \rightarrow \\
&\qquad (p : (n : \mathsf{nat}) \rightarrow \mathsf{ilist}\ n \rightarrow *) \Rightarrow \\
&\qquad p\ \mathsf{zero}\ \mathsf{inil} \rightarrow \\
&\qquad ((n : \mathsf{nat}) \rightarrow (x : \mathsf{bool}) \rightarrow (xs : \mathsf{ilist}\ n) \rightarrow \\
&\qquad\qquad p\ n\ xs \rightarrow p\ (\mathsf{succ}\ n)\ (\mathsf{icons}\ n\ x\ xs)) \rightarrow \\
&\qquad p\ n\ t.
\end{aligned}$$

and the pre-erasure computation rule for $\mathit{elim}_{\mathsf{ilist}}$ on icons is

$$\begin{aligned}
&\mathit{elim}_{\mathsf{ilist}}\ @(\mathsf{succ}\ N)\ (\mathsf{icons}\ @N\ H\ T)\ @P\ M_n\ M_c \\
&\rightarrow_\beta \quad M_c\ N\ H\ T\ (\mathit{elim}_{\mathsf{ilist}}\ @N\ T\ @P\ M_n\ M_c).
\end{aligned}$$

The post-erasure version of this computation rule is

$$\mathit{elim}_{\mathsf{ilist}}\ (\mathsf{icons}\ H\ T)\ M_n\ M_c \quad \rightarrow_\beta \quad M_c\ N\ H\ T\ (\mathit{elim}_{\mathsf{ilist}}\ T\ M_n\ M_c).$$

However, this rule exhibits a phase error. On the right-hand side of this rule an N has appeared out of nowhere, rendering the rule non-deterministic and broken.

We may fix the rule by ensuring that M_c not require at run-time that which is not available. This is achieved by updating the type of M_c from

$$((n : \mathsf{nat}) \to (x : \mathsf{bool}) \to (xs : \mathsf{ilist}\ n) \to p\ n\ xs \to p\ (\mathsf{succ}\ n)\ (\mathsf{icons}\ n\ x\ xs))$$

to

$$((n : \mathsf{nat}) \Rightarrow (x : \mathsf{bool}) \to (xs : \mathsf{ilist}\ n) \to p\ n\ xs \to p\ (\mathsf{succ}\ n)\ (\mathsf{icons}\ n\ x\ xs))$$

(Note the change in the relevance of $n : \mathsf{nat}$). Then the type system will guarantee that M_c does not depend computationally on n (i.e., N) and the rule will erase to the phase-correct rule

$$elim_{\mathsf{ilist}}\ (\mathsf{icons}\ H\ T)\ M_n\ M_c \quad \to_\beta \quad M_c\ H\ T\ (elim_{\mathsf{ilist}}\ T\ M_n\ M_c).$$

Therefore the ultimate type of the eliminator for ilist is

$$\begin{aligned} elim_{\mathsf{ilist}} : {} & (n : \mathsf{nat}) \Rightarrow (t : \mathsf{ilist}\ n) \to \\ & (p : (n : \mathsf{nat}) \to \mathsf{ilist}\ n \to *) \Rightarrow \\ & p\ \mathsf{zero}\ \mathsf{inil} \to \\ & ((n : \mathsf{nat}) \Rightarrow (x : \mathsf{bool}) \to (xs : \mathsf{ilist}\ n) \to \\ & \qquad p\ n\ xs \to p\ (\mathsf{succ}\ n)\ (\mathsf{icons}\ n\ x\ xs)) \to \\ & p\ n\ t. \end{aligned}$$

One further possibility is that a recursive argument to a constructor is declared computationally irrelevant. Consider the following (somewhat contrived) variant of the natural numbers:

$$\begin{aligned} &\underline{\mathsf{data}}\ \mathsf{bnat} : * \ \underline{\mathsf{where}} \\ &\qquad \mathsf{bzero} : \mathsf{bnat} \\ &\qquad \mathsf{bsucc} : \mathsf{bnat} \Rightarrow \mathsf{bnat} \end{aligned}$$

This type is a strange hybrid of the natural number type (at compile-time) and the boolean type (at run-time) because the sole argument of bsucc is always marked for erasure.

At this point, the reduction rule for $elim_{\mathsf{bnat}}$ on bsucc is

$$elim_{\mathsf{bnat}}\ (\mathsf{bsucc}\ @N)\ @P\ Z\ S \rightarrow_\beta S\ @N\ (elim_{\mathsf{bnat}}\ N\ @P\ Z\ S)$$

which erases to

$$elim_{\mathsf{bnat}}\ (\mathsf{bsucc})\ Z\ S \rightarrow_\beta S\ (elim_{\mathsf{bnat}}\ N\ Z\ S).$$

This final rule is phase-incorrect due to the occurrence of N on the right-hand side.

We cannot arrange for the N argument to $elim_{\mathsf{bnat}}$ to be erased, because eliminators always depends computationally on their target argument. Instead, our only recourse is to force S not to depend computationally on the result of the recursive call. This solution leads to the following type for $elim_{\mathsf{bnat}}$

$$\begin{aligned} elim_{\mathsf{bnat}} : {} & (n : \mathsf{bnat}) \rightarrow \\ & (p : \mathsf{bnat} \rightarrow *) \Rightarrow \\ & p\ \mathsf{bzero} \rightarrow \\ & ((n : \mathsf{bnat}) \Rightarrow p\ n \Rightarrow p\ (\mathsf{bsucc}\ n)) \rightarrow \\ & p\ n \end{aligned}$$

and the following pre-erasure

$$elim_{\mathsf{bnat}}\ (\mathsf{bsucc}\ @N)\ @P\ Z\ S \rightarrow_\beta S\ @N\ @(elim_{\mathsf{bnat}}\ N\ @P\ Z\ S)$$

and post-erasure

$$elim_{\mathsf{bnat}}\ (\mathsf{bsucc})\ Z\ S \rightarrow_\beta S$$

reduction rules for $elim_{\mathsf{bnat}}$ on bsucc.

In general, any constructor argument may be declared computationally irrelevant by the choice of arrow ($\rightarrow$ or $\Rightarrow$) used in the constructor's type. The type

of the corresponding method argument of the eliminator must then be similarly updated so that it does not depend computationally on either the constructor argument in question or, in the case that this constructor argument is a recursive one, the result of the recursive call of the eliminator on that argument (i.e., the evidence of the induction hypothesis for that argument).

Brady et al. study further omissions that can be made in the representation of a datatype [14]. Erasure annotations allow us to declare some of the representation schemes that they develop. However, their analysis is more general in that it sometimes infers that the *tag* component of a datatype representation is redundant given the other arguments of the eliminator. This sort of redundancy analysis is foreign to our approach.

5.2.3 Eliminator Target Erasure

In Section 5.2.1 we stated that the target argument of an eliminator is always computationally relevant to the computation of that eliminator. In this section, we discuss two classes of inductively defined types in which erasure of the eliminator's target argument at first appears to be warranted, but upon further inspection ends up having undesirable consequences.

Empty Type Target Erasure

An extreme class of inductive types are those having no constructors. Consider the type bottom.

$$\underline{\mathsf{data}}\ \mathsf{bottom} : * \ \underline{\mathsf{where}}$$
$$(* \text{ no constructors } *)$$

In the sequel, we abbreviate bottom as $\bot$. When read logically, the type $\bot$ corresponds to the propositional constant false. Since $\bot$ has no constructors, the eliminator

$$elim_{\bot} : (t : \bot) \to (p : \bot \to *) \Rightarrow p\ t$$

has no method arguments nor any computation rules.

The $\bot$ type is useful when we find ourselves in a contradictory context. Assuming our language is consistent as a logic and we use a weak evaluation strategy at run-time, subterms in contradictory contexts will never be evaluated and are thus dead code. Because we prefer to avoid writing code for dead execution branches, it is useful to have an *undefined* expression that can take on any type in such a situation. The $elim_{\bot}$ operation provides a way out in such situations because we are in a context with contradictory assumptions, we can produce a proof of $\bot$ and then, by $elim_{\bot}$, produce a term of whatever type is required.

Since $elim_{\bot}$ has no computation rules, it does not ever reduce to anything and therefore it does not depend computationally on any of its arguments, even the target $t : \bot$. Therefore, we may prefer to give $elim_{\bot}$ the more precise type

$$elim_{\bot} : (t : \bot) \Rightarrow (p : \bot \rightarrow *) \Rightarrow p\ t.$$

A system in which $elim_{\bot}$ is assigned this type is said to support *empty type target erasure* (ETTE).

ETTE ensures that any application $elim_{\bot}\ @T\ @P$ will simply erase to $elim_{\bot}$, so that the proof T of $\bot$ is erased at run-time. Erasure of T, however, may enable further erasure of λ-binders introducing assumptions upon which T depends computationally. This ripple effect eventually leads to an undesirable consequence.

Theorem 5.2.1 *In a language with empty type target erasure, it is the case that, for any contradictory context* $\Gamma = x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n$*, there is a closed term* M *of type*

$$(x_1 : A_1) \Rightarrow (x_2 : A_2) \Rightarrow \cdots \Rightarrow (x_n : A_n) \Rightarrow B$$

such that the erasure of M *evaluates to some normal form stuck on* $elim_{\bot}$ *rather than a canonical value of type* B*.*

Proof: If Γ is contradictory then, by definition, we can derive $\Gamma \vdash T :^{\mathsf{r}} \bot$ for some

term T. In this case, the term

$$M = \lambda\!\!\lambda x_1{:}A_1.\, \lambda\!\!\lambda x_2{:}A_2. \cdots \lambda\!\!\lambda x_n{:}A_n.\, elim_{\bot}\ @T\ @B$$

has the required type because each assumption $x_i{:}^{\mathsf{c}}A_i$ is reset to $x_i{:}^{\mathsf{r}}A_i$ while typing T. The erasure of M is simply $elim_{\bot}$, a term in normal form. □

If our language is consistent as a logic (i.e., there is no closed proof of $\bot$) and the target language of erasure is evaluated using a weak reduction strategy (as is standard in functional languages), then the situation outlined in the previous theorem seems to be the only one in which the token $elim_{\bot}$ can cause post-erasure evaluation to become stuck.

Theorem 5.2.1 contradicts somewhat the previously given motivation for the type $\bot$: providing an escape hatch in the form of $elim_{\bot}$ for avoiding writing dead code. The theorem says that such branches may not actually be dead code after erasure, because evaluation of the erasure of a program may depend on $elim_{\bot}$.

This mismatch of motivation and outcome is entirely due to empty type target erasure and may be avoided by eliding that feature. For this reason, we feel that ETTE is too permissive and therefore we do not consider it further in the sequel.

The Top Type

Besides the type $\bot$, the simplest possible inductive type is top, which has only a single constructor unit with no arguments.

$$\underline{\mathsf{data}}\ \mathsf{top} : * \ \underline{\mathsf{where}}$$
$$\mathsf{unit} : \mathsf{top}$$

In the sequel, we abbreviate top as $\top$.

The sole inhabitant unit of type $\top$ contains no information. It is merely a token that we may pass around and inspect to determine that which we already know: that it equals unit. This sort of inspection is accomplished by the $\top$ eliminator.

$$elim_{\top} : (t : \top) \to (p : \top \to *) \Rightarrow p\ \mathsf{unit} \to p\ t$$

The eliminator has the following behavior.

$$elim_{\top}\ \mathsf{unit}\ @P\ M \quad \rightarrow_{\beta} \quad M$$

Its type indicates that computation of $elim_{\top}$ depends on its target argument $t : \top$, but is that really true? In general, an eliminator behaves as follows:

1. The topmost data constructor of the target is inspected and the corresponding method argument is picked for execution.

2. The immediate subterms of the target (arguments of the aforementioned topmost constructor) are passed along to the selected method, along with the results of any required recursive calls to the eliminator.

In the case of the type $\top$, neither of these steps imply a computational dependence of $elim_{\top}$ on the target argument $t : \top$.

1. There is only one method, so we always know to pick it without inspecting the top-level constructor of t.

2. As unit has no arguments, there are no components of t that must be passed to the selected method.

For these reasons, we may try to strengthen the type of $elim_{\top}$ to

$$elim_{\top} : (t : \top) \Rightarrow (p : \top \rightarrow *) \Rightarrow p\ \mathsf{unit} \rightarrow p\ t$$

This eliminator has the following behavior.

$$elim_{\top}\ @\mathsf{unit}\ @P\ M \quad \rightarrow_{\beta} \quad M$$

At run-time (i.e., after erasure), this rule becomes

$$elim_{\top}\ M \quad \rightarrow_{\beta} \quad M\ .$$

Equality Types (a.k.a Identity Types)

Recall from Section 5.1.3 the definition of equality as a parameterized indexed inductive type.

$$\begin{aligned}&\underline{\mathsf{data}}\ \mathsf{equal}\ (a : *)\ (x : a) : (y : a) \to * \ \underline{\mathsf{where}}\\ &\qquad \mathsf{refl} : \mathsf{equal}\ a\ x\ x\end{aligned}$$

As we now know, refl can be assigned the type $(a : *) \Rightarrow (x : a) \Rightarrow \mathsf{equal}\ a\ x\ x$. This means that refl has no computationally relevant arguments, so all proofs $\mathsf{refl}\ @a\ @x$ will erase to the same run-time object, namely the token refl. In other words, every equality type $\mathsf{equal}\ a\ x\ y$ will become a unit type at run-time.

The usefulness of the equality type comes from its ability to let us cast from one type to another if we can prove those types are equal. We do this using the eliminator for equal.

$$\begin{aligned}elim_{\mathsf{equal}} : {} & (a : *) \Rightarrow (x, y : a) \Rightarrow (t : \mathsf{equal}\ a\ x\ y) \to\\ & (p : (y : a) \to \mathsf{equal}\ a\ x\ y \to *) \Rightarrow\\ & p\ x\ (\mathsf{refl}\ @a\ @x) \to\\ & p\ y\ t\end{aligned}$$

which has the single run-time reduction rule:

$$elim_{\mathsf{equal}}\ \mathsf{refl}\ M \quad \to_\beta \quad M$$

This eliminator is used to define a *cast* operation between provably equal types.

$$\begin{aligned}cast\ \ : {} & (a : *) \Rightarrow (x, y : a) \Rightarrow \mathsf{equal}\ a\ x\ y \to\\ & (p : a \to *) \Rightarrow p\ x \to p\ y\\ cast = {} & \lambda\!\!\lambda a.\ \lambda\!\!\lambda x.\ \lambda\!\!\lambda y.\ \lambda t.\ \lambda\!\!\lambda p.\ \lambda m.\\ & elim_{\mathsf{equal}}\ @a\ @x\ @y\ t\ @(\lambda y.\ \lambda\_.\ p\ y)\ m\end{aligned}$$

If x equals y then $p\,x$ and $p\,y$ are equal as types. Therefore we may safely cast from one type to the other. Given its definition in terms of $elim_{\mathsf{equal}}$, the *cast* operation

reduces as follows at run-time:

$$\mathit{cast}\ \mathsf{refl}\ M \rightarrow_\beta^* M$$

Token Type Target Erasure

The types equal and $\top$ share the following property: at run-time they are both represented by a single token that carries no sub-component information. We argued in the case of $\top$ that the eliminator $\mathit{elim}_\top$ does not depend computationally on its target. Since the single token refl is the run-time representation of all values of type $\mathsf{equal}\ a\ x\ y$, the same argument applies to $\mathit{elim}_{\mathsf{equal}}$.

This means we can change the type of $\mathit{elim}_{\mathsf{equal}}$ to

$$\begin{aligned}\mathit{elim}_{\mathsf{equal}} : (a : *) \Rightarrow (x, y : a) \Rightarrow (t : \mathsf{equal}\ a\ x\ y) \Rightarrow \\ (p : (y : a) \rightarrow \mathsf{equal}\ a\ x\ y \rightarrow *) \Rightarrow \\ p\ x\ (\mathsf{refl}\ @a\ @x) \rightarrow \\ p\ y\ t\end{aligned}$$

(note the computational irrelevance of the target argument $t : \mathsf{equal}\ a\ x\ y$). The behavior of $\mathit{elim}_{\mathsf{equal}}$ then becomes

$$\mathit{elim}_{\mathsf{equal}}\ @A\ @X\ @X\ @(\mathsf{refl}\ @A\ @X)\ @P\ M \quad \rightarrow_\beta \quad M\ .$$

At run-time (after erasure) the behavior of $\mathit{elim}_{\mathsf{equal}}$ is

$$\mathit{elim}_{\mathsf{equal}}\ M \quad \rightarrow_\beta \quad M\ .$$

The optimization of token type eliminators just described destroys any hope of preserving an analog of Theorem 3.3.9 that says any post-erasure reduction of a type- and phase-correct term corresponds to one or more pre-erasure reductions. Theorem 3.3.9 was the basis for Theorem 3.3.10, which also fails to have an analogous version in the language with both erasure annotations and inductive types. The following example proves these claims.

Example: Consider the following variation on the token type equal that allows us to state the equivalence of two types:

$$\underline{\mathsf{data}}\ \mathsf{tyeq}\ (a : *) : * \to * \ \underline{\mathsf{where}}$$
$$\mathsf{tyrefl} : \mathsf{tyeq}\ a\ a$$

By the principle of token type target erasure, it has the following eliminator:

$$\begin{aligned} elim_{\mathsf{tyeq}} \quad : \quad & (a, b : *) \Rightarrow (t : \mathsf{tyeq}\ a\ b) \Rightarrow \\ & (p : (b : *) \to \mathsf{tyeq}\ a\ b \to *) \Rightarrow \\ & p\ a\ (\mathsf{tyrefl}\ @a) \to \\ & p\ b\ t \end{aligned}$$

This eliminator has the following behavior at compile-time:

$$elim_{\mathsf{tyeq}}\ @A\ @B\ @(\mathsf{tyrefl}\ @A)\ @P\ M \quad \to_\beta \quad M$$

and the following behavior at run-time (after erasure):

$$elim_{\mathsf{tyeq}}\ M \quad \to_\beta \quad M$$

Figure 5.1 defines a term *loopy* in terms of tyeq that, after erasure, reduces to $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, the canonical divergent λ-calculus term. The trick used to define *loopy* is to make a patently false assumption, namely $p : \mathsf{tyeq}\ a\ (a \to b)$. Given this assumption, we can apply something of type a (or $a \to b$) to itself after some suitable coercions. Token type target erasure allows us to write the example in such a way that the binder for p, and all case analysis on p (calls to $elim_{\mathsf{tyeq}}$) are erased, leaving only a diverging term.

This example shows that token type target erasure ruins any hope for results analogous to the following properties of erasure in EPTS:

1. Erasure Reflects Reductions (Theorem 3.3.9), and

2. Erasure Preserves Strong Normalization (Theorem 3.3.10)

$$\underline{\mathsf{data}}\ \mathsf{tyeq}\ (a : *) : * \to * \ \underline{\mathsf{where}}$$

$$\mathsf{tyrefl} : \mathsf{tyeq}\ a\ a$$

$$\begin{array}{rcl}
\mathit{coerce} & : & (a, b : *) \Rightarrow \mathsf{tyeq}\ a\ b \Rightarrow a \to b \\
\mathit{coerce} & = & \Lambda\!\!\Lambda a, b, t.\ \lambda x{:}a.\ \mathit{elim}_{\mathsf{tyeq}}\ @a\ @b\ @t\ @(\lambda b.\ \lambda_.\ b)\ x
\end{array}$$

$$\begin{array}{rcl}
\mathit{symm} & : & (a, b : *) \Rightarrow \mathsf{tyeq}\ a\ b \Rightarrow \mathsf{tyeq}\ b\ a \\
\mathit{symm} & = & \Lambda\!\!\Lambda a, b, t.\ \mathit{elim}_{\mathsf{tyeq}}\ @a\ @b\ @t\ @(\lambda c.\ \lambda_.\ \mathsf{tyeq}\ c\ a)\ (\mathsf{tyrefl}\ @a)
\end{array}$$

$$\begin{array}{rcl}
\mathit{loopy} & : & (a, b : *) \Rightarrow \mathsf{tyeq}\ a\ (a \to b) \Rightarrow b \\
\mathit{loopy} & = & \Lambda\!\!\Lambda a, b.\ \Lambda\!\!\Lambda p. \\
 & & \quad \underline{\mathsf{let}}\ w : a \to b \\
 & & \qquad w = \lambda x{:}a.\ \mathit{coerce}\ @a\ @(a \to b)\ @p\ x\ x \\
 & & \quad \underline{\mathsf{in}}\ w\ (\mathit{coerce}\ @(a \to b)\ @a\ @(\mathit{symm}\ @a\ @(a \to b)\ @p)\ w)
\end{array}$$

$$\begin{array}{rcll}
\mathit{coerce}^{\bullet} & = & \lambda x.\ \mathit{elim}_{\mathsf{tyeq}}\ x & \to_{\beta}^{*}\ \lambda x.\ x \\
\mathit{loopy}^{\bullet} & = & \underline{\mathsf{let}}\ w = \lambda x.\ \mathit{coerce}^{\bullet}\ x\ x\ \underline{\mathsf{in}}\ w\ (\mathit{coerce}^{\bullet}\ w) & \to_{\beta}^{*}\ (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)
\end{array}$$

Figure 5.1: Example showing that token type target erasure prevents one from extending theorems that erasure reflects reductions (Theorem 3.3.9) and preserves strong normalization (Theorem 3.3.10)

If we do not allow token type target erasure, the terms *coerce*, *symm*, and *loopy* must be re-annotated with fewer opportunities for erasure. The resulting erasure of *loopy* becomes

$$\begin{aligned} loopy^{\bullet} \;=\; & \lambda p.\, elim_{\mathsf{tyeq}}\ p \\ & (elim_{\mathsf{tyeq}}\ (elim_{\mathsf{tyeq}}\ p\ \mathsf{tyrefl})\ (\lambda x.\, elim_{\mathsf{tyeq}}\ p\ x\ x)) \\ & (elim_{\mathsf{tyeq}}\ (elim_{\mathsf{tyeq}}\ p\ \mathsf{tyrefl})\ (\lambda x.\, elim_{\mathsf{tyeq}}\ p\ x\ x)). \end{aligned}$$

This term is in normal form, because every call to $elim_{\mathsf{tyeq}}$ is blocked from executing due to the variable p.

This is a serious problem. Termination is an important program property that should be preserved by an erasure semantics. For this reason, we do not allow token type target erasure.

5.3 A PARADIGMATIC EXAMPLE: VARIOUS SUM TYPES

Chapter 2 discusses how certain typed λ-calculi may be viewed as both formal logics and primitive programming languages. In this discussion, we saw that universal quantifiers from logic correspond to dependent product types. This suggests the question: what types, if any, correspond to existential quantifiers?

The existentially quantified formula $\exists x{:}A.\, B$, says that there exists an x of type A such that B holds of x (in this formula the scope of x is B, so B may certainly mention x). A proof of this formula consists of

1. a *witness* M of type A, and

2. a proof N that B holds of M (i.e., N proves $B[M/x]$)

In type theory, therefore, $\exists x{:}A.\, B$ is interpreted as a type of pairs $\langle M, N\rangle$ in which M has type A and N has type $B[M/x]$. In this way, the type of the second component of the pair is allowed to depend on the value of the first component. Type theoretically, this is a sum type and it is usually written as $\Sigma x{:}A.\, B$.

5.3.1 Strong Sums

Some type theories include sum types as a built-in type former. However, in a language supporting inductively defined types, we may define them on our own as the following family of inductive types:

$$\begin{array}{c} \underline{\mathsf{data}}\ \mathsf{sum}\ (a : *)\ (b : a \to *) : * \ \underline{\mathsf{where}} \\ \mathsf{pair} : (x : a) \to b\ x \to \mathsf{sum}\ a\ b \end{array}$$

This declaration introduces the following type constructor and data constructor

$$\begin{array}{rcl} \mathsf{sum} & : & (a : *) \to (b : a \to *) \to * \\ \mathsf{pair} & : & (a : *) \Rightarrow (b : a \to *) \Rightarrow (x : a) \to b\ x \to \mathsf{sum}\ a\ b \end{array}$$

as well as the following eliminator

$$\begin{array}{l} \mathit{elim}_{\mathsf{sum}} : (a : *) \Rightarrow (b : a \to *) \Rightarrow (t : \mathsf{sum}\ a\ b) \to \\ \qquad (p : \mathsf{sum}\ a\ b \to *) \Rightarrow \\ \qquad ((x : a) \to (y : b\ x) \to p\ (\mathsf{pair}\ @a\ @b\ x\ y)) \to \\ \qquad p\ t \end{array}$$

which behaves as follows at run-time:

$$\mathit{elim}_{\mathsf{sum}}\ (\mathsf{pair}\ X\ Y)\ M \quad \to_\beta \quad M\ X\ Y$$

This sum type is strong in the sense that both components of such pairs, including the witness, may be projected out. These projections are provided by the functions

$$\begin{array}{rcl} \mathit{fst} & : & (a : *) \Rightarrow (b : a \to *) \Rightarrow \mathsf{sum}\ a\ b \to a \\ \mathit{snd} & : & (a : *) \Rightarrow (b : a \to *) \Rightarrow (t : \mathsf{sum}\ a\ b) \to b\ (\mathit{fst}\ @a\ @b\ t) \end{array}$$

which are defined as follows:

$$\begin{array}{rcll} \mathit{fst} & = & \Lambda a.\, \Lambda b.\, \lambda t.\, \mathit{elim}_{\mathsf{sum}}\ @a\ @b\ t\ @(\lambda \_.\, a) & (\lambda x.\, \lambda y.\, x) \\ \mathit{snd} & = & \Lambda a.\, \Lambda b.\, \lambda t.\, \mathit{elim}_{\mathsf{sum}}\ @a\ @b\ t\ @(\lambda \_.\, b\ (\mathit{fst}\ @a\ @b\ t))\ (\lambda x.\, \lambda y.\, y) \end{array}$$

An example of a strong sum is $\mathsf{sum\ nat}\ (\lambda n.\, \mathsf{list}\ A\ n)$, the type of length indexed lists paired with their length. In the usual notation, we would write this type as $\Sigma n : \mathsf{nat}.\, \mathsf{list}\ A\ n$. It is easy to extract both the length of a list stored in this way as well as the underlying length-indexed list using *fst* and *snd*.

5.3.2 Weak Sums

There is also a weaker form of sum type in which the witness may not be directly observed through projection. We may define this type as follows.

$$\begin{array}{l}\underline{\mathsf{data}}\ \mathsf{exists}\ (a : *)\ (b : a \to *) : * \ \underline{\mathsf{where}} \\ \qquad \mathsf{pack} : (x : a) \Rightarrow b\ x \to \mathsf{exists}\ a\ b\end{array}$$

This type has the eliminator

$$\begin{array}{l}\mathit{elim}_{\mathsf{exists}} : (a : *) \Rightarrow (b : a \to *) \Rightarrow (t : \mathsf{exists}\ a\ b) \to \\ \qquad (p : \mathsf{exists}\ a\ b \to *) \Rightarrow \\ \qquad ((x : a) \Rightarrow (y : b\ x) \to p\ (\mathsf{pack}\ @a\ @b\ @x\ y)) \to \\ \qquad p\ t\end{array}$$

with the following lone run-time reduction rule:

$$\mathit{elim}_{\mathsf{exists}}\ (\mathsf{pack}\ Y)\ M \quad \to_\beta \quad M\ Y$$

The only difference between sum and exists is that the first component of pairs in the latter type are marked for erasure. The type of the eliminator reflects this change, as outlined in the Section 5.2.2. Due to this change, $\mathit{elim}_{\mathsf{exists}}$ has a weaker type than $\mathit{elim}_{\mathsf{sum}}$, because it expects a stronger requirement on the method argument, namely that it not depend computationally on the first component of the pair. For this reason, the definition of *fst* given in Section 5.3.1 may not be adapted to the exists type (and renamed *witness*), as it would be phase-incorrect.

$$\begin{array}{lcl}\mathit{witness} & : & (a : *) \Rightarrow (b : a \to *) \Rightarrow \mathsf{exists}\ a\ b \to a \\ \mathit{witness} & = & \lambda\!\!\lambda a.\, \lambda\!\!\lambda b.\, \lambda t.\, \mathit{elim}_{\mathsf{exists}}\ @a\ @b\ t\ @(\lambda\_.\, a)\ (\lambda\!\!\lambda x.\, \lambda y.\, x)\end{array}$$

We may see the phase error clearly by inspecting the erasure of the definiens

$$\lambda t.\ elim_{\mathsf{exists}}\ (\lambda y.\ x)\ t$$

and noting the exposed variable x.

What can be done then with the first component of the pair? The eliminator for exists can be used to program the following unpacking operator:

$$\begin{aligned} unpack\ &:\ (a : *) \Rightarrow (b : a \to *) \Rightarrow \mathsf{exists}\ a\ b \to \\ &\qquad (c : *) \Rightarrow ((x : a) \Rightarrow b\ x \to c) \to c \\ unpack\ &= \lambda\!\!\lambda a.\ \lambda\!\!\lambda b.\ \lambda t.\ \lambda\!\!\lambda c.\ \lambda m.\ elim_{\mathsf{exists}}\ @a\ @b\ t\ @(\lambda\_.\ c)\ m \end{aligned}$$

Notice the restriction that clients m of *unpack* may not depend computationally on the existential witness since they have the type $(x : a) \Rightarrow b\ x \to c$ which is polymorphic in $x : a$. This feature of exists is what earns it the name "weak" sum: no program may depending computationally on the witness component of an existential.

An example making use of the weak sum is the following function that injects boolean lists into length-indexed boolean lists of some particular length n.

$$\begin{aligned} embed\ &:\ \mathsf{blist} \to \mathsf{exists\ nat}\ (\lambda n.\ \mathsf{ilist}\ n) \\ embed\ &=\ \lambda t.\ elim_{\mathsf{blist}}\ t\ @(\lambda\_.\ \mathsf{exists\ nat}\ (\lambda n.\ \mathsf{ilist}\ n)) \\ &\qquad (\mathsf{pack}\ @\mathsf{nat}\ @(\lambda n.\ \mathsf{ilist}\ n)\ @\mathsf{zero}\ \mathsf{inil}) \\ &\qquad (\lambda x{:}\mathsf{bool}.\ \lambda xs{:}\mathsf{blist}.\ \lambda fxs{:}\mathsf{exists\ nat}\ (\lambda n.\ \mathsf{ilist}\ n). \\ &\qquad\quad unpack\ @\mathsf{nat}\ @(\lambda n.\ \mathsf{ilist}\ n)\ fxs\ @(\mathsf{exists\ nat}\ (\lambda n.\ \mathsf{ilist}\ n)) \\ &\qquad\qquad (\lambda\!\!\lambda m{:}\mathsf{nat}.\ \lambda ys{:}\mathsf{ilist}\ m. \\ &\qquad\qquad\quad \mathsf{pack}\ @\mathsf{nat}\ @(\lambda n.\ \mathsf{ilist}\ n)\ @(\mathsf{succ}\ m)\ (\mathsf{icons}\ @m\ x\ ys))) \end{aligned}$$

Since definitions in terms of eliminators can be difficult to read, we offer the fol-

lowing pseudocode for *embed* as a convenience to the reader.

$$\begin{aligned} \mathit{embed}\ (\mathsf{bnil}) &= \mathsf{pack}\ @\mathsf{nat}\ @(\lambda n.\, \mathsf{ilist}\ n)\ @\mathsf{zero}\ \mathsf{inil} \\ \mathit{embed}\ (\mathsf{bcons}\ x\ xs) &= \underline{\mathsf{let}}\ \mathsf{pack}\ @\_\ @\_\ @m\ ys = \mathit{unpack}\ (\mathit{embed}\ xs)\ \underline{\mathsf{in}} \\ &\quad\ \mathsf{pack}\ @\mathsf{nat}\ @(\lambda n.\, \mathsf{ilist}\ n)\ @(\mathsf{succ}\ m)\ (\mathsf{icons}\ @m\ x\ ys) \end{aligned}$$

The *embed* function may be useful if we have a blist and we want to perform some ilist operations on it. Note how the recursive case of this function uses *unpack* to temporarily handle in a non-computational way the length m of ys, the list resulting from the recursive call. This length is passed along to two compile-time contexts, namely the new length of the return value, and the tail length argument of icons.

Weak sums are useful for data abstraction [54]. When x is a type[1], then the existential construction hides the representation of that type from clients of data packaged in this way and therefore protects them from any future changes made to that representation that do not affect the interface b.

5.3.3 Subset Types

Another variation on dependent sum types are the so-called *subset types*. The idea here is to have a type former analogous to set comprehensions in set theory. For example, the set $\{n \in \mathsf{nat} \mid \mathit{even}\ n\}$ is the set of all the even natural numbers. More generally, the set $\{x \in A \mid B\}$ is the subset of A consisting of elements x for which B holds (in general, B mentions x). We prefer the name "type comprehensions" for this construction, but "subset types" is already an established term.

At first glance, dependent sums seem like a good implementation for subset types. A dependent pair contains both the element $x : A$ as well as the evidence

[1]This is only possible given the definition of exists if there is some sort $\triangle : *$, so that we may have $a = \triangle$ in $x : a : *$. Otherwise, a variant of exists must be used in which a has a sort higher than $*$.

that it satisfies the property B. However one does not usually consider evidence for B to be a component of elements of $\{x \in A \mid B\}$. This is where erasure comes into play.

We can define subset types as dependent sums with the evidence component of the pair marked for erasure.

$$\begin{array}{l}\underline{\mathsf{data}}\ \mathsf{subset}\ (a : *)\ (b : a \to *) : * \ \underline{\mathsf{where}} \\ \qquad \mathsf{member} : (x : a) \to b\ x \Rightarrow \mathsf{subset}\ a\ b\end{array}$$

The subset type has the following eliminator:

$$\begin{array}{l}\mathit{elim}_{\mathsf{subset}} : (a : *) \Rightarrow (b : a \to *) \Rightarrow (t : \mathsf{subset}\ a\ b) \to \\ \qquad (p : \mathsf{subset}\ a\ b \to *) \Rightarrow \\ \qquad ((x : a) \to (y : b\ x) \Rightarrow p\ (\mathsf{member}\ @a\ @b\ x\ @y)) \to \\ \qquad p\ t\end{array}$$

with the following lone run-time reduction rule:

$$\mathit{elim}_{\mathsf{subset}}\ (\mathsf{member}\ X)\ M \quad \to_\beta \quad M\ X$$

Since the inhabitant of A is computationally relevant to member, we may define *inject*, the analog of *fst* from Section 5.3.1,

$$\begin{array}{lcl}\mathit{inject} & : & (a : *) \Rightarrow (b : a \to *) \Rightarrow \mathsf{subset}\ a\ b \to a \\ \mathit{inject} & = & \lambda\!\!\lambda a.\, \lambda\!\!\lambda b.\, \lambda t.\, \mathit{elim}_{\mathsf{subset}}\ @a\ @b\ t\ @(\lambda\_.\, a)\ (\lambda x.\, \lambda\!\!\lambda y.\, x)\end{array}$$

but the analog of *snd* is not well-formed

$$\begin{array}{lcl}\mathit{evidence} & : & (a : *) \Rightarrow (b : a \to *) \Rightarrow (t : \mathsf{subset}\ a\ b) \to b\ (\mathit{inject}\ @a\ @b\ t) \\ \mathit{evidence} & = & \lambda\!\!\lambda a.\, \lambda\!\!\lambda b.\, \lambda t.\, \mathit{elim}_{\mathsf{subset}}\ @a\ @b\ t\ @(\lambda\_.\, b\ (\mathit{inject}\ @a\ @b\ t))\ (\lambda x.\, \lambda\!\!\lambda y.\, y)\end{array}$$

due to the phase-incorrect occurrence of y.

Salvesen and Smith [79], working in the context of Martin-Löf's intensional type theory extended with subset types, prove that one cannot project out the

evidence $B(x)$ from a subset element $x \in \{z \in A \mid B(z)\}$ unless one can do so for the entire type, that is, unless $\forall x : A.\, B(x)$ is provable. Their result is consistent with our observations here.

5.4 SQUASH TYPES

Just as we divided function types into computational and non-computational varieties, we now define type constructor that serves to divide the world of types into computational and non-computational types. This type is reminiscent of the squash type of *Nuprl* [22, Section 10.3], so we name it squash:

$$\underline{\mathsf{data}}\ \mathsf{squash}\ (a : *) : * \ \underline{\mathsf{where}}$$
$$\mathsf{poof} : a \Rightarrow \mathsf{squash}\ a$$

The data constructor poof is so named because all its arguments disappear (are erased) before run-time. In fact, squash a is a token type and therefore all its inhabitants share a common run-time representation: the token poof. The name "squash" was chosen (in Nuprl), precisely because all elements of a squash type are identified — all informational content has been squashed out of its inhabitants. As we will see in the Section 5.4.4, squash is a logical modality. We therefore abbreviate squash A as $\bigcirc A$ (and squash as $\bigcirc$).

The eliminator for squash is

$$\begin{aligned} elim_{\bigcirc} : {} & (a : *) \Rightarrow (t : \bigcirc a) \rightarrow \\ & (p : \bigcirc a \rightarrow *) \Rightarrow \\ & ((x : a) \Rightarrow p\ (\mathsf{poof}\ @a\ @x)) \rightarrow \\ & p\ t. \end{aligned}$$

A special case of this eliminator is

$$\begin{aligned} case_{\bigcirc} &: (a, c : *) \Rightarrow \bigcirc a \rightarrow (a \Rightarrow c) \rightarrow c \\ case_{\bigcirc} &= \lambda\!\!\lambda a.\, \lambda\!\!\lambda c.\, \lambda t.\, \lambda m.\, elim_{\bigcirc}\ @a\ t\ @(\lambda\_.\, c)\ m \end{aligned}$$

The behavior of this operator is as follows:

$$\begin{array}{lll} \text{(at compile-time)} & \mathit{case}_{\bigcirc}\ @A\ @C\ (\mathsf{poof}\ @A\ @X)\ M & \rightarrow^*_\beta\ \ M\ @X \\ \text{(at run-time)} & \mathit{case}_{\bigcirc}\ \mathsf{poof}\ M & \rightarrow^*_\beta\ \ M \end{array}$$

5.4.1 Relating $\bigcirc$ and $\Rightarrow$

The modality $\bigcirc$ is essentially the type constructor form of the non-dependent non-computational function arrow $\Rightarrow$. In fact, we have the following isomorphism:

$$A \Rightarrow B \quad \cong \quad \bigcirc A \rightarrow B$$

Data witnessing the isomorphism between these two types appears in Figure 5.2.

- Functions *in* and *out* between these two types, and
- Terms *out_in* and *in_out* proving that *in* and *out* are inverses (each one cancels out the other)

The first proof (*out_in*) follows immediately by reflexivity, since the two sides of the equality are definitionally equal.

$$\begin{array}{lll} \mathit{out}\ (\mathit{in}\ m)\ @x & \rightarrow^*_\beta\ \ \mathit{in}\ m\ (\mathsf{poof}\ @A\ @x) & \\ \qquad = & \mathit{case}_{\bigcirc}\ @A\ @C\ (\mathsf{poof}\ @A\ @x)\ m & \rightarrow^*_\beta\ \ m\ @x \end{array}$$

The second proof (*in_out*), makes use of $\mathit{elim}_{\bigcirc}$ to reduce the problem of proving

$$(y : \bigcirc A) \rightarrow \mathsf{equal}\ C\ (\mathit{in}\ (\mathit{out}\ f)\ y)\ (f\ y)$$

to the simpler problem of proving

$$(x : A) \Rightarrow \mathsf{equal}\ C\ (\mathit{in}\ (\mathit{out}\ f)\ (\mathsf{poof}\ @A\ @x))\ (f\ (\mathsf{poof}\ @A\ @x)).$$

This latter problem is solved quite easily since the two sides of the equality are definitionally equal.

$$\begin{array}{lll} \mathit{in}\ (\mathit{out}\ f)\ (\mathsf{poof}\ @A\ @x) & \rightarrow^*_\beta\ \ \mathit{case}_{\bigcirc}\ @A\ @C\ (\mathsf{poof}\ @A\ @x)\ (\mathit{out}\ f) & \\ & \rightarrow^*_\beta\ \ \mathit{out}\ f\ @x & \rightarrow^*_\beta\ \ f\ (\mathsf{poof}\ @A\ @x) \end{array}$$

$$
\begin{array}{rcl}
in & = & \lambda f.\, \lambda t.\, case_{\bigcirc}\ @A\ @C\ t\ f \quad : \quad (A \Rightarrow C) \to (\bigcirc A \to C) \\
out & = & \lambda f.\, \lambda\!\!\lambda x.\, f\ (\mathsf{poof}\ @A\ @x) \quad : \quad (\bigcirc A \to C) \to (A \Rightarrow C)
\end{array}
$$

$$
\begin{array}{rcl}
out\_in & : & (m : A \Rightarrow C) \to (x : A) \Rightarrow \mathsf{equal}\ C\ (out\ (in\ m)\ @x)\ (m\ @x) \\
in\_out & : & (f : \bigcirc A \to C) \to (y : \bigcirc A) \to \mathsf{equal}\ C\ (in\ (out\ f)\ y)\quad (f\ y) \\
out\_in & = & \lambda m.\, \lambda\!\!\lambda x.\, \mathsf{refl}\ @A\ @(m\ @x) \\
in\_out & = & \lambda f.\, \lambda t.\, elim_{\bigcirc}\ @A\ t \\
 & & \quad @(\lambda y{:}\bigcirc A.\, \mathsf{equal}\ C\ (in\ (out\ f)\ y)\ (f\ y)) \\
 & & \quad (\lambda\!\!\lambda x{:}A.\, \mathsf{refl}\ @C\ @(f\ (\mathsf{poof}\ @A\ @x)))
\end{array}
$$

Figure 5.2: An isomorphism between $A \Rightarrow B$ and $\bigcirc A \to B$

Although the non-dependent function space $A \Rightarrow B$ is equivalent to $\bigcirc A \to B$, the *dependent* function space $(x : A) \Rightarrow B$ is *not* equivalent to $(x : \bigcirc A) \to B$. To see why, recall the typing rule for $(x : A) \Rightarrow B$ from Chapter 3.

$$
\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\mathsf{c}} x{:}A.\, B :^{\mathsf{r}} s_3}\ \Pi^{\mathrm{c}}\text{-Form}
$$

(Remember that $(x : A) \Rightarrow B$ is syntactic sugar for $\Pi^{\mathsf{c}} x{:}A.\, B$.) This rule says that the B in $(x : A) \Rightarrow B$ may depend computationally on $x : A$, but in the type $(x : \bigcirc A) \to B$, the B may depend computationally only on $x : \bigcirc A$, a far weaker premise.

The following example makes essential use of the extra expressiveness afforded by our rule for typing $(x : A) \to B$. Consider the following definition of a boolean list type of a particular length.

$$
\begin{array}{rcl}
blist & : & \mathsf{nat} \to * \\
blist\ \mathsf{zero} & = & \top \\
blist\ (\mathsf{succ}\ n) & = & \mathsf{bool} \times blist\ n
\end{array}
$$

In contrast to the inductively defined type blist, this type is defined by recursion over a natural number. An equivalent definition can be encoded using an eliminator for natural numbers allowing one to eliminate into a higher sort than $*$. Clearly *blist* depends computationally on its natural number argument. Now consider the following specialized identity function for *blist*s.

$$\begin{array}{lcl} \mathit{blist\_identity} & : & (n : \mathsf{nat}) \Rightarrow \mathit{blist}\ n \rightarrow \mathit{blist}\ n \\ \mathit{blist\_identity} & = & \lambda\!\!\lambda n{:}\mathsf{nat}.\, \lambda xs{:}\mathit{blist}\ n.\, xs \end{array}$$

This is a perfectly valid definition in which the well-formedness of the type of *blist_identity* relies essentially on the fact that $\mathit{blist}\ n \rightarrow \mathit{blist}\ n$ can depend computationally on $n : \mathsf{nat}$.

We conclude that $\Rightarrow$ is more expressive than $\bigcirc$ in a language with a facility for defining inductive types, since we can express $\bigcirc$ in terms of $\Rightarrow$, but we cannot express non-computational dependent function space $(x : A) \Rightarrow B$ in terms of $\bigcirc$. We view this fact as a principal advantage of our approach over squash types.

5.4.2 Correspondence with Nuprl's Squash Type

In Nuprl, the squash type is written as $\downarrow A$ and is defined as

$$\downarrow A = \{x : \top \mid A\}.$$

One justification for calling $\bigcirc A$ a squash type is that $\bigcirc A$ is isomorphic to $\downarrow A$ given the corresponding definition of $\downarrow A$ in terms of the subset type defined in Section 5.3.3.

$$\downarrow A \quad = \quad \mathsf{subset}\ \top\ (\lambda\_.\, A)$$

$$
\begin{array}{rcl}
to & : & \bigcirc A \to \downarrow A \\
from & : & \downarrow A \to \bigcirc A \\
to & = & \lambda t.\, elim_{\bigcirc}\ @A\ t\ @(\lambda\_.\downarrow A)\ (\lambda\!\!\lambda y.\, poof\ @A\ @y) \\
from & = & \lambda t.\, elim_{\downarrow}\ @A\ t\ @(\lambda\_.\bigcirc A)\ (\lambda\!\!\lambda y.\, \mathsf{poof}\ @A\ @y)
\end{array}
$$

$$
\begin{array}{rcl}
to\_from & : & (t : \downarrow A) \to \mathsf{equal}\ (\downarrow A)\ (to\ (from\ t))\ t \\
from\_to & : & (t : \bigcirc A) \to \mathsf{equal}\ (\bigcirc A)\ (from\ (to\ t))\ t \\
to\_from & = & \lambda t.\, elim_{\downarrow}\ @A\ t \\
 & & \quad @(\lambda t.\, \mathsf{equal}\ (\downarrow A)\ (to\ (from\ t))\ t) \\
 & & \quad (\lambda\!\!\lambda x.\, \mathsf{refl}\ @(\downarrow A)\ @(poof\ @A\ @x)) \\
from\_to & = & \lambda t.\, elim_{\bigcirc}\ @A\ t \\
 & & \quad @(\lambda t.\, \mathsf{equal}\ (\bigcirc A)\ (from\ (to\ t))\ t) \\
 & & \quad (\lambda\!\!\lambda x.\, \mathsf{refl}\ @(\bigcirc A)\ @(\mathsf{poof}\ @A\ @x))
\end{array}
$$

Figure 5.3: An isomorphism between two squash type representations

It is easier to work with this type if we define our own custom introduction and elimination functions:

$$
\begin{array}{lcl}
\mathit{poof} & : & (a : *) \Rightarrow a \Rightarrow \downarrow a \\
\mathit{poof} & = & \lambda\!\!\lambda a, x.\, \mathsf{member}\ @\top\ @(\lambda\_.\, a)\ \mathsf{unit}\ @x \\
\mathit{elim}_{\downarrow} & : & (a : *) \Rightarrow (t : \downarrow a) \rightarrow (p : \downarrow a \rightarrow *) \Rightarrow ((x : a) \Rightarrow p\ (\mathit{poof}\ @a\ @x)) \rightarrow p\ t \\
\mathit{elim}_{\downarrow} & = & \lambda\!\!\lambda a.\, \lambda t.\, \lambda\!\!\lambda p.\, \lambda f. \\
 & & \quad \mathit{elim}_{\mathsf{subset}}\ @\top\ @(\lambda\_.\, a)\ t\ @p \\
 & & \qquad (\lambda x.\, \mathit{elim}_{\top}\ x\ @(\lambda x.\, (y : a) \Rightarrow p\ (\mathsf{member}\ @\top\ @(\lambda\_.\, a)\ x\ @y))\ f)
\end{array}
$$

Note that $\mathit{elim}_{\downarrow}$ also simulates $\mathit{elim}_{\bigcirc}$

$$\mathit{elim}_{\downarrow}\ @A\ (\mathit{poof}\ @A\ @X)\ @P\ F \quad \rightarrow^*_{\beta} \quad F@X$$

Given this interface for $\downarrow A$, it is almost trivial to construct an isomorphism between $\downarrow A$ and $\bigcirc A$, as is shown in Figure 5.3.

5.4.3 Usage of the Squash Type

In Nuprl squash types are defined in terms of subset types (see Section 5.3.3) and these two mechanisms are put to use to demarcate the non-computational portions of a program development which mostly involve proofs of properties of the rest of the program. Squash types have been put to good use in the Nuprl system for many years.

In Section 5.3.3, we discussed the impossibility of projecting out the evidence component of a subset type $\{x \in A \mid B\}$. We can now repair that example using the squash type.

$$
\begin{array}{lcl}
\mathit{evidence} & : & (a : *) \Rightarrow (b : a \rightarrow *) \Rightarrow (t : \mathsf{subset}\ a\ b) \rightarrow \bigcirc\,(b\ (\mathit{inject}\ @a\ @b\ t)) \\
\mathit{evidence} & = & \lambda\!\!\lambda a.\, \lambda\!\!\lambda b.\, \lambda t.\, \mathit{elim}_{\mathsf{subset}}\ @a\ @b\ t \\
 & & \qquad @(\lambda t.\, \bigcirc(b\ (\mathit{inject}\ @a\ @b\ t))) \\
 & & \qquad (\lambda x.\, \lambda\!\!\lambda y.\, \mathsf{poof}\ @(b\ x)\ @y)
\end{array}
$$

As noted in Section 5.3.3, the subset construction in Martin-Löf type theory is somewhat weak, since one cannot prove the seemingly trivial statement that $x \in \{z \in A \mid B(z)\}$ implies $B(x)$. To remedy this deficiency, Nordström, Petersson, and Smith (NPS) developed the so-called *subset theory*, a type theory with the subset type that can be interpreted in basic type theory without subsets [70, Part II]. In the subset theory, the notions of type and proposition are distinct, and one has the judgment form A is true in addition to the usual judgment form $a \in A$. Finally, the details of the interpretation guarantee that one can prove $B(a)$ is true given $a \in \{x \in A \mid B(x)\}$.

In light of how the type $\bigcirc$ similarly revives the effort to extract evidence of the subset property (see the type of *evidence* above), it may be fruitful to think of $\bigcirc A$ as a type internalizing the judgment A is true from the subset theory of NPS. In his doctoral thesis [15, Section 3.3.1], Caldwell cites Salvesen and Smith's negative result [79] concerning the weakness of the subset type in intensional type theory. Caldwell concludes that

> [Salvesen and Smith's result] does seem to indicate an essential weakness in the intensional theory since they show unequivocally that it cannot be extended to reasonably accommodate a subset type.

To the contrary, the type subset appears to be a reasonable implementation of subset types in an intensional type theory. The extension of type theory to handle erasure annotations seems to us less cumbersome than the additional layer of interpretation found in the NPS approach, especially since erasure annotations have the additional benefit of expressing parametric polymorphism in the source language.

5.4.4 Laws Concerning $\bigcirc$

We informally interpret the $\bigcirc$ modality as "for some unknown reason". This modality satisfies the following basic laws:

$$\begin{array}{ll} \text{(Forgetfulness)} & A \to \bigcirc A \\ \text{(Blind Reasoning)} & \bigcirc(A \to B) \to \bigcirc A \to \bigcirc B \end{array}$$

Forgetfulness says that we may intentionally forget the reason for the truth of a proposition A at any time by moving from A to $\bigcirc A$. Blind Reasoning says that modus ponens applies even when we do not know the original reasons for $A \to B$ and A, although we must remain ignorant about the ultimate reason for B as it is composed from reasons about which we know nothing.

Forgetfulness and Blind Reasoning may be proved by applying the isomorphism $A \Rightarrow B \cong \bigcirc A \to B$ to the following terms:

$$\begin{array}{rcl} \lambda\!\!\lambda x.\, \mathsf{poof}\ @A\ @x & : & A \Rightarrow \bigcirc A \\ \lambda\!\!\lambda f, x.\, \mathsf{poof}\ @A\ @(f\ x) & : & (A \to B) \Rightarrow A \Rightarrow \bigcirc B \end{array}$$

The laws of Forgetfulness and Blind Reasoning ensure that the modality $\bigcirc$ is a *normal modal logic*.

McBride and Paterson recently introduced *applicative functors* as a means of structuring effectful programs in a purely functional language [61]. It is straightforward to prove (within the language itself) that $\bigcirc$ is an applicative functor using the obvious morphisms indicated above.

A potential law that seems impossible to prove is $\bigcirc\bigcirc A \to \bigcirc A$. This law holds of squash types in Nuprl. We could definitely prove this if we allow token type target erasure in $\bigcirc$-elimination (by instantiating $case_{\bigcirc}$ at $c = \bigcirc A$). In this case, $\bigcirc$ becomes a monad (with monad laws provable within the language itself). Although monads have proved useful in functional programming, the usefulness of this particular monad seems suspect. Perhaps more importantly one can prove, given squash target erasure, that $\bigcirc\bigcirc A$ and $\bigcirc A$ are isomorphic types.

However, we have eschewed token type target erasure because of its bad meta-theoretical properties. A more restricted form of token type target erasure that only holds for inductive types with no type indices (but perhaps type parameters)

would allow $\bigcirc$ to be a monad while precluding target erasure for the type equal, which was the basis of the problematic example in Section 5.2.3. The question remains, however, whether this restricted form of target erasure admits a similarly problematic example.

Note that the squash types in Nuprl satisfy many more laws than our $\bigcirc$ (see Caldwell [15] Section 3.3). This undoubtedly has to do with the fact that Nuprl is an extensional type theory wherein the notion of definitional equality is much stronger than mere β-conversion. In contrast, this dissertation deals only with intensional type theory.

5.5 COMPARISON WITH NUPRL AND COQ

Squash types (and the fact that they are definable in terms of erasure annotations) demonstrate another way that our approach is *flexible*: The squash type serves to distinguish between non-computational and computational versions of *the same type*, so that distinct computational and non-computational versions of A need not be defined separately.

We have seen that a primitive distinction between non-computational and computational function space is more expressive than the addition of a squash type to the language, since the former has more liberal rules for non-computational dependent function space formation.

Also, it seems that programs using $\Rightarrow$ will have more opportunities for erasure than those using $\bigcirc$. To see why this is, look at the isomorphism $A \Rightarrow B \cong \bigcirc A \to B$. Inhabitants of the type $A \Rightarrow B$ are inherently more concise than those of $\bigcirc A \to B$ at run-time, as the former need not ever be applied, but the latter do.

The universe hierarchy of Coq is divided up as shown in Figure 5.4. The universes Set and Prop are more or less logically equivalent. The reason for distinguishing between them is to express a distinction between computationally relevant and irrelevant portions of a program. Proofs (i.e., inhabitants of propositions) are

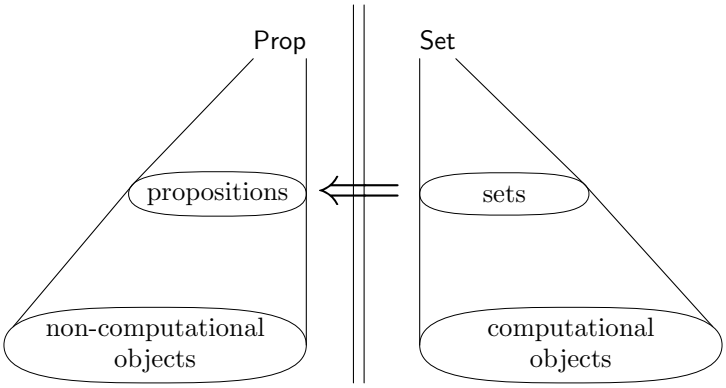


Figure 5.4: Division of computational and non-computational entities into two type universes

allowed to depend computationally on programs (i.e., inhabitants of sets), but programs may depend on proofs only in a very limited set of circumstances.[2] The arrow in the diagram represents the allowable direction of computational dependence. In type theory, computation happens when the introduction and elimination forms for a type come together and cancel each other out. Therefore the prohibition of programs depending on proofs is expressed in rules prohibiting the elimination of proofs (i.e., non-computational entities) while constructing programs (i.e., computational entities).

However, the purpose of the type hierarchy is not to distinguish between computational and non-computational data, but rather to outline the conceptual landscape of types and the allowable logical dependencies between levels they may express. In our opinion, the computational/non-computational distinction does not belong in the type hierarchy, but within the language of types itself. Using squash types, one may achieve the same separation of computational and non-computational types within a single universe, as depicted in Figure 5.5. In this

[2]These limited circumstances are closely related to empty type target erasure and token type target erasure.

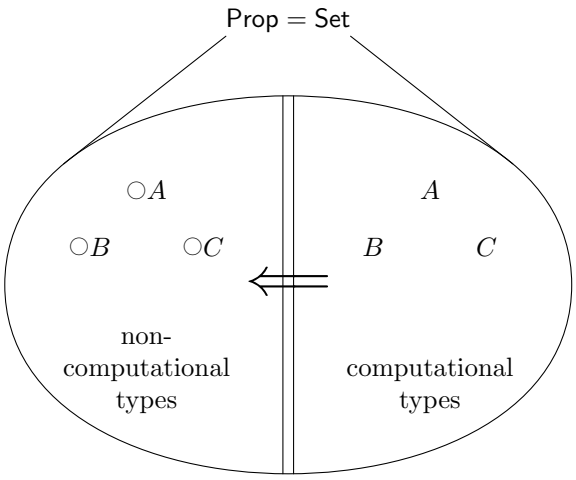


Figure 5.5: Division of computational and non-computational entities inside a single type universe using squash types

setting, non-computational entities may depend on computational ones, as is evidenced by the Law of Forgetfulness ($A \to \bigcirc A$). However, computational entities may not depend on non-computational ones, as there is no way to prove $\bigcirc A \to A$ in general.

In the solution based on squash types, an isomorphism between $\bigcirc\bigcirc A$ and $\bigcirc A$ would indicate that there is not an infinite hierarchy of non-computational types, non-computationally non-computational types, etc. The $\bigcirc$ modality divides the language of types into exactly two halves, the squash types and the non-squash types. The difficulty in proving such an isomorphism is discussed in the previous section.

It is a generally accepted rule of language design that implementations of distinct language features should have as few interactions as possible, so that they may be combined in arbitrarily complex ways. The universe hierarchy and the mechanism for distinguishing computationally relevant and irrelevant portions of a program should be completely orthogonal concerns in a fully satisfying language

design. We believe that the Coq approach of grafting a distinction between non-computational and computational entities onto the universe hierarchy violates this principle.

In contrast, the approach outlined in this dissertation is more satisfying in this respect. The consequences of our approach are

- increased flexibility in that the same type A may be used in both in computationally relevant and irrelevant ways. This flexibility is available both for function parameters ($A \to B$ versus $A \Rightarrow B$) and function results (A versus $\bigcirc A$).

- the distinction between computational and non-computational entities may be applied anywhere in the type hierarchy. This means our approach lends itself to language extensions supporting intensional polymorphism, or run-time inspection of types.

- the language design does not encourage the erroneous idea that all proofs and propositions are computationally irrelevant and all non-proofs are computationally relevant. In our view, the question of whether or not a type can be interpreted as a proposition is independent of whether or not values of that type may be inspected during a computation.

We find these to be compelling advantages over the Coq approach.

5.6 CONCLUSIONS

In this section we have explored the design space of a programming language with both erasure annotations and inductively defined datatypes. There are two ways in which erasure annotations affect the implementation of inductive types

- Use of the non-computational function space in the declared type of a constructor is seen to indicate an omission from the run-time representation of

that datatype.

- The index, parameter, and motive arguments of an eliminator of an inductive type are always computationally irrelevant and may therefore safely be marked for erasure in the eliminator's type.

There are two special cases in which erasure of an eliminator's target argument seems warranted at first. In each case, however, the additional erasure has unfortunate consequences on the meta-theory of post-erasure evaluation.

- Empty type target erasure — When the run-time representation of a type is equivalent to $\bot$, the trivial type with zero inhabitants, target erasure inadvertently introduces additional post-erasure *normal forms* at certain types involving contradictory assumptions.

- Token type target erasure — When the run-time representation of a type is equivalent to $\top$, the trivial type with one inhabitant, target erasure inadvertently introduces additional post-erasure *non-normalizing terms* at certain types involving contradictory assumptions.

For these reasons, we abandon all target argument erasure for eliminators.

Sum types of various strengths can be defined by varying erasability of two of the arguments of the lone data constructor of the sum type. Among these include weak sums (a.k.a. weak existentials) and subset types. In each case, the types so defined seem to have the essential characteristics of the types from the literature whose names they share.

Finally, a squash type is defined that seems to have the essential properties of a type by the same name studied in the setting of Martin-Löf's intensional type theory. Use of the squash type in conjunction with subset types indicates that the latter can be given a reasonable implementation in intensional type theory.

The squash type, though no replacement for the more expressive feature of non-computational dependent function spaces, seems to provide a mechanism for distinguishing between computational and non-computational types within a single universe as opposed to the two universe mechanism in Coq. Whereas non-computational functions are useful for specifying when a function argument is computationally irrelevant, squash types are useful for specifying when a function *result* is non-computational.

Chapter 6

PROOF IRRELEVANCE

In a dependently typed language, the conversion typing rule reflects the semantics of the language back into its type system.

$$\textsc{Conv}\quad \frac{\Gamma \vdash M :^{\mathsf{r}} A \qquad \Gamma \vdash B :^{\mathsf{c}} s \qquad A =_\beta B}{\Gamma \vdash M :^{\mathsf{r}} B}$$

Two terms that reduce to the same normal form are considered definitionally equal and the type system can not distinguish between them as subterms of types.

In EPTS, however, there are two notions of operational semantics. The Conv rule of EPTS reflects the default semantics rather than the erasure semantics. In a sense EPTS, when considered with an erasure semantics, is a sort of hybrid language in which definitional equality does not reflect the full semantics of the language. We may recover internal consistency in this area by modifying the Conv rule as follows:

$$\textsc{Conv}^\bullet\quad \frac{\Gamma \vdash M :^{\mathsf{r}} A \qquad \Gamma \vdash B :^{\mathsf{c}} s \qquad A^\bullet =_\beta B^\bullet}{\Gamma \vdash M :^{\mathsf{r}} B}$$

We choose the name EPTS$^\bullet$ for the resulting EPTS variant.

The notion of definitional equality in EPTS$^\bullet$ is more permissive than that of EPTS, so that more pairs of terms are considered definitionally equal. The remainder of this chapter investigates the expressiveness of EPTS$^\bullet$ as compared to EPTS when both are extended with inductive types, as well as the meta-theory of

pure EPTS$^\bullet$. In particular, we will see that EPTS$^\bullet$ admits an elective notion of proof irrelevance.

6.1 EXTRA EXPRESSIVENESS OF CONV$^\bullet$

What sorts of programs are accepted by the type system of EPTS$^\bullet$ that are not accepted by the type system of EPTS? The only difference is that definitional equality in EPTS$^\bullet$ is more permissive than definitional equality in EPTS. This extra permissiveness has three broad consequences of which we are aware.

1. Elective Proof Irrelevance (Section 6.1.1)
2. Internalized Behavioral Uniformity Principle (Section 6.1.2)
3. Provability of Streicher's K "axiom" (Section 6.1.3)

Note: All the code in this section is type-checked using the rule Conv$^\bullet$ rather than Conv.

6.1.1 Elective Proof Irrelevance

Recall the type subset A B from Section 5.3.3. This type captures the notion of a "type comprehension" — the type of all elements x of type A for which the the type B x is inhabited (i.e., the proposition B x holds, when B x is considered a proposition). For example subset nat *even* is the type of even natural numbers.

Consider two inhabitants M = member 8 @P and N = member 8 @Q of type subset nat *even*. Given our informal understanding of subsets, one may reasonably consider M and N to be the *same* inhabitant of subset nat *even* — they both represent the even number 8 — even when P and Q are two different proofs of 8's evenness. Indeed, since M and N both erase to the same term member 8, they are definitionally equal given the modified conversion rule Conv$^\bullet$. However, the

original EPTS conversion rule distinguishes between M and N since the two terms are not β-convertible before erasure.

In system with inductive types and the CONV$^\bullet$ typing rule, one can prove

$$(a : *) \Rightarrow (b : a \to *) \Rightarrow (s, t : \mathsf{subset}\ a\ b) \to$$
$$\mathsf{equal}\ a\ (\mathit{inject}\ @a\ @b\ s)\ (\mathit{inject}\ @a\ @b\ t) \to \mathsf{equal}\ (\mathsf{subset}\ a\ b)\ s\ t$$

where *inject* is the previously defined injection from subset a b back to a.

The feature of considering two proofs definitionally equal whenever they prove the same thing, regardless of how they prove it, is called *proof irrelevance*. The subset example shows that CONV$^\bullet$ affords EPTS$^\bullet$ with a form of proof irrelevance: proofs of propositions that are marked for erasure will never be distinguished by the post-erasure β-conversion notion of definitional equality.

We call this form of proof irrelevance *elective* because the programmer determines which portions of a data structure are irrelevant in terms of the conversion check by means of erasure annotations placed on the types of data constructors. This notion stands in contrast to what may be termed a *universal* notion of proof irrelevance, whereby any two proofs of any proposition are always considered computationally equivalent. In Coq, for example, where proofs are identified by the sort of their type, a universal proof irrelevance principle may be introduced by means of the following axiom

$$\forall p{:}\mathsf{Prop}.\, \forall a, b{:}p.\, a =_p b$$

This axiom says that any two proofs a and b of any particular proposition p are considered to be equal.

Using squash types, we may prove a similar result, namely that all inhabitants

of any particular squash type are identified.

$$
\begin{array}{lcl}
\mathit{poof\_irrelevance} & : & (a : \mathit{type}) \Rightarrow (u, v : \bigcirc a) \rightarrow \mathsf{equal}\ (\bigcirc a)\ u\ v \\
\mathit{poof\_irrelevance} & = & \lambda\!\!\lambda a.\, \lambda u.\, \lambda v. \\
 & & \quad \mathit{elim}_{\bigcirc}\ @a\ u\ @(\lambda u.\, \mathsf{equal}\ (\bigcirc a)\ u\ v)\ (\lambda\!\!\lambda x. \\
 & & \quad\quad \mathit{elim}_{\bigcirc}\ @a\ v\ @(\lambda v.\, \mathsf{equal}\ (\bigcirc a)\ (\mathsf{poof}\ @a\ @x)\ v)\ (\lambda\!\!\lambda y. \\
 & & \quad\quad\quad \mathsf{refl}\ @(\bigcirc a)\ @(\mathsf{poof}\ @a\ @x)))
\end{array}
$$

Again, the type-correctness of this term depends essentially on the Conv$^\bullet$ typing rule so that the type $\mathsf{equal}\ (\bigcirc a)\ (\mathsf{poof}\ @a\ @x)\ (\mathsf{poof}\ @a\ @x)$ of the application of refl is definitionally equal to the required type $\mathsf{equal}\ (\bigcirc a)\ (\mathsf{poof}\ @a\ @x)\ (\mathsf{poof}\ @a\ @y)$.

6.1.2 Uniformity Principle

However, elective proof irrelevance is not the only use of the modified conversion rule. The Conv$^\bullet$ rule also identifies the following terms from the EPTS with the underlying PTS specification of System F:

$$
\begin{array}{rcl}
M & = & \lambda x{:}((a : \star) \Rightarrow a \rightarrow a).\, x\ @((a : \star) \Rightarrow a \rightarrow a)\ x \\
N & = & \lambda x{:}((a : \star) \Rightarrow a \rightarrow a).\, \lambda\!\!\lambda a{:}\star.\, x\ @(a \rightarrow a)\ (x\ @a)
\end{array}
$$

In this example, M and N both have type $((a : \star) \Rightarrow a \rightarrow a) \rightarrow ((a : \star) \Rightarrow a \rightarrow a)$ and both erase to $\lambda x.\, x\ x$. In this case, there seems to be an interesting interplay between impredicativity and type-erasure. Using relational parametricity (see Section 2.4.2) one can prove that M and N are extensionally equivalent. This example indicates that Conv$^\bullet$ places our system somewhere between intensional and extensional type theory.

6.1.3 Streicher's K "Axiom" is Provable

In 1993, Thomas Streicher and Thorsten Altenkirch introduced the idea of *uniqueness of identity proofs*, whereby one may prove that any proof of $\mathsf{equal}\ a\ x\ x$ is

equivalent to $\mathsf{refl}\ @a\ @x$. This proposition is stated by the following type:

$$(a : *) \Rightarrow (x : a) \Rightarrow (t : \mathsf{equal}\ a\ x\ x) \rightarrow \mathsf{equal}\ (\mathsf{equal}\ a\ x\ x)\ (\mathsf{refl}\ @a\ @x)\ t$$

Though this result seems like it should be straightforward to prove, it resisted all attempts at a proof, and eventually was shown to be unprovable in type theory by model theoretic means [41]. However, it is satisfied by most known models of type theory, so we might reasonably accept it as an axiom.

This principle is not merely of theoretical interest. It is essential for integrating into type theory the programming style of function definition by pattern matching as is usual in functional programming languages [26, 58]. This ought to be welcome news to anyone who attempted to read code written in the more logically motivated "eliminator style" from the previous chapter. Just as functional programmers prefer not to write all their programs in terms of "fold-like" operators, we prefer some options besides "eliminator style" programming.

The good news we present here is that uniqueness of identity proofs is in fact provable in our system. The extra flexibility afforded by CONV$^\bullet$ enriches our language enough to prove this elusive proposition. The proof follows an argument of Thorsten Altenkirch as related by Thomas Streicher [88, Section 1.5]. The argument goes like this: Assuming $a : *$, $x : a$, and $t : \mathsf{equal}\ a\ x\ x$, one can prove the following two propositions:

(1) $\mathsf{equal}\ (\mathsf{equal}\ a\ x\ x)\ (\mathsf{refl}\ @a\ @x)\ (\mathit{cast}\ @a\ @x\ @x\ t\ @(\lambda z{:}a.\,\mathsf{equal}\ a\ z\ x)\ t)$

(2) $\mathsf{equal}\ (\mathsf{equal}\ a\ x\ x)\ (\mathit{cast}\ @a\ @x\ @x\ t\ @(\lambda z{:}a.\,\mathsf{equal}\ a\ x\ x)\ t)\ t$

Terms $\mathit{thorsten}_1$ and $\mathit{thorsten}_2$ in Figure 6.1 prove these two propositions. Propositions (1) and (2) contain subterms

$$\lambda z{:}a.\,\mathsf{equal}\ a\ z\ x \qquad \text{and} \qquad \lambda z{:}a.\,\mathsf{equal}\ a\ x\ x,$$

respectively. Although these two terms are distinct, they both occur in computationally irrelevant portions of an enclosing *cast* expression, so we see that they

$$
\begin{array}{lcl}
\mathit{thorsten}_1 & : & (a : *) \Rightarrow (x : a) \Rightarrow (t : \mathsf{equal}\ a\ x\ x) \rightarrow \\
 & & \mathsf{equal}\ (\mathsf{equal}\ a\ x\ x) \\
 & & \quad (\mathsf{refl}\ @a\ @x)\ (\mathit{cast}\ @a\ @x\ @x\ t\ @(\lambda z{:}a.\, \mathsf{equal}\ a\ z\ x)\ t) \\
\mathit{thorsten}_1 & = & \lambda\!\!\lambda a{:}*.\, \lambda\!\!\lambda x{:}a.\, \lambda t{:}\mathsf{equal}\ a\ x\ x. \\
 & & \mathit{elim}_{\mathsf{equal}}\ @a\ @x\ @x\ t \\
 & & \quad @(\lambda y{:}a.\, \lambda q{:}\mathsf{equal}\ a\ x\ y. \\
 & & \qquad \mathsf{equal}\ (\mathsf{equal}\ a\ y\ y) \\
 & & \qquad\quad (\mathsf{refl}\ @a\ @y) \\
 & & \qquad\quad (\mathit{cast}\ @a\ @x\ @y\ q\ @(\lambda z{:}a.\, \mathsf{equal}\ a\ z\ y)\ q)) \\
 & & \quad (\mathsf{refl}\ @(\mathsf{equal}\ a\ x\ x)\ @(\mathsf{refl}\ @a\ @x)) \\
\mathit{thorsten}_2 & : & (a : *) \Rightarrow (x : a) \Rightarrow (t : \mathsf{equal}\ a\ x\ x) \rightarrow \\
 & & \mathsf{equal}\ (\mathsf{equal}\ a\ x\ x) \\
 & & \quad (\mathit{cast}\ @a\ @x\ @x\ t\ @(\lambda z{:}a.\, \mathsf{equal}\ a\ x\ x)\ t)\ t \\
\mathit{thorsten}_2 & = & \lambda\!\!\lambda a{:}*.\, \lambda\!\!\lambda x{:}a.\, \lambda t{:}\mathsf{equal}\ a\ x\ x. \\
 & & \mathit{elim}_{\mathsf{equal}}\ @a\ @x\ @x\ t \\
 & & \quad @(\lambda y{:}a.\, \lambda q.\, \mathsf{equal}\ a\ x\ y \\
 & & \qquad \mathsf{equal}\ (\mathsf{equal}\ a\ x\ x) \\
 & & \qquad\quad (\mathit{cast}\ @a\ @x\ @y\ q\ @(\lambda z{:}a.\, \mathsf{equal}\ a\ x\ x)\ t) \\
 & & \qquad\quad t) \\
 & & \quad (\mathsf{refl}\ @(\mathsf{equal}\ a\ x\ x)\ @t)
\end{array}
$$

Figure 6.1: Lemmas in Altenkirch's proof that a behavioral uniformity principle implies uniqueness of identity proofs

$$
\begin{array}{rcl}
\mathit{trans} & : & (a:*) \Rightarrow (x,y,z:a) \Rightarrow \mathsf{equal}\ a\ x\ y \rightarrow \mathsf{equal}\ a\ y\ z \rightarrow \mathsf{equal}\ a\ x\ z \\
\mathit{trans} & = & \lambda\!\!\lambda a,x,y,z.\,\lambda s,t.\,\mathit{cast}\ @a\ @y\ @z\ t\ @(\lambda z.\,\mathsf{equal}\ a\ x\ z)\ s \\
\mathit{urip} & : & (a:*) \Rightarrow (x:a) \Rightarrow (t:\mathsf{equal}\ a\ x\ x) \rightarrow \\
 & & \quad \mathsf{equal}\ (\mathsf{equal}\ a\ x\ x)\ (\mathsf{refl}\ @a\ @x)\ t \\
\mathit{urip} & = & \lambda\!\!\lambda a{:}*.\,\lambda\!\!\lambda x{:}a.\,\lambda t{:}\mathsf{equal}\ a\ x\ x. \\
 & & \quad \mathit{trans}\ @(\mathsf{equal}\ a\ x\ x) \\
 & & \quad\quad @(\mathsf{refl}\ @a\ @x)\ @(\mathit{cast}\ @a\ @x\ @x\ t\ @(\lambda z{:}a.\,\mathsf{equal}\ a\ z\ x)\ t)\ @t \\
 & & \quad\quad (\mathit{thorsten}_1\ @a\ @x\ t)\ (\mathit{thorsten}_2\ @a\ @x\ t)
\end{array}
$$

Figure 6.2: Proof of Uniqueness of (Reflexive) Identity Proofs

have the same behavior. Altenkirch could only go this far. He said if there is some way to consider as definitionally equal types that differ only by such behaviorally equivalent types, then one could then prove the uniqueness of identity proofs by transitivity of equality.

The rule Conv$^\bullet$ gives us exactly this power. The two cast expressions in (1) and (2) both erase to *cast* t t, and so according to Conv$^\bullet$, these two terms are definitionally equal. Given this observation, one may easily prove by transitivity that any reflexive identity proof of type $\mathsf{equal}\ a\ x\ x$ equals the canonical proof $\mathsf{refl}\ @a\ @x$. The proof term *urip* is shown in Figure 6.2. The well-formedness of *urip* relies essentially on this aspect of definitional equality.

Using the uniqueness of identity proofs, one can define an alternative elimination rule for equal that only operates "along the diagonal", known as Streicher's

K eliminator.

$$\begin{array}{lcl} \mathit{streichersK} & : & (a : *) \Rightarrow (x : a) \Rightarrow (t : \mathsf{equal}\ a\ x\ x) \rightarrow \\ & & (p : \mathsf{equal}\ a\ x\ x \rightarrow *) \Rightarrow p\ (\mathsf{refl}\ @a\ @x) \rightarrow p\ t \\ \mathit{streichersK} & = & \lambda\!\!\lambda a, x.\ \lambda t.\ \lambda\!\!\lambda p.\ \lambda m. \\ & & \mathit{cast}\ @(\mathsf{equal}\ a\ x\ x)\ @(\mathsf{refl}@a@x)\ @t\ (\mathit{urip}\ @a\ @x\ t)\ @p\ m \end{array}$$

The erasure of *streichersK* normalizes to

$$\mathit{streichersK}^\bullet = \lambda t.\ \lambda m.\ \mathit{elim}_{\mathsf{equal}}\ (\mathit{elim}_{\mathsf{equal}}\ (\mathit{elim}_{\mathsf{equal}}\ t\ \mathsf{refl})\ (\mathit{elim}_{\mathsf{equal}}\ t\ \mathsf{refl}))\ m$$

Therefore, the post-erasure reduction behavior of *streichersK* is

$$\mathit{streichersK}^\bullet\ \mathsf{refl}\ M \quad \rightarrow_\beta^* \quad M$$

Note that this behavior is identical to that of the standard eliminator for equal.

$$\mathit{elim}_{\mathsf{equal}}\ \mathsf{refl}\ M \quad \rightarrow_\beta \quad M$$

Some other consequences the uniqueness of identity proofs can be found in in the Coq standard library[1]. Two of these consequences are proved in Figure 6.3. Another consequence is that McBride's heterogeneous equality is programmable in our language [58, Section 5.1].

6.2 NON-COMPUTATIONAL AXIOMS

In a language with Conv$^\bullet$, after a definition $x = M : A$ is type-checked, the non-computational parts of M and A will never be needed again. There are only two circumstances in which the definition of x may be required after x is defined: (1) in order to type-check some term mentioning x in the remainder of the program, and (2) in order to evaluate some other term mentioning x at run-time. Given

[1]See `Coq.Logic.EqdepFacts` at the URL `http://coq.inria.fr/library/`.

Substitution Invariance

$$
\begin{array}{lcl}
\mathit{subst\_invariance} & : & (a : *) \Rightarrow (x : a) \Rightarrow (p : a \to *) \Rightarrow (m : p\ x) \to \\
 & & (t : \mathsf{equal}\ a\ x\ x) \to \mathsf{equal}\ (p\ x)\ (\mathit{cast}\ @a\ @x\ @x\ t\ @p\ m)\ m \\
\mathit{subst\_invariance} & = & \lambda\!\!\lambda a{:}*.\ \lambda\!\!\lambda x{:}a.\ \lambda\!\!\lambda p{:}a \to *.\ \lambda m{:}p\ x.\ \lambda t{:}\mathsf{equal}\ a\ x\ x. \\
 & & \mathit{streichersK}\ @a\ @x\ t \\
 & & \quad @(\lambda t.\ \mathsf{equal}\ (p\ x)\ (\mathit{cast}\ @a\ @x\ @x\ t\ @p\ m)\ m) \\
 & & \quad (\mathsf{refl}\ @(p\ x)\ @m)
\end{array}
$$

Uniqueness of (not necessarily reflexive) identity proofs

$$
\begin{array}{lcl}
\mathit{uip} & : & (a : *) \Rightarrow (x, y : a) \Rightarrow (t, s : \mathsf{equal}\ a\ x\ y) \to \mathsf{equal}\ (\mathsf{equal}\ a\ x\ y)\ t\ s \\
\mathit{uip} & = & \lambda\!\!\lambda a{:}*.\ \lambda\!\!\lambda x{:}a.\ \lambda\!\!\lambda y{:}a.\ \lambda t{:}\mathsf{equal}\ a\ x\ y. \\
 & & \mathit{elim}_{\mathsf{equal}}\ @a\ @x\ @y\ t \\
 & & \quad @(\lambda z{:}a.\ \lambda t{:}\mathsf{equal}\ a\ x\ z.\ (s : \mathsf{equal}\ a\ x\ z) \to \mathsf{equal}\ (\mathsf{equal}\ a\ x\ z)\ t\ s) \\
 & & \quad (\mathit{urip}\ @a\ @x)
\end{array}
$$

Figure 6.3: Two consequences of Streicher's K eliminator and Uniqueness of Reflexive Identity Proofs

an erasure semantics, (2) only happens after erasure, so that we need $M^\bullet$ rather than M. In a language with the CONV$^\bullet$ conversion rule, the notion of definitional equality is post-erasure β-conversion. Therefore (1) only ever requires $A^\bullet$ and $M^\bullet$.

Consequently, after type-checking a definition of the form $x = \mathsf{poof}\ @A\ @M : \bigcirc A$, we need only store the erasure of the definition of x, namely $x = \mathsf{poof} : \bigcirc(A^\bullet)$. Any subsequent evaluation of x will immediately return the value poof.

Now suppose one would like to introduce an axiom A in the non-computational fragment of the language. We may do so by simply introducing the run-time definition $my\_axiom = \mathsf{poof} : \bigcirc A$ into the global typing context. In this case no proof M is given. We simply type-check $\bigcirc A$ to make sure it is a valid type and continue checking the rest of the program. We call *my_axiom* a *non-computational axiom*. From here on out, we use the syntax

$$\underline{\mathsf{axiom}}\quad my\_axiom\quad :\quad \bigcirc A$$

to introduce the non-computational axiom A.

6.2.1 Axioms for Classical Reasoning

For example, one may perform classical reasoning in the $\bigcirc$ fragment of the language by introducing any one of the following axioms:

$$\underline{\mathsf{axiom}}\quad excluded\_middle\quad :\quad \bigcirc((a : *) \Rightarrow a \vee not\ a) \tag{6.1}$$

$$\underline{\mathsf{axiom}}\quad non\_contradiction\quad :\quad \bigcirc((a : *) \Rightarrow not\ (not\ a) \to a) \tag{6.2}$$

$$\underline{\mathsf{axiom}}\quad pierce\quad :\quad \bigcirc((a : *) \Rightarrow (not\ a \to a) \to a) \tag{6.3}$$

Where $not : * \to *$ is defined as $not = \lambda x.\, x \to \bot$ and $\vee$ (i.e., propositional disjunction) is defined as a parameterized inductive type as usual. Since each axiom has the same computational content, namely the constructor poof, there is no need to extend the language with control structures to evaluate terms using these classical axioms.

6.2.2 The ○-flattening Axiom

Another interesting axiom one might add (in the ○ fragment) is that $\bigcirc A$ implies A for any type A:

$$\underline{\mathsf{axiom}} \quad \mathit{flat} \quad : \quad \bigcirc((a : *) \Rightarrow \bigcirc a \to a) \tag{6.4}$$

This axiom may be used in the ○ fragment of the language. Effectively, it states that $\bigcirc A$ and A are logically equivalent underneath ○, so that there is no infinite hierarchy of degrees of non-computationality. The infinite sequence of types

$$A,\ \bigcirc A,\ \bigcirc\bigcirc A,\ \bigcirc\bigcirc\bigcirc A,\ \cdots \qquad \text{collapses to} \qquad A,\ \bigcirc A,\ \bigcirc A,\ \bigcirc A,\ \cdots,$$

thus explaining why the axiom in question is named *flat*.

An immediate consequence of *flat* is that ○ becomes a monad. In the formulation of monads in terms of *map*, *return*, and *join*, the only difficult function to define is *join*. Using *flat*, we may define *join* as follows.

$$\begin{aligned}
\mathit{join} \quad &: \quad (a : *) \Rightarrow \bigcirc\bigcirc a \to \bigcirc a \\
\mathit{join} \quad &= \quad \lambda\!\!\lambda a.\, \lambda m. \\
&\qquad \mathit{elim}_{\bigcirc}\ @((a : *) \Rightarrow \bigcirc a \to a)\ \mathit{flat}\ @(\lambda_.\, \bigcirc a)\ (\lambda\!\!\lambda \mathit{run}. \\
&\qquad\quad \mathit{elim}_{\bigcirc}\ @(\bigcirc a)\ m\ @(\lambda_.\, \bigcirc a)\ (\lambda\!\!\lambda x. \\
&\qquad\qquad \mathsf{poof}\ @a\ @(\mathit{run}\ @a\ x)))
\end{aligned}$$

Once *join* is defined, the monad laws are trivial to prove due to *poof_irrelevance*.

The pattern $\mathit{elim}_{\bigcirc}\ @((a : *) \Rightarrow \bigcirc a \to a)\ \mathit{flat}\ @(\lambda\_.\, \bigcirc a)\ (\lambda\!\!\lambda \mathit{run}.\, \cdots)$ in the definition of *join* occurs over and over again when using *flat*, so we abstract it out into the following function:

$$\begin{aligned}
\mathit{withflat} \quad &: \quad (c : *) \Rightarrow (((a : *) \Rightarrow \bigcirc a \to a) \Rightarrow c) \to c \\
\mathit{withflat} \quad &= \quad \lambda\!\!\lambda c.\, \lambda f.\, \mathit{elim}_{\bigcirc}\ @((a : *) \Rightarrow \bigcirc a \to a)\ \mathit{flat}\ @(\lambda_.\, c)\ f
\end{aligned}$$

Another consequence of *flat* is a version of the axiom of choice for subset types:

$$\begin{aligned} ac \;\; : \;\; & (a : *) \Rightarrow (b : a \to *) \Rightarrow (p : (x : a) \to b\ x \to *) \Rightarrow \\ & ((x : a) \to \mathsf{subset}\ (b\ x)\ (\lambda y.\, p\ x\ y)) \to \\ & \quad \mathsf{subset}\ ((x : a) \to b\ x)\ (\lambda f.\, (x : a) \to p\ x\ (f\ x)) \end{aligned}$$

Using a nicer notation for subsets, the proposition becomes.

$$((x{:}A) \to \{y : B(x) \mid P(x, y)\}) \quad \to \quad \{f : (x : A) \to B(x) \mid (x{:}A) \to P(x, f\ x)\}$$

This type corresponds to the following proposition: if for all $x : A$ there is some $y : B(x)$ such that $P(x, y)$ is true, then there is a function $f : (x : A) \to B(x)$ such that for all $x : A$, it is true that $P(x, f\ x)$. This formulation of the axiom of choice may be proved using *withflat* as follows:

$$\begin{aligned} ac \;\; = \;\; & \lambda\!\!\lambda a, b, p.\, \lambda g{:}((x : a) \to \mathsf{subset}\ (b\ x)\ (\lambda y.\, p\ x\ y)). \\ & \mathit{withflat} \\ & \quad @(\mathsf{subset}((x : a) \to b\ x)\ (\lambda f.\, (x : a) \to p\ x\ (f\ x))) \\ & \quad (\lambda\!\!\lambda \mathit{run}{:}((a : *) \Rightarrow \bigcirc a \to a). \\ & \qquad \mathsf{member}\ @((x : a) \to b\ x)\ @(\lambda f.\, (x : a) \to p\ x\ (f\ x)) \\ & \qquad\quad (\lambda x.\, \mathit{witness}\ @(b\ x)\ @(\lambda y.\, p\ x\ y)\ (g\ x)) \\ & \qquad\quad @(\lambda x.\, \mathit{run} \\ & \qquad\qquad @(p\ x\ (\mathit{witness}\ @(b\ x)\ @(\lambda y.\, p\ x\ y)\ (g\ x))) \\ & \qquad\qquad (\mathit{evidence}\ @(b\ x)\ @(\lambda y.\, p\ x\ y)\ (g\ x)))) \end{aligned}$$

Without *flat*, the closest thing one can prove to the axiom of choice for subset types is the proposition

$$\begin{aligned} & (a : *) \Rightarrow (b : a \to *) \Rightarrow (p : (x : a) \to b\ x \to *) \Rightarrow \\ & \quad ((x : a) \to \mathsf{subset}\ (b\ x)\ (\lambda y.\, p\ x\ y)) \to \\ & \qquad \mathsf{subset}\ ((x : a) \to b\ x)\ (\lambda f.\, (x : a) \to \bigcirc(p\ x\ (f\ x))) \end{aligned}$$

containing an unfortunate $\bigcirc$ in the conclusion.

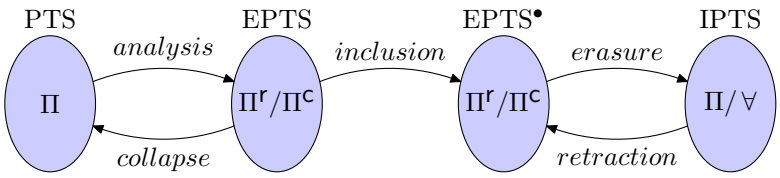


Figure 6.4: Relationships between four PTS variants

6.3 EPTS•

In this section, we formally investigate the properties of EPTS•. We will see that it is essentially an explicitly typed version of IPTS.

6.3.1 Meta-theory of Erasure

EPTS and EPTS• have identical syntax. Only their type systems differ. For this reason, the erasure operation defined in Section 3.3 also maps EPTS• terms to IPTS terms.

Happily, each meta-theoretical result for EPTS that appears in Figure 3.4 remains valid for EPTS•. The proofs for these results change only slightly.

- The Π-Form case of the subject reduction proof for EPTS• depends on the result that erasure respects reductions. No such dependency exists in the EPTS meta-theory.

- The Conv case of the proof that erasure respects types in EPTS depends on the fact that erasure respects reductions. No such dependency exists in the EPTS• meta-theory, because the Conv• rule is stronger.

So it happens that some proofs become simpler, while others become more difficult. The updated set of proof dependencies remains acyclic, however.

One significant property about EPTS• is that it is strongly normalizing whenever the same can be said of the underlying IPTS. This follows immediately from

the subject reduction property and the fact that erasure preserves both reductions and types.

Lastly, we show that EPTS$^\bullet$ is roughly equivalent to IPTS because there is a direct mapping from IPTS derivations to EPTS$^\bullet$ derivations, for which erasure is a retraction.

6.3.2 Equivalence with IPTS

We have already seen that erasure maps well-typed EPTS$^\bullet$ terms to well-typed IPTS terms. In this section, we show that typing derivations in IPTS also map back to typing derivations in EPTS$^\bullet$.

In order to state the result, we first need a notion of well-formed contexts. A context is well-formed when every type in it is well-formed as a type in the preceding portion of the overall context.

Definition 6.3.1 (Well-formed contexts) $\boxed{\vdash \Gamma\ \mathsf{ctx}}$

$$\textsc{OkNil}\ \frac{}{\vdash \varepsilon\ \mathsf{ctx}} \qquad \textsc{OkExt}\ \frac{\vdash \Gamma\ \mathsf{ctx} \qquad \Gamma \vdash A :^c s}{\vdash \Gamma, x{:}^r A\ \mathsf{ctx}}$$

The reset operation on typing contexts preserves well-formedness.

Lemma 6.3.2 *If* $\vdash \Gamma\ \mathsf{ctx}$ *then* $\vdash \Gamma^\circ\ \mathsf{ctx}$.

Now we can state the property that IPTS typing derivations map back to EPTS$^\bullet$ ones. The proof has two main parts. First, we prove that any typing derivation under the erasure of a well-formed EPTS$^\bullet$ context Γ maps back to an EPTS$^\bullet$ typing derivation under Γ. The only stipulation is that when we want the resulting EPTS$^\bullet$ judgment to be in r-mode, it is required that all free variables of the subject M of the conclusion typing judgment are r-marked in Γ. Secondly we prove that there is such an EPTS$^\bullet$ context corresponding to every context in a valid IPTS typing judgment.

Theorem 6.3.3 (Elaboration in r mode)

$$\frac{\vdash \Gamma\ \mathsf{ctx} \qquad \Gamma^\bullet \vdash M : A \qquad FV(M) \subseteq RV(\Gamma)}{(\exists\, M'\, A') \qquad \Gamma \vdash M' :^r A' \qquad M'^\bullet = M \qquad A'^\bullet = A}$$

Corollary 6.3.4 (Elaboration in c mode)

$$\frac{\vdash \Gamma\ \mathsf{ctx} \qquad \Gamma^\bullet \vdash M : A}{(\exists\, M'\, A') \qquad \Gamma \vdash M' :^{\mathsf{c}} A' \qquad M'^\bullet = M \qquad A'^\bullet = A}$$

Then we show that can extract a well-formed EPTS$^\bullet$ context from the context of any typing judgment in IPTS.

Lemma 6.3.5 (Context Elaboration)

$$\frac{\Gamma \vdash M : A}{(\exists\, \Gamma') \qquad \vdash \Gamma'\ \mathsf{ctx} \qquad \Gamma'^\bullet = \Gamma}$$

From these three lemmas, it immediately follows that for any mode τ, we may map any IPTS typing derivation back to an EPTS$^\bullet$ typing derivation in that mode.

Corollary 6.3.6 (Elaboration)

$$\frac{\Gamma \vdash M : A}{(\exists\, \Gamma'\, M'\, A') \qquad \Gamma' \vdash M' :^\tau A' \qquad \Gamma'^\bullet = \Gamma \qquad M'^\bullet = M \qquad A'^\bullet = A}$$

6.4 ERASABILITY ANALYSIS

How might we try to extend the constraint-generation process (of Section 4.2) for Conv? The original rule for testing convertibility of applications in head normal form was

$$\textsc{CongApp}\ \frac{\mathcal{C} \vdash M =_\beta M' \qquad \mathcal{D} \vdash N =_\beta N'}{\alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D} \vdash M@^\alpha N =_\beta M'@^{\alpha'} N'}.$$

However, if either α or α' is c, then erasure will prevent the conversion test between M' and N'. There are four cases to consider

1. $\alpha = \mathsf{r} = \alpha'$ — the generated constraint should be $\mathcal{C} \wedge \mathcal{D}$
2. $\alpha = \mathsf{c} = \alpha'$ — the generated constraint should be $\mathcal{C}$
3. $\alpha = \mathsf{c},\ \alpha' = \mathsf{r}$ — compare M with $M'@^{\mathsf{r}}N'$ to obtain the constraint $\mathcal{E}$, and then return $\mathcal{E}$
4. $\alpha = \mathsf{r},\ \alpha' = \mathsf{c}$ — compare $M@^{\mathsf{r}}N$ with M' to obtain the constraint $\mathcal{F}$, and then return $\mathcal{F}$

These considerations together yield the following rule

$$\textsc{CongApp}\quad \frac{\begin{array}{cc}\mathcal{C} \vdash M =_\beta M' & \mathcal{D} \vdash N =_\beta N' \\ \mathcal{E} \vdash M =_\beta M'@^{\mathsf{r}}N' & \mathcal{F} \vdash M@^{\mathsf{r}}N =_\beta M'\end{array}}{\begin{array}{l}(\neg\alpha \wedge \neg\alpha' \Rightarrow \mathcal{C} \wedge \mathcal{D}) \wedge (\alpha \wedge \alpha' \Rightarrow \mathcal{C}) \\ \quad \wedge\ (\alpha \wedge \neg\alpha' \Rightarrow \mathcal{E}) \wedge (\alpha \wedge \neg\alpha' \Rightarrow \mathcal{F})\end{array} \vdash M@^{\alpha}N =_\beta M'@^{\alpha'}N'}$$

The result of this rule will be to run conversion tests on every possible combination of c and r assignments to application annotation variables in neutral terms. For instance, comparing $x@^{\alpha_1}M_1@^{\alpha_2}M_2 \cdots @^{\alpha_m}M_m$ and $y@^{\beta_1}N_1@^{\beta_2}N_2 \cdots @^{\beta_n}N_n$ will require testing $n \cdot m$ different pairs (M_i, N_j) for conversion. This strategy seems hopelessly inefficient.

One possible way out is to sacrifice completeness, using the following simple rule that ignores the possibility that α may not equal α'.

$$\textsc{CongApp}\quad \frac{\mathcal{C} \vdash M =_\beta M' \qquad \mathcal{D} \vdash N =_\beta N'}{\alpha = \alpha' \wedge \mathcal{C} \wedge (\alpha \vee \mathcal{D}) \vdash M@^{\alpha}N =_\beta M'@^{\alpha'}N'}.$$

This restriction still allows the applications of elective irrelevance outlined above, but precludes the possibility of inferring convertibility of the exotic example from Section 6.1.2.

The constraint of the form $\alpha \vee \mathcal{D}$ may be transformed into conjunctive normal form (CNF) when $\mathcal{C}$ is already in CNF by distributing the $\vee$ over each $\wedge$ in the

conjunction $\mathcal{C}$. Furthermore, this operation preserves the invariant that each clause in a CNF constraint has *at most* one negated literal.

Of course, in an implementation of a constraint-generator following these rules, we would treat concrete annotations specially when they allow us to "short circuit" certain tests. Such optimizations are justified by instantiations of the previous rule, such as

$$\textsc{CongAppShortCircuit1}\ \frac{\mathcal{C} \vdash M =_\beta M'}{\alpha' \wedge \mathcal{C} \vdash M@^{\mathsf{c}} N =_\beta M'@^{\alpha'} N'} \quad \text{and} \quad \textsc{CongAppShortCircuit2}\ \frac{\mathcal{C} \vdash M =_\beta M'}{\alpha \wedge \mathcal{C} \vdash M@^{\alpha} N =_\beta M'@^{\mathsf{c}} N'}.$$

However, these rules over-optimize CongApp in the sense that we test N and N' not only to generate a constraint, but also to see whether such a constraint exists. In other words, N and N' may fail to be convertible altogether in the rule CongApp. In this case, CongApp is too strong and we need another rule asserting that both $\alpha = \mathsf{c}$ and $\alpha' = \mathsf{c}$.

$$\textsc{CongAppFail}\ \frac{\mathcal{C} \vdash M =_\beta M' \qquad \not\vdash N =_\beta N'}{\alpha \wedge \alpha' \wedge \mathcal{C} \vdash M@^{\alpha} N =_\beta M'@^{\alpha'} N'}.$$

The upshot is that a complete analysis is infeasible for EPTS$^\bullet$, but a heuristic and sound analysis is feasible. It remains to be seen whether the heuristic approach is practical for implementations of programming languages with erasure semantics based on EPTS$^\bullet$.

Chapter 7

RELATED WORK

In this chapter, we outline several bodies of related work and discuss how our research is related to them.

7.1 USELESS VARIABLE ELIMINATION

The simplest body of related work is on a problem known as *useless variable elimination* (UVE) for functional programming languages.

In 1991, Olin Shivers introduced UVE in his doctoral dissertation [84, Section 7.2]. UVE is a program analysis and optimization whereby variables whose values never affect the outcome of a computation are eliminated from the program. Shivers presents UVE as an application of his control flow analysis for functional programs. In a follow up workshop paper [85], he provides more details of how to implement UVE.

In 1999, Mitchell Wand and Igor Siveroni [94] formalized a constraint-based useless variable analysis, proved it sound, and then showed that correctness of the subsequent UVE step follows from soundness of the analysis. The presentation is much more precise than that of Shivers, but the algorithm is essentially the same. They note that UVE can be thought of as a form of "dead code elimination" where code is considered dead if it contributes nothing to the end result of a computation.

In 2000, Naoki Kobayashi showed how to do UVE for a typed language as a simple variation on the usual Hindley-Milner type inference algorithm [44]. His UVE algorithm is based on *pruning* — replacing subterms of the original program

with unit, the sole constructor of a unit type such as $\top$. To be useful, pruning must be followed by a unit removal phase to reduce time spent passing around unit values. The second phase consists of selectively applying the following type isomorphisms

$$\top \to B \cong B \qquad A \to \top \cong \top \qquad A \times \top \cong A \qquad \top \times B \cong B$$

The analysis phase does a type inference in a demand-driven way so that any subterm that may be assigned the type $\top$ is replaced with unit.

A useful feature of each UVE algorithm listed so far is that dead code identified by the analysis phase does not affect which parts of the code are marked as dead by the analysis. Consider the following example program and its pruned version, due to Wand and Siveroni [94]:

$$\begin{array}{lrl}
\underline{\mathsf{let}} & f_1 & = \lambda x.\, \lambda y.\, x \\
& f_2 & = \lambda x.\, \lambda y.\, x + x \\
& f_3 & = \lambda x.\, \lambda y.\, y \\
& g & = \underline{\mathsf{if}}\ P\ \underline{\mathsf{then}}\ f_1\ \underline{\mathsf{else}}\ f_2 \\
& h & = \underline{\mathsf{if}}\ Q\ \underline{\mathsf{then}}\ f_1\ \underline{\mathsf{else}}\ f_3 \\
\underline{\mathsf{in}} & g\ x\ h &
\end{array}
\qquad
\begin{array}{lrl}
\underline{\mathsf{let}} & f_1 & = \lambda x.\, \lambda y.\, x \\
& f_2 & = \lambda x.\, \lambda y.\, x + x \\
& f_3 & = \mathsf{unit} \\
& g & = \underline{\mathsf{if}}\ P\ \underline{\mathsf{then}}\ f_1\ \underline{\mathsf{else}}\ f_2 \\
& h & = \mathsf{unit} \\
\underline{\mathsf{in}} & g\ x\ \mathsf{unit} &
\end{array}$$

The use of f_1 and f_3 as opposite branches of the if expression comprising the definition of h seems to indicate that, for the purposes of static analysis, one must assume that both f_1 and f_3 depend computationally on their second argument (since f_3 does, so must f_1, because we cannot know which one will be the value of h). However, as the body of h is dead code, we are free from any considerations arising from the analysis of this expression.

In 2001, Adam Fischbach and John Hannan developed an alternative approach to type-based UVE that comes close to our own approach for erasure semantics [34]. As we do, they divide function types into two categories, based on whether the function parameter is *needed* for the computation of the function's result.

- $A \xrightarrow{\mathsf{u}} B$ — the function parameter *is not* needed (unneeded)
- $A \xrightarrow{\mathsf{n}} B$ — the function parameter *may be* needed

They study UVE for languages with "necessity" annotations n and u that decorate function arrows (as above) and application nodes. Function abstractions $\lambda x.\, M$ are not annotated since their "mode" is determined in a completely local fashion based on whether $x \in FV(M)$. This presents no problems since their UVE program transformation does not erase any λ-binders.

The type system for their language includes a subtyping mechanism. The subtyping relation is generated by the axiom that $A \xrightarrow{\mathsf{u}} B$ is a subtype of $A \xrightarrow{\mathsf{n}} B$ — any function that definitely does not need its parameter is also a function that *may* (but just happens not to) need its parameter. Therefore the annotation n indicates a lack of precise knowledge and the subtype relation orders types by precision. More precise types are subtypes of less precise types.

Fischbach and Hannan's UVE does not erase λ-binders or unneeded function arguments, but rather it simply replaces unneeded function arguments with free dummy variables. A non-standard evaluation relation then discards unneeded arguments and λ-binders on the fly as they come into contact with each other. The semantics they give their language includes[1] the non-standard reduction rule

$$(\lambda x.\, M)@^{\mathsf{u}} N \rightarrow_\beta M$$

for application of a function to an unneeded argument.

Fischbach and Hannan also study a form of annotation polymorphism. Just as System F allows one to form expressions that are polymorphic in a particular type by explicit parameterization over that type and then instantiate polymorphic values to a particular type, they study a language extension whereby expressions may

[1]Actually, they give a big-step operational semantics to their language, but the rule we give here accomplishes the same thing for a small-step operational semantics.

be parameterized by and instantiated to "neededness" annotations. This feature enables a sort of dynamically determined erasure where evaluation of one instantiation of an annotation-polymorphic function may enjoy more erasure than another at run-time. As far as expressiveness is concerned, annotation polymorphism allows the language to express the results of a polyvariant UVA.

Comparison

Our approach to erasure semantics for dependently type languages has much in common with UVE. Erasure annotations on context entries in EPTS correspond to the usefulness designation of a variable: r-marked variables are (conservatively) considered useful while c-marked variables are considered useless.

The UVE process may be broken into two phases, analysis and program transformation. For the remainder of this section we reserve the term UVE for the program transformation phase and refer to the analysis phase as UVA (useless variable analysis).

For UVA, both Shivers [85] and Wand and Siveroni [94] make use of the n-CFA family of control flow analyses introduced by Shivers in his dissertation [84]. These analyses try to determine which λ-abstractions occurring in the source program evaluate to function values that may end up being applied at particular application sites during the course of program evaluation. The underlying reasoning is set-theoretic, as it involves sets of λ-abstractions.

In contrast, Kobayashi [44] and Fischbach and Hannan [34] consider typed languages and base UVA on the flow analysis implicit in the type system. We also take this approach. Kobayashi's UVA does demand-driven type inference where expressions of type $\top$ are not type-checked because they will be replaced with unit during UVE. In this way, his UVA algorithm identifies useless expressions rather than useless variables. It is only a later pass (after UVE) that removes certain λ-binders of $\top$ type.

In a sense, Fischbach and Hannan do not do UVA at all, because the onus is on the programmer to provide "neededness" annotations. Their type system merely checks that those annotations are consistent with each other. In contrast, our UVA algorithm (developed in Chapter 4) infers optimal erasure annotations for an unannotated PTS program. Our approach does not, however, extend completely to EPTS$^\bullet$, the extension of EPTS with proof irrelevance (discussed in Chapter 6).

Of all the UVA algorithms reported here, we find our own to be the simplest to understand, because we use no ad hoc constructions and merely state it as an optimizing SAT problem. The SAT algorithm we use is built from easy to understand, off-the-shelf algorithms and data structures developed for SAT solvers. However, the relative strength of our UVA as compared to the others is not obvious. On the one hand, our analysis works for dependently typed languages whereas the other UVAa for typed languages deal with weaker type systems. On the other hand, some of those UVAs enjoy the previously mentioned property that dead code identified by UVA can not weaken the precision of the analysis of other code.

However, this deficiency may be remedied by introducing η-expansions in the source program. Such a transformation is well-known to improve the precision of various automatic program analyses, such as binding time analysis. First, let us make the type application and abstraction of the previous example explicit.

$$
\begin{array}{lrcl}
\underline{\mathsf{let}} & f_1 & = & \lambda a{:}*.\, \lambda x{:}\mathsf{nat}.\, \lambda y{:}a.\, x \\
 & f_2 & = & \lambda a{:}*.\, \lambda x{:}\mathsf{nat}.\, \lambda y{:}a.\, x + x \\
 & f_3 & = & \lambda a{:}*.\, \lambda x{:}a.\, \lambda y{:}\mathsf{nat}.\, y \\
 & g & = & \underline{\mathsf{if}}\ P\ \underline{\mathsf{then}}\ f_1\ \underline{\mathsf{else}}\ f_2 \\
 & h & = & \underline{\mathsf{if}}\ Q\ \underline{\mathsf{then}}\ f_1\ \mathsf{nat}\ \underline{\mathsf{else}}\ f_3\ \mathsf{nat} \\
\underline{\mathsf{in}} & g\ (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat})\ x\ h &&
\end{array}
$$

Our UVA will not obtain the desired level of erasure for this program as listed. But if we η-expand the occurrences of f_1 and f_3 in the body of h, then our UVA

yields the following annotation:

$$\begin{array}{rcl}
\underline{\mathsf{let}} \quad f_1 & = & \lambda\!\!\lambda a{:}*.\, \lambda x{:}\mathsf{nat}.\, \lambda\!\!\lambda y{:}a.\, x \\
f_2 & = & \lambda\!\!\lambda a{:}*.\, \lambda x{:}\mathsf{nat}.\, \lambda\!\!\lambda y{:}a.\, x + x \\
@f_3 & = & \lambda\!\!\lambda a{:}*.\, \lambda\!\!\lambda x{:}a.\, \lambda y{:}\mathsf{nat}.\, y \\
g & = & \underline{\mathsf{if}}\ P\ \underline{\mathsf{then}}\ f_1\ \underline{\mathsf{else}}\ f_2 \\
@h & = & \underline{\mathsf{if}}\ Q\ \underline{\mathsf{then}}\ \lambda x{:}\mathsf{nat}.\, \lambda y{:}\mathsf{nat}.\, f_1\ \mathsf{nat}\ x\ @y \\
 & & \quad \underline{\mathsf{else}}\ \ \lambda x{:}\mathsf{nat}.\, \lambda y{:}\mathsf{nat}.\, f_3\ \mathsf{nat}\ @x\ y \\
\underline{\mathsf{in}} \quad g\ (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat})\ x\ h &&
\end{array}$$

We are treating let as syntactic sugar for a β-redex, which may be annotated throughout either with r or c. The f_3 and h bindings are annotated with c by our UVA, as indicated by the extra @ syntax on their let-bindings. This program now erases to

$$\begin{array}{rcl}
\underline{\mathsf{let}} \quad f_1 & = & \lambda x.\, x \\
f_2 & = & \lambda x.\, x + x \\
g & = & \underline{\mathsf{if}}\ P\ \underline{\mathsf{then}}\ f_1\ \underline{\mathsf{else}}\ f_2 \\
\underline{\mathsf{in}} \quad g\ x\ h & &
\end{array} ,$$

just as in the UVE algorithms of Shivers, Wand and Siveroni, and Kobayashi.

Fischbach and Hannan start out in much the same way that we do, distinguishing between definitely non-computational functions and possibly computational ones by having two categories of function types. However, our approaches differ with regards to subtyping and erasure.

Fischbach and Hannan assign all dummy λ-abstraction a u-annotated type, regardless of whether or not all the application sites at which they may be applied only ever apply other dummy λ-abstractions (Recall the discussion on erasability of λ-binders in Section 1.4.2). This choice does not cause any typing difficulties, because their subtyping mechanism always allows for implicit type coercions that serve to reduce the precision of the local analysis whenever a type mismatch may be avoided by doing so.

However, the use of subtyping with implicit coercion is fundamentally incompatible with the erasure of dummy λ-binders and unneeded arguments, because implicit coercion of a post-erasure value of type $A \xrightarrow{\mathsf{u}} B$ to a value of type $A \xrightarrow{\mathsf{u}} B$ would require the introduction of a dummy λ-binder. In EPTS or EPTS$^\bullet$, such coercions may be introduced explicitly. If M has type $\Pi^{\mathsf{c}} x{:}A.\, B$ then $\lambda^{\mathsf{r}} x{:}A.\, M@^{\mathsf{c}} x$ has type $\Pi^{\mathsf{r}} x{:}A.\, B$. Note that the post-erasure effect of such a coercion is that $M^\bullet$ is transformed into $\lambda x.\, M^\bullet$ where λx is a dummy binder. One may construct coercions that operate deep within a function type by nesting η-expansions that alternate from r to c mode whenever necessary, as in the above example. For example, if $F[\ ]$ coerces from A' to A and $G[\ ]$ coerces from B to B', then $H[\ ] = \lambda^{\tau'} x{:}A'.\, G[[\ ]@^{\tau} F[x]]$ coerces from $\Pi^{\tau} x{:}A.\, B$ to $\Pi^{\tau'} x{:}A'.\, B'$ (assuming that $\tau = \mathsf{c}$ only if $\tau' = \mathsf{c}$, as is required by the subtyping relation).

The relative lack of erasure of Fischbach and Hannan's UVE means that erasure must be delayed until run-time, resulting in an ad hoc semantics as well as additional run-time overhead for annotation inspection. In short, they give up some efficiency and simplicity of the run-time execution mechanism in exchange for increased flexibility in the type system so that programmers need not write their own coercions. However, this trade-off does not seem to be advantageous, since one may add coercive subtyping support to the language, if required, in the manner described above with automatically constructed coercions. This solution would retain the advantages of both compile-time erasure and implicit subtyping.

Annotation polymorphism as studied by Fischbach and Hannan is also at odds with our desire that the erasure take place at compile-time. Unlike implicit subtyping, however, this feature undoubtedly increases the expressiveness of the language. It is not clear, however, whether the additional run-time overhead required to create closures for annotation abstractions and pass around annotations at run-time is worth the savings one would obtain from additional dynamically determined erasure.

7.2 SUBSET AND SQUASH TYPES

As mentioned in Chapter 5, subset types and squash types have been introduced in the context of Martin-Löf type theory in order to cope with computationally irrelevant portions of dependently typed programs. Both concepts originated with Nuprl, an implementation of the extensional version of Martin-Löf type theory [22, Sections 2.4 and 10.3] developed by the research group of Robert Constable at Cornell in the 1980's. The implementation contained subset types as a primitive and squash types defined in terms of subset types.

To review, a term M is an inhabitant of the subset type $\{x : A \mid B\}$ if $M : A$ and $B[M/x]$ is inhabited. In other words, $\{x : A \mid B\}$ is the subtype of A containing exactly those inhabitants x that satisfy the proposition B.

These ideas never caught on in implementations of intensional type theories, in part due to the observations of Salvesen and Smith that the information that an inhabitant of $\{x : A \mid B(x)\}$ satisfies B cannot be used in non-trivial ways [79, 78]. In particular, they proved that the type $(y : \{x : A \mid B\}) \to B(y)$ is inhabited in intensional type theory only if $(y : A) \to C(y)$ is also inhabited. This fact corresponds to our inability to define a second projection function for subset types in Section 5.3.3.

In 1990, Bengt Nordström, Kent Petersson, and Jan M. Smith further developed the notion of subset types in intensional type theory in their book "Programming in Martin-Löf's Type Theory" [70, Part II]. Their *subset theory* is a complete revision of Martin-Löf type theory around the idea of subset types. This theory includes two additional judgments. In addition to the judgment that A is a type, there is a new judgment that A is a proposition. In addition to the judgment $M : A$ that says M is an inhabitant of the type A, there is a new judgment A true that says A is a true proposition. This language solves the aforementioned problem with subset types because now one can conclude that $B(y)$ true from the

assumption $y : \{x : A \mid B\}$.

The squash type $\bigcirc A$ may be thought of as an internalization of the judgment A true of the subset theory. The definability of the function *evidence* from Section 5.4.3 in terms of the squash type supports this assertion. Chapter 5 shows how subset types may be useful in intensional type theory without requiring the semantic interpretive overhead of the subset theory. Thus Caldwell's suggestion that intensional type theory is not suitable for reasoning with subset types [15, Section 3.3.1] is unwarranted.

In 1992, Thompson [90] also argues that the complexity of the subset theory is too high a price to pay for subset types. He claims that subsets are not in fact necessary because programs can always be reorganized in such a way as to isolate the core algorithms from the proofs of correctness, and that choosing a lazy evaluation strategy for our language ensures that computationally irrelevant portions of a program will never be evaluated. However, it is widely held that lazy evaluation imposes significant overhead on the efficiency of most programs and that it also makes it very difficult to reason about the space behavior of programs (how much memory they will consume and when they will release it for garbage collection).

I believe that subset and squash types never caught on in implementations of intensional type theory because, until now, no one knew how to support them without radically restructuring the entire language. Our results show how one may accomplish this goal.

7.3 PROGRAM EXTRACTION

It is well-known that proofs have computational content, but sometimes the computational content is obscured by non-computational content. The goal of *program extraction* is to identify a program embodying the computational content of a given proof. This is an old problem that many researchers have tackled. We divide the

bodies of work into three categories: realizability interpretations, the theory of specifications, and pruning methods.

7.3.1 Realizability Interpretations

The earliest research on this topic was done even before the invention of the digital computer when mathematicians were investigating the constructive nature of intuitionistic logic. In this context, Kleene introduced the notion of *realizability* in 1945 [43]. He defines a relation between numbers n and intuitionistic formulas ϕ that says, roughly speaking, that n encodes just enough information to allow one to reconstruct an intuitionistic proof of ϕ from n (assuming one knows the coding scheme and the formula ϕ). In this case we say that n *realizes* ϕ or n is a *realizer* of ϕ. The information encoded in n is a simple value where a value is either (1) a pair of values, (2) a computable function from values to values, or (3) a natural number[2].

In 1959, Kreisel introduced *modified realizability* [45, Paragraph 3.52], which differs from Kleene's realizability in that a realizer is no longer merely a number but a simply typed entity with a type τ formed according to the following grammar:

$$\tau ::= \mathsf{nat} \mid \tau \to \tau \mid \tau \times \tau$$

The type of a realizer is determined by the structure of the formula that it realizes. In this way, the notion of modified realizability of a formula ϕ involves a type τ of *potential* realizers as well as a predicate over τ satisfied by the *actual realizers* of ϕ, namely the relation x *realizes* ϕ [87].

In 1989, Christine Paulin-Mohring developed a program extraction algorithm for the Calculus of Constructions (CC) based on modified realizability [71]. This algorithm eventually became the basis for the program extraction facility of Coq.

[2]The language under Kleene's study was intuitionistic number theory, so the witness of an existential formula was a number.

In this case the realizers are System F^{ω} terms, which are also CC terms because System F^{ω} is a sub-language of CC. Since types and terms in CC live in the same syntactic category, a single extraction function $\mathcal{E}$ serves both to define the System F^{ω} type of potential realizers of a formula A in CC as well as to extract the actual realizer from a proof of A.

Paulin-Mohring also partitions the type structure of CC into so-called *informative* and *non-informative* fragments by splitting the sort $*$ of CC into *Set* (informative) and *Prop* (non-informative), as described in Section 5.5. The function $\mathcal{E}$ is similar to our erasure translation in that it erases λ-binders and function arguments when the range of the corresponding function type belongs to the non-informative fragment of the language. However, $\mathcal{E}$ also erases Π-binders in this case, which we do not. The erasure of Π-binders accounts for the fact that the language of realizers (System F^{ω}) has a strictly simpler type structure than the source language (CC).

In 2005, Ulrich Berger developed a realizability interpretation for Heyting arithmetic, one of the oldest formal languages based on intuitionistic logic [10]. The novelty of his approach was the use of so-called *uniform* or *non-computational* quantifiers $\{\forall\}$ and $\{\exists\}$[3]. The introduction rule for $\{\forall\}$ is stricter than that for $\forall$ in that it additionally requires that the parameter x in the introduction form for $\{\forall\}$ may not appear as a *computationally relevant variable* in its scope. Conversely, the elimination rule for $\{\exists\}$ is stricter than that for $\exists$. The reader may have guessed (correctly) that $\{\forall\}$ corresponds to our Π^{c} and $\{\exists\}$ to our exists, although in Heyting arithmetic the range of quantification is limited to simply typed values while we may quantify over much more complicated types in EPTS.

[3]Actually Berger introduced $\{\forall\}$ in 1993, but it was not explained in as much detail then [9].

Comparison

With the exception of the uniform quantifiers, all of these approaches erase λ-binders and their corresponding arguments only when the domain A of the corresponding function space is considered to be non-computational as a type. In other words, the notion of computational irrelevance is intrinsic. The thesis of this dissertation is that an extrinsic view of computational irrelevance is more flexible to use for programming. Section 1.4.1 discusses a problems with the intrinsic approach that is overcome by the extrinsic approach.

One advantage of the realizability approach over our own is that extracted programs have much simpler types than the original proof development. The way this happens is that the type of realizers of $\Pi x{:}A.\, B$ is taken to be simply the type of realizers of B whenever A is a non-informative type. However, the type we assign in this circumstance, namely $\forall x{:}A^\bullet.\, B^\bullet$, contains more information than $B^\bullet$ while being represented in the same way!

In fact, it seems our erasure translation can itself be viewed as a realizability interpretation similar to that of Paulin-Mohring. First we define program extraction simply as our erasure translation

$$\mathcal{E}[M] = M^\bullet \qquad\qquad \mathcal{E}[\Gamma] = \Gamma^\bullet$$

Then we define the realization relation: $\mathcal{R}[A](M)$ states that the IPTS term M realizes the EPTS (or EPTS$^\bullet$) type A. This function is also defined on typing contexts.

$$\mathcal{R}[s] = \lambda a.\, a \to s \qquad \begin{aligned} \mathcal{R}[\Pi^{\mathsf{r}} x{:}A.\, B] &= \lambda f.\, \Pi x{:}\mathcal{E}[A].\, \Pi \hat{x}{:}\mathcal{R}[A](x).\, \mathcal{R}[B](f\ x) \\ \mathcal{R}[\Pi^{\mathsf{c}} x{:}A.\, B] &= \lambda f.\, \Pi x{:}\mathcal{E}[A].\, \Pi \hat{x}{:}\mathcal{R}[A](x).\, \mathcal{R}[B](f) \end{aligned}$$

$$\mathcal{R}[\lambda^\tau x{:}A.\, M] = \lambda x.\, \lambda \hat{x}.\, \mathcal{R}[M] \qquad \mathcal{R}[M @^\tau N] = \mathcal{R}[M]\ \mathcal{E}[N]\ \mathcal{R}[N] \qquad \mathcal{R}[x] = \hat{x}$$

$$\mathcal{R}[\varepsilon] = \varepsilon \qquad\qquad \mathcal{R}[\Gamma, x{:}^\tau A] = \mathcal{R}[\Gamma], x{:}\mathcal{E}[A], \hat{x}{:}\mathcal{R}[A](x)$$

The meta-theoretical results we now need to prove are the following.

Proposition 7.3.1 (Correctness of Realizability Interpretation)

$$\frac{\Gamma \vdash M :^{\tau} A}{\mathcal{E}[\Gamma] \vdash \mathcal{E}[M] : \mathcal{E}[A]} \qquad \frac{\Gamma \vdash M :^{\tau} A}{\mathcal{R}[\Gamma] \vdash \mathcal{R}[M] : \mathcal{R}[A](\mathcal{E}[M])}$$

The first states that program extraction respects the type systems involved. This has already been proved for EPTS (and EPTS$^{\bullet}$). The second proposition states that any program extracted from a well-typed term M realizes the type A of that term. It seems that this result should be straightforward to prove by induction except perhaps in the Weak and Conv cases.

As Paulin-Mohring points out, a realizability interpretation provides one with a means for demonstrating the consistency of axioms introduced in the source language. Say we want to add the type A as an axiom to EPTS. The axiom is consistent with the rest of the theory if, and only if, $x : A \vdash \bot$ is not derivable. If A is realizable, then $x : A \vdash m : \bot$ is not derivable for any M, since the realizability interpretation would send such a derivation to a derivation of $x : \mathcal{E}[A], \hat{x} : \mathcal{R}[A](x) \vdash \mathcal{R}[\bot](\mathcal{E}[M])$ which implies that $\bot$ is provable because A is realizable and $\mathcal{R}[\bot](N)$ implies $\bot$. Therefore A's realizability implies its consistency as an axiom. This argument is easily extended to the case of several axioms: if $A_1 \ldots A_n$ are realizable, then consistency of the source language is preserved by adding them axioms.

For example, if the realizability interpretation extends to encompass inductive datatypes, then we may use it to prove consistency of the axiom $join : (a : *) \Rightarrow \bigcirc\bigcirc a \rightarrow \bigcirc a$ discussed previously. The realization predicate for $\bigcirc$ (i.e., squash) is defined as follows:

$$\underline{\mathsf{data}}\ \mathsf{rsquash}\ (a : *)\ (r : a \rightarrow *) : \mathsf{squash}\ a \rightarrow *\ \underline{\mathsf{where}}$$
$$\mathsf{rpoof} : (x : a) \rightarrow r\ x \rightarrow \mathsf{rsquash}\ a\ r\ \mathsf{poof}$$

To see why this is so, recall the definition of squash

$$\underline{\mathsf{data}}\ \mathsf{squash}\ (a : *) : *\ \underline{\mathsf{where}}$$
$$\mathsf{poof} : (x : a) \rightarrow \mathsf{squash}\ a$$

and apply the realizability interpretation defined above.

The realizer for *join* must be some term M of type

$$\mathcal{E}[(a : *) \Rightarrow \bigcirc\bigcirc a \to \bigcirc a] \quad = \quad (a : *) \Rightarrow \bigcirc\bigcirc a \to \bigcirc a$$

that satisfies the realizability predicate

$$\begin{aligned}
&\mathcal{R}[(a : *) \Rightarrow \bigcirc\bigcirc a \to \bigcirc a] \\
&\quad = \lambda f.\, (a : *) \to (r : a \to *) \to \\
&\qquad\qquad (x : \bigcirc\bigcirc a) \to (e : \mathsf{rsquash}\ (\bigcirc a)\ (\mathsf{rsquash}\ a\ r)\ x) \to \mathsf{rsquash}\ a\ r\ (f\ x)
\end{aligned}$$

Note: we have α-renamed $\hat{a}$ to r and $\hat{x}$ to e for improved clarity. One such realizer is simply $M = \lambda\_.\, \mathsf{poof}$.

An interesting consequence of the definition of rsquash is that

Proposition 7.3.2 *$\bigcirc A$ is realizable iff A is realizable.*

An immediate corollary is

Proposition 7.3.3 *If $\bigcirc A$ is inhabited then A is realizable.*

This justifies somewhat our use of non-computational axioms. However, the axiom $\bigcirc((a : *) \Rightarrow \bigcirc a \to a)$ proposed in Section 6.2 does not appear to be realizable.

7.3.2 The Theory of Specifications

In the years 2001-2003, Paula Severi, Nora Szasz, Femke van Raamsdonk, Maribel Fernández, and Ian Mackie developed an extension of type theory called the *theory of specifications* for the purpose of program extraction [81, 92, 33, 32]. The best way to describe this language is by way of analogy. Just as calculi of explicit substitutions differ from more standard λ-calculi by internalizing the meta-level operation of substitution, the theory of specifications differs from more standard type theories by internalizing the meta-level operation of realizability interpretation.

The technical means by which realizability is internalized is called *ultra* Σ-*types*. An ultra Σ-type $\Sigma x{:}A.B$ is a *specification* consisting of a type A of x, the entity specified, together with some property B that x must satisfy. Inhabitants of this type are pairs of a realizer x together with evidence that x realizes the specification. The reduction rules for inhabitants of ultra Σ-types incrementally accomplish the realizability interpretation (just as additional reduction rules in calculi of explicit substitutions incrementally accomplish substitution). The rules for well-formedness of specification expressions prevent the realizer component x (of type A) from depending on the evidence of realization (of type $B(x)$).

Comparison

For our purposes, the theory of specifications offers no advantages over a standard realizability interpretation. A program whose execution involves dynamic program extraction is certainly less efficient, both in space and time, than one that has already been extracted statically.

However, the incremental presentation of a realizability interpretation embodied in the theory of specifications does help one understand them better. Ultra Σ-types serve to highlight the relationship between modified realizability and subset types. Both involve a type of potential realizers/programs satisfying some desired correctness predicate over that type. In both cases, the inhabitant of the former may not depend computationally on the evidence for the latter.

This explanation of realizability in terms of subset types highlights the fact that the semantics given by Nordström, Petersson, and Smith [70, Part II] to their subset theory is essentially a modified realizability interpretation. They interpret each type in the subset theory as pair of a type and a predicate over that type in the underlying standard type theory without subset types.

7.3.3 Pruning Methods

Another area of related work is *pruning methods* for program extraction. A pruning algorithm replaces some of the subterms of a program with dummy terms. The goal is to prune away computationally irrelevant portions of the program in order to improve program efficiency.

In 1994, Stefano Berardi introduced the idea of pruning in the context of the simply typed λ-calculus [7]. His algorithm prunes by replacing subterms with the constant unit of type $\top$. He proved that a pruning that leaves the overall type of a program undisturbed yields a program operationally equivalent to the original. Later that same year, Luca Boerio extended these results from simply typed λ-calculus to System F [12]. The work of Kobayashi cited in Section 7.1 is essentially a re-implementation of these results with a supposedly more efficient algorithm.

In 1996, Mario Coppo, Ferruccio Damiani, and Paola Giannini further developed the work of Berardi and Boerio by annotating types as either computational or not (rather than using the catch-all type $\top$ for all non-computational entities) and introducing a notion of subtyping whereby the non-computational types are subsumed by their computational counterparts [23]. Note that their notion of computational relevance is intrinsic rather than extrinsic — a type A is annotated as A^{ω} if its inhabitants may not be used in the computation and as A^{δ} otherwise. By their subtyping relation, A^{δ} is a subtype of A^{ω}, indicating that an entity may waive its right to be used in a computationally relevant context at any time.

The use of subtyping goes some way towards silencing our objections to the intrinsic approach to computational irrelevance. Just as we may always apply (instantiate) a function of type $\Pi^{\mathsf{c}}x{:}A.\,B$ to a value $y{:}^{\mathsf{r}}A$ in our approach, the type $A^{\omega} \to B^{\delta}$ may be applied to $y : A^{\delta}$ by subtyping. In each case, the result is that computational relevance depends on context. What is relevant in one context may be irrelevant in another.

The pruning algorithms discussed so far in this section are only the first half of

program extraction. To be practical, pruning must be followed by a function and tuple simplification phase wherein λ-binders and function arguments corresponding to non-computational function parameters are erased and non-computational components of tuples are erased. In 2000, several of the same authors — Stefano Berardi, Mario Coppo, Ferruccio Damiani, Paola Giannini — showed how to fuse these two phases into one [8]. The resulting program transformation is similar to our erasure translation.

In 2002, Pierre Letouzey overhauled the program extraction mechanism of Coq [51]. The approach of Paulin-Mohring based on a realizability interpretation was abandoned since it did not handle the full Coq language. Letouzey's extraction instead uses a pruning algorithm that simply replaces certain subterms with a dummy token $\Box$. A second, post-pruning pass elides all token type eliminations, replaces empty type eliminations with code that raises an exception, does an optimization akin to the Haskell newtype optimization[4], and removes superfluous lambda-binders and applications, leaving a protecting dummy binder whenever necessary.

Comparison

Of all the work cited here, Letouzey's program extraction is the closest to our work on erasure semantics, since Coq is a a dependently typed language. None of the other pruning algorithms deal with type systems more complex than System F.

The optimization of token type and empty type elimination is reminiscent of the language features of token type target erasure and empty type target erasure discussed in Chapter 5. Figure 7.1 shows our *loopy* example of Figure 5.1 ported

[4]In Haskell, the declaration newtype $\mathsf{T}\ a\ b\ c\ =\ \mathsf{C}\ A$ introduces a datatype with a single constructor C with a single argument of type A. Haskell guarantees that the run-time representation of T-values is simply the representation of A with no extra information corresponding to the constructor C. In general, this optimization is possible whenever the lone constructor C has multiple arguments, so long as exactly one of them survives program extraction. Two examples of such types are weak sums and subset types as defined in Chapter 5.

A Coq development.

```
Inductive TyEq (A : Set) : Set -> Prop := TyRefl : TyEq A A.
(* coerce = \x. x *)
Definition coerce (A B : Set) (p : TyEq A B) (x : A) :=
  TyEq_rec A (fun C : Set => C) x B p.
(* symm = TyRefl *)
Definition symm (A B : Set) (p : TyEq A B) :=
  TyEq_ind A (fun C : Set => TyEq C A) (TyRefl A) B p.
(* loopy = (\x. x x) (\x. x x) *)
Definition loopy (A B : Set) (p : TyEq A (A -> B)) :=
  let selfapp x := coerce A (A->B) p x x in
  selfapp (coerce (A -> B) A (symm A (A -> B) p) selfapp).
Extraction Language Scheme.  Recursive Extraction loopy.
```

The resulting extracted Scheme program.

```
(define coerce (lambda (x) x))
(define loopy (lambda (_)
  (let ((selfapply (lambda (x) ((coerce x) x))))
       (selfapply (coerce selfapply)) ) ))
```

Figure 7.1: A Coq term and its corresponding extracted looping program

to Coq. Note that Letouzey's extraction algorithm removes everything that token type target erasure would have, except that the resulting extracted program is protected from non-termination by wrapping it with a dummy λ-binder.

```
(define loopy (lambda (_) ...))
```

This means that program extraction preserves termination, and it is the responsibility of the programmer compiling against extracted code to only dispatch to that code when the appropriate preconditions are met, or else risk the possibility of a raised exception.

We see nothing preventing one from performing Letouzey's post-processing simplifications after our erasure translation. In fact, some such optimization may be required for practical applications since equality reasoning and manipulation of other token types is common in formal proofs.

As a final remark, we note one flaw common to both Paulin-Mohring's and Letouzey's approach to program extraction in Coq: they both consider the non-computational aspects of a proof to be the "logical" parts, namely the types and proofs. This assumption seems true since those are the most pervasive (in the case of types) and largest (in the case of proofs) examples of computationally irrelevant portions in actual developments. However, these two notions notions are actually orthogonal.

- *Types are not necessarily non-computational.* Letouzey cites a program of David Monniaux whose purpose is to compute types of lattices [51, 67]. In this case, the program definitely depends computationally on a type!

- *Proofs are not necessarily non-computational.* In a case study of Coq's program extraction facility, Luís Cruz-Filipe and Bas Spitters report that proofs are often computationally relevant [29] and this fact must be taken into account when developing proofs with an eye towards eventual program extraction.

- *Non-computational parts are not necessarily types or proofs.* Consider a mapping function over lists of a particular length

$$map : (a, b : *) \Rightarrow (n : \mathsf{nat}) \Rightarrow (a \to b) \to \mathsf{list}\ a\ n \to \mathsf{list}\ b\ n$$

  The execution of *map* only depends on the function and list arguments. In particular, the list length argument is computationally irrelevant, though it is neither a type nor a proof.

In light of these insights, our approach decouples the notion of computational irrelevance from proofs and types.

7.4 MISCELLANEOUS

We now discuss several related works that do not fit into any particular category.

The EPTS type system was heavily inspired by Frank Pfenning's treatment of proof irrelevance in the context of the Edinburgh Logical Framework [73] in 2001. Pfenning associates various logical modalities such as proof-irrelevance and intensionality with different "flavors" of function space. Our notion of computational irrelevance is closely related to his notion of proof irrelevance. In the same paper, Pfenning also informally considers the connection between his proof-irrelevant function space and the squash type of Nuprl. One may view our work as an extension of Pfenning's to calculi with more complicated type structure.

One significant difference between our work and Pfenning's is in the well-formedness rules for the type $\Pi^{\mathsf{c}} x{:}A.\, B$ — Pfenning checks B in a context where x is c-bound to the type A whereas in our work, x is considered r-bound in B. This aspect of our work follows from the decision to treat computational irrelevance extrinsically. In this case, the function parameter x may be computationally relevant to the type of a function's return value even if it is not computationally relevant to the return value itself.

Obviously, Alexandre Miquel's work on the Implicit Calculus of Constructions (ICC) is related to IPTS [64, 65]. As has already been mentioned in Chapter 3, ICC is basically Zhaohui Luo's Extended Calculus of Constructions [53] (ECC) extended with $\forall$, an implicit product type former denoting large type intersection. Roughly speaking, ICC is the particular IPTS with the underlying PTS specification of ECC. We say "roughly speaking" because there are still several important differences between ICC and that particular IPTS, as outlined in Chapter 3.

Independently from our work, Bruno Barras and Bruno Bernardo have recently been studying ICC*, an explicitly typed version of ICC, as a type theory with a built-in notion of program extraction [5]. The extensions that they make to ICC to ensure decidable type checking bring the language very close to our own EPTS. One may view our work on EPTS and EPTS$^\bullet$ as showing how to efficiently implement type theories in the style of ICC*.

In 1989, Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman identified a class of extensions to type theory involving "types with information loss" [2, Section 3.4] wherein certain premises of the introduction rule for a particular type are omitted in the conclusion from the inhabitant of that type. They discuss several examples of this phenomenon, including subset types, union types, and intersection types. Their subset types are as we have described subset types here. Their union types are effectively the weak sums of Section 5.3.2 and their intersection types are effectively our non-computational dependent product (i.e., function) types. The interpretation of weak existentials and non-computational function spaces as unions and intersections is quite an interesting semantic viewpoint. In fact, Miquel interprets $\forall$ types as intersections in his denotational semantics for IPTS [63].

Chapter 8

CONCLUSION

In this section, we summarize our thesis and supporting research, outlining its significance, limitations, and the new questions that it raises.

8.1 SUMMARY

The thesis of this dissertation is that an extrinsic view of computational relevance results in (1) a flexible erasure semantics for dependently typed languages; (2) a generic form of parametric polymorphism; and (3) an elective notion of proof irrelevance.

In Chapter 1, we discussed the main problem with existing approaches to combining dependent types and erasure semantics — viewing computational relevance as an intrinsic property of a term determined by its type forces programmers to duplicate certain type and function definitions so that they may be used in both computationally relevant and irrelevant settings. We argued that the intrinsic view of computational irrelevance is flawed: the notion of computational relevance of a particular term depends on the overall term whose value we are trying to compute. A subterm may be computationally relevant with respect to some enclosing term, but irrelevant with respect to another, larger, enclosing term.

Given the considerations of Chapter 1, we developed a core erasure semantics for PTS in Chapter 3. This erasure semantics depends on an intermediate language EPTS with erasure annotations indicating which variables and expressions are computationally irrelevant with respect to particular contexts of use (represented in the

language by functions). The chief technical device used is that all function types $\Pi^{\tau} x{:}A.\, B$ are categorized as either computational ($\tau = \mathsf{r}$) or non-computational ($\tau = \mathsf{c}$) depending on whether the functions they classify use their parameter in the computation of their result. Such a classification is necessarily approximate. The type system errs on the side of assuming functions to be computational if they do not meet a simple syntactic criterion for being non-computational. We show how this simple criterion can be efficiently implemented using a clever representation of typing contexts.

An erasure translation cuts out both the formal parameters (λ-binders) and actual parameters (function arguments) of non-computational functions. We prove that erasure respects both the operational semantics and type systems of the source and target languages. Effectively, $\rightarrow_{\beta}^{*}$ in the target language simulates $\rightarrow_{\beta}$ in the source language, and $\rightarrow_{\beta}^{+}$ in the source language simulates $\rightarrow_{\beta}$ in the target language. The proofs of these statements show that the erasure translation forms an effective erasure semantics because it (1) eliminates old work, (2) introduces no new work, and (3) preserves the meaning of programs (with respect to both static and dynamic semantics).

The erasure translation targets IPTS, a generalization of Miquel's Implicit Calculus of Constructions. IPTS supports explicit and implicit dependent products. The explicit product $\Pi x{:}A.\, B$ is the type of functions introduced by λ-abstraction and eliminated by function application. The implicit product $\forall x{:}A.\, B$ is the type of polymorphic values and there are no corresponding syntactic cues for introduction and elimination. Erasure sends Π^{r} to Π and Π^{c} to $\forall$, indicating that non-computational functions exhibit a highly general form of parametric polymorphism: polymorphism not only over types, but also over numbers, proofs, or any other entity whose type may appear as the domain of a well-formed Π^{c} type.

To use the results of Chapter 3 directly as the basis for an erasure semantics, one must work in a source language with erasure annotations. However, this is

not always possible or desirable. Chapter 4 presented an algorithm for automatic optimal annotation of a PTS term to obtain an EPTS term. The basic idea is to augment the EPTS type system so that it generates constraints in terms of those variables. Solutions to these constraints correspond to correct annotations of the original program and therefore to a type-preserving map from PTS to EPTS for the particular term with which we are working. We showed that the constraints generated by the augmented type system are SAT problems (under a suitable interpretation of erasure annotations as booleans). We prove that the resulting SAT problem ϕ has an solution σ that is optimal in the following sense: if a particular propositional variable α is set to true under any solution to ϕ, then it is set to true by σ. Since true corresponds to c, this means that the analysis algorithm marks for erasure as much of the original program as possible.

Chapter 5 explored the consequences of programming directly with erasure annotations in a dependently typed language with inductively defined datatypes. We first introduced such types as they appear in modern languages, and then we examined the reduction rules for inductive types to see what opportunities they afford for additional erasure in a natural extension to the erasure semantics of Chapter 3. We found that elimination of an inhabitant of an inductively defined type only depends computationally on the target of the elimination (i.e., the entity of the type to be eliminated) and on the methods for the elimination (i.e., the functions with which elimination replaces the various data constructors of that type). In two cases, erasure of the target argument of a datatype eliminator appears to be warranted, but upon further inspection these two cases lead either to additional unwanted normal forms or to the possibility of non-termination in post-erasure execution of programs.

The use of the non-computational function space in types assigned to data constructors affords the programmer with a crude mechanism for simplifying the run-time representation of an inductive type by causing certain constructor ar-

guments at certain positions to be erased prior to run-time. We show several paradigmatic examples of this mechanism, namely weak sum types, subset types, and squash types. We study several properties of squash types, including their the relationship with the types of the same name in Nuprl. Finally, we discuss how squash types provide an alternative way to partition the space of types into "informative" and "non-informative" fragments, as in Coq, while avoiding the ad hoc distinction between Prop and Set.

Chapter 6 explored the consequences of integrating the erasure semantics of EPTS into the notion of definitional equality used in the EPTS type system. We proved that the language EPTS$^\bullet$ resulting from this modification is basically an explicitly typed version of IPTS. This is important, because type checking IPTS is always undecidable whereas type-checking EPTS is decidable if the underlying IPTS is strongly normalizing. The meta-theory of EPTS carries over to EPTS$^\bullet$ essentially unchanged, so that all the same properties of the erasure semantics hold when EPTS$^\bullet$ is considered to be the source language rather than EPTS.

Adding inductive types to an EPTS$^\bullet$-style base language has several interesting consequences. Firstly, computationally irrelevant constructor arguments play no part in the compile-time conversion check (i.e., definitional equality). The effect is that certain types exhibit a form of proof irrelevance. For instance, two inhabitants of a particular subset type are considered definitionally equal so long as their first components (of the "superset" type) are definitionally equal, regardless of whether the second components providing evidence for the defining property of the subset are equal.

Another consequence is that token types such as squash types and equality types have the property that any two elements of that type are provably equivalent. In the case of equality types, this property is well-studied and has the important consequence that Streicher's "axiom K" in fact becomes provable. The importance of this result is that the familiar functional programming style of defining functions

by pattern matching equations is only justified in the presence of the K axiom.

In Chapter 7, we outlined the major bodies of work related to the research presented here, namely useless variable elimination, subset and squash types, and program extraction. The major program extraction techniques we discussed were realizability interpretations, the theory of specifications, and pruning methods.

Useless variable elimination (UVE) is a technique for simplifying functional programs that bind variables to values that have no impact on the ultimate value of the overall program. While human programmers would likely not ever write such code, machine generated code often exhibits this property. Early UVE algorithms leveraged general control flow analyses in identifying useless variables and code. Later approaches were integrated with the type system of the source language, and fell into the category of pruning techniques. UVE is analogous to program extraction, but studied in the functional programming community rather than the dependent type theory community.

Subset and squash types were studied in the context of Martin-Löf type theory as a way of delimiting the non-computational aspects of a proof development so that when proofs are considered as programs, they do not carry along computationally irrelevant baggage. As language features, subset and squash types only caught on in proof assistants based on *extensional* type theory, such as Nuprl. In intensional type theory, subsets are practically impossible to use unless one has a way to distinguish between computationally relevant and computationally irrelevant *conclusions* drawn from some assumption of a subset type. The subset theory of Nordström et al. introduces an additional judgment form of non-computational conclusions in order to overcome this limitation. The semantics of the subset theory is essentially a realizability interpretation of a type theory with subset types into one without them.

One way to understand the squash type is as an internalization of the non-computational judgment form of the subset theory. An EPTS or EPTS$^\bullet$-based

language allows one to work in an intensional type theory with subsets and squash types in a similar way as one would in the subset theory but without having to understand a complex realizability interpretation.

Most research in program extraction falls into three general categories, realizability interpretations, theory of specifications, and pruning methods. Realizability interpretations have the oldest history, originating in the work of Kleene in 1945. Most applications to proof extraction make use of Kreisel's modified realizability because it takes typed terms in a functional language as realizers. Paulin-Mohring extended the technique of modified realizability to the sophisticated logical system of the calculus of constructions. Berger showed how non-computational quantifiers can be used in conjunction with a realizability interpretation in order to improve the efficiency (reduce the size) of extracted programs. Our erasure translation also appears to be a realizability interpretation combining the best features of these two prior works: supporting a very expressive type theory as does Paulin-Mohring and non-computational quantifiers as does Berger.

The theory of specifications internalizes the notion of realizability interpretation into the language in much the same way as calculi of explicit substitutions internalize the notion of substitution. The internalization makes use of an extremely strong version of Σ types that represent program specifications. One defining characteristic of this work that, in our view, makes it unsuitable as the basis for program extraction is the fact that extraction happens dynamically every time a program is executed instead of statically once and for all.

Pruning methods are those whereby computationally irrelevant subterms of an input program are replaced by some dummy expressions that require no further evaluation. Combined with subtyping, this approach seems to yield good improvements in program space and execution speed. However, for languages with expressive type systems, the approach relies on a vaguely defined second pass that attempts to eliminate unnecessary run-time manipulation of dummy values. The

latest survey of work on pruning arrives at an erasure based approach quite similar to our own.

8.2 SIGNIFICANCE

The chief contribution of our research is a language design combining dependent types with an erasure semantics. This design advances the state of the art by avoiding the problem of forcing programmers to duplicate code in order to achieve the amount of erasure one desires. Our solution admits an efficient implementation, both in terms of automatically annotating unannotated programs and in terms of type-checking annotated programs.

Our investigations have uncovered a strong correspondence between our particular notion of computational irrelevance and the widely known and practically useful notion of parametric polymorphism. In light of this correspondence, we feel that functional programmers familiar with the statically typed languages like ML and Haskell should have little problem programming in a language with explicit erasure annotations.

Once inductive types are included in such a language, we see how to program from scratch certain language constructs that previously required direct language support such as weak sum types, subset types, and squash types. Accounting for these old constructs in a common framework yields a conceptual economy with practical benefits — there are now fewer primitives to understand and implement.

If we start with EPTS as the basis for a programing language with erasure annotations, it is possible to integrate our erasure semantics into the notion of definitional equality used to type our programs. Doing so yields a much more liberal definitional equality relation while still remaining intensional. The benefits of this extra freedom include a user-directed form of proof-irrelevance that can be used to justify the common functional programming style of programming functions by pattern matching equations. Equational reasoning about programs written in this

style is often more natural than equational reasoning about programs written with eliminators as in Chapter 5. Theoretically, the move from EPTS to EPTS$^\bullet$ further underscores the relationship between computational irrelevance and parametric polymorphism because EPTS$^\bullet$ is a closer relative to IPTS than is EPTS.

8.3 LIMITATIONS AND FUTURE WORK

We know of several limitations of our research and list them here. Each limitation is a starting point for further research.

We do not know if our erasability analysis scales from EPTS to EPTS$^\bullet$. At this point it seems as though it does not. If it does not, then one must forego either (1) the benefits of programming without any regard to erasure and still reaping the benefits of an erasure semantics, or (2) the extra liberality afforded by the modified conversion rule Conv$^\bullet$ that implies a certain measure of user-directed proof irrelevance and the derivability of the theoretically important "axiom K". We leave as future work the question of what is the best practical way to handle this tradeoff, or if it can be avoided altogether.

The treatment of the program analysis in terms of a non syntax-directed type system such as is standard for PTS means that different theoretical runs of the constraint generation "algorithm" may yield different outputs for the same inputs. Though we feel that the same approach we have taken could easily be replayed for a syntax directed type system, we have no formal proof of this statement.

A final known limitation of the work presented in this dissertation is the lack of known models for IPTS with inductively defined types. Currently known models for IPTS are based on coherence spaces which do not support indexed union as a type-forming operation in the same way that they support indexed intersection (the interpretation of the $\forall$ types in IPTS). This means that the extension of such models to handle inductive types is suspect, because, in particular, the weak sum

type should be interpreted as a union[1]. One hopes that either this limitation can be removed from models based on coherence spaces or that entirely new models can be constructed that support inductively defined types.

It is not known whether the post-pruning suite of program optimizations described by Letouzey could be applied after our own erasure translation with similar effectiveness. We do know that our erasure semantics scales from two levels (c and r) to three (c and r and d) where c conclusions may depend on any assumptions, r conclusions may depend only on r and d assumptions, and d conclusions may depend only on d assumptions. In this way, the old r phase splits into the new r and d phases. Entities marked c are erased as before prior to run-time (and perhaps even prior to conversion checks). The d mark is used exclusively for the target arguments of token types and empty types. Entities marked d cannot be erased in the way we have outlined in this dissertation, but perhaps they can be erased in the slightly less aggressive way that Letouzey describes in his post-processing phase. Experience with an implementation of this hybrid strategy is required to determine feasibility of the approach.

[1]Many thanks to Bruno Barras for explaining to us the limitations of models of Miquel's Implicit Calculus of Constructions.

REFERENCES

[1] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[2] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, 1989.

[3] Henk Barendregt. Introduction to generalised type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[4] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[5] Bruno Barras and Bruno Bernardo. The Implicit Calculus of Constructions as a programming language with dependent types. In *Proceedings of the Eleventh International Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379, 2004.

[6] Gilles Barthe and Thierry Coquand. An introduction to dependent type theory. In *Applied Semantics. Lecture Notes for the APPSEM Summer School*, volume 2395 of *Lecture Notes in Computer Science*, pages 1–41, 2002.

[7] Stefano Berardi. Pruning simply typed lambda terms. *Journal of Logic and Computation*, 6(5):663–681, 1996.

[8] Stefano Berardi, Mario Coppo, Ferruccio Damiani, and Paola Giannini. Type-based useless-code elimination for functional programs. In *Proceedings of the Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, pages 172–189, 2000.

[9] Ulrich Berger. Program extraction from normalization proofs. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 91–106, 1993.

[10] Ulrich Berger. Uniform Heyting Arithmetic. *Annals of Pure and Applied Logic*, 133:125–148, 2005.

[11] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475, 2006.

[12] Luca Boerio. Extending pruning techniques to polymorphic second order lambda-calculus. In *Programming Languages and Systems, Fifth European Symposium on Programming*, pages 120–134, 1994.

[13] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

[14] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129, 2004.

[15] James L. Caldwell. *Decidability Extracted: Synthesizing "Correct-by-Construction" Decision Procedures from Constructive Proofs*. Ph.D. thesis, Cornell University, 1998.

[16] Luca Cardelli. Phase distinctions in type theory. Available at `http://research.microsoft.com/Users/luca/Papers/PhaseDistinctions.pdf`, 1988.

[17] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77, 2005.

[18] James Cheney and Ralf Hinze. First class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[19] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics, 2nd Ser.*, 33(2):346–366, April 1932.

[20] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics, 2nd Ser.*, 34(4):839–864, October 1933.

[21] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940.

[22] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[23] Mario Coppo, Ferruccio Damiani, and Paola Giannini. Refinement types for program analysis. In *Third International Symposium on Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 143–158, 1996.

[24] The Coq proof assistant. `http://coq.inria.fr`.

[25] Thierry Coquand. *Une Théorie des Constructions*. Thèse de troisième cycle, Université Paris VII, 1985.

[26] Thierry Coquand. Pattern matching with dependent types. In *In Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, 1992.

[27] Thierry Coquand and Gérard Huet. A theory of constructions. Presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis, June 1984.

[28] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, Feb/Mar 1988.

[29] Luís Cruz-Filipe and Bas Spitters. Program extraction from large proof developments. In *16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 205–220, 2003.

[30] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[31] N. G. de Bruijn. A survey of the project AUTOMATH. In Seldin and Hindley [80], pages 579–606.

[32] Maribel Fernández, Ian Mackie, Paula Severi, and Nora Szasz. Reduction strategies for program extraction. *CLEI Electronic Journal*, 6(1), 2003.

[33] Maribel Fernández and Paula Severi. An operational approach to program extraction in the calculus of constructions. In *12th International Workshop on Logic Based Program Synthesis and Tranformation*, pages 111–125, 2002.

[34] Adam Fischbach and John Hannan. Type systems for useless-variable elimination. In *Proceedings of the Second Symposium on Programs as Data Objects*, pages 25–38, 2001.

[35] Gerhard Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934. Translated in [36] Szabo (ed.), *The Collected Papers of Gerhard Gentzen* as "Investigations into Logical Deduction".

[36] Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

[37] J.-Y. Girard. Une extension de l'interprétation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Second Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[38] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de doctorat d'etat, University of Paris VII, 1972.

[39] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[40] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proceedings of the Seventeenth ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 341–354, 1990.

[41] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-Five Years of Constructive Type Theory*, pages 83–111. Oxford University Press, 1998.

[42] W. A. Howard. The formulae-as-types notion of construction. In Seldin and Hindley [80], pages 479–490. A version of this paper was privately circulated in 1969.

[43] Stephen Cole Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Computation*, 10(4):109–124, 1945.

[44] Naoki Kobayashi. Type-based useless-variable elimination. *Higher-Order and Symbolic Computation*, 14(2-3):221–260, 2001.

[45] Georg Kreisel. Interpretation of analysis by means of functionals of finite type. In Arend Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, 1959.

[46] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[47] P. J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, 1965.

[48] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part II. *Communications of the ACM*, 8(3):158–167, 1965.

[49] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[50] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 42–54, 2006.

[51] Pierre Letouzey. A new extraction for Coq. In *Second International Workshop on Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219, 2003.

[52] Chunxiao Lin, Andrew McCreight, Zhong Shao, Yiyun Chen, and Yu Guo. Foundational typed assembly language with certified garbage collection. In

First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, pages 326–338. IEEE Computer Society Press, 2007.

[53] Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, 1994.

[54] David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 277–286, 1986.

[55] Per Martin-Löf. A theory of types. Unpublished manuscript, October 1971.

[56] Per Martin-Löf. An intuitionistic theory of types. Unpublished manuscript, 1972.

[57] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[58] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[59] Conor McBride. Elimination with a motive. In *Selected papers from the International Workshop on Types for Proofs and Programs (TYPES '00)*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216, 2002.

[60] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[61] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[62] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[63] Alexandre Miquel. A model for impredicative type systems, universes, intersection types and subtyping. In *15th Annual Symposium on Logic in Computer Science*, pages 18–29, 2000.

[64] Alexandre Miquel. The implicit calculus of constructions. In *Proceedings of 5th International Conference on Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359, 2001.

[65] Alexandre Miquel. *Le Calcul des Constructions Implicite: Syntaxe et Sémantique*. PhD thesis, Université Paris 7, 2001.

[66] Alberto Momigliano. *Elimination of Negation in a Logical Framework*. PhD thesis, School of Computer Science, 2000. available as Technical Report CMU-CS-00-175.

[67] David Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998.

[68] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM Press, 2001.

[69] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[70] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[71] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. In *Proceedings of the Sixteenth ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 89–104, 1989.

[72] Simon Peyton-Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, 2006.

[73] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pages 221–230. IEEE Computer Society Press, June 2001.

[74] *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, 1999.

[75] J. C. Reynolds. Introduction to polymorphic lambda-calculus. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 77–86. Addison-Wesley, 1990.

[76] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings, Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, 1974.

[77] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523. North-Holland, 1983.

[78] Anne Salvesen. On specifications, subset types and interpretation of proposition in type theory. *BIT Numerical Mathematics*, 32(1):84–101, 1992.

[79] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 384–391, 1988.

[80] J. P. Seldin and J. R. Hindley, editors. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[81] Paula Severi and Nora Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27(1):61–87, 2001.

[82] Tim Sheard. Languages of the future. In *Proceedings of the Nineteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 116–119, 2004. OOPSLA Companion Volume.

[83] Tim Sheard and Emir Pašalić. Meta-programming with built-in type equality. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages*, pages 106–124, 2004.

[84] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-145, School of Computer Science.

[85] Olin Shivers. Useless-variable elimination. In *Proceedings of the Workshop on Static Analysis of Equational, Functional and Logic Programs*, pages 197–201, 1991.

[86] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

[87] Thomas Streicher. Intensional type theory, modified realizability. TYPES mailing list, May 1992. Archived at `http://www.cis.upenn.edu/~bcpierce/types/archives/1992/msg00076.html`.

[88] Thomas Streicher. *Investigations into Intensional Type Theory*. Habilitationsschrift, LMU Munich, 1993.

[89] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[90] Simon Thompson. Are subsets necessary in Martin-Löf type theory? In *Constructivity in Computer Science, Summer Symposium*, volume 613 of *Lecture Notes in Computer Science*, pages 46–57, 1992.

[91] L. S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.

[92] Femke van Raamsdonk and Paula Severi. Eliminating proofs from programs. *Electronic Notes in Theoretical Computer Science*, 70(2), 2002.

[93] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989.

[94] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In POPL:99 [74], pages 291–302.

[95] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

[96] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, 2003.

[97] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In POPL:99 [74], pages 214–227.

Appendix A

PROOFS

A.1 META-THEORY OF EPTS

Lemma A.1.1 (Idempotence of Context Reset)

$$\Gamma^{\circ\circ} = \Gamma^{\circ}$$

Proof. By induction on the structure of Γ.

Case	**Step**			**Justification**
ε	$\varepsilon^{\circ\circ}$	$=$	ε°	def. of $\circ$
$\Gamma, x{:}^{\tau}A$	$(\Gamma, x{:}^{\tau}A)^{\circ\circ}$	$=$	$(\Gamma^{\circ}, x{:}^{\mathsf{r}}A)^{\circ}$	def. of $\circ$
		$=$	$\Gamma^{\circ\circ}, x{:}^{\mathsf{r}}A$	def. of $\circ$
		$=$	$\Gamma^{\circ}, x{:}^{\mathsf{r}}A$	ind. hyp. on Γ
		$=$	$(\Gamma, x{:}^{\tau}A)^{\circ}$	def. of $\circ$

□

Lemma A.1.2 (Context Phase Weakening)

$$\frac{\Gamma, \Delta \vdash M :^{\tau} A}{\Gamma^{\circ}, \Delta \vdash M :^{\tau} A}$$

Proof. By induction on the derivation of $\Gamma, \Delta \vdash M :^{\tau} A$.

Step	Justification
AXIOM case	
$\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$	
1. $M = s_1$	hypothesis
2. $A = s_2$	hypothesis
3. $\Gamma, \Delta = \varepsilon$	hypothesis
4. $\Gamma = \varepsilon$	} by 3
5. $\Delta = \varepsilon$	
6. $(s_1, s_2) \in \mathcal{A}$	hypothesis
7. $\Gamma^\circ = \varepsilon$	by 4, def. of $\circ$
8. $\Gamma^\circ, \Delta \vdash s_1 :^{\mathsf{r}} s_2$	AXIOM, 6, 7, 5
9. $\Gamma^\circ, \Delta \vdash M :^{\mathsf{r}} A$	by 8, 1, 2
VAR case	
$\left(\dfrac{\Gamma' \vdash A :^{\mathsf{c}} s}{\Gamma', x{:}^{\mathsf{r}}A \vdash x :^{\mathsf{r}} A}\right)$	
1. $M = x$	hypothesis
2. $\Gamma, \Delta = \Gamma', x{:}^{\mathsf{r}}A$	hypothesis
3. $\Gamma' \vdash A :^{\mathsf{c}} s$	hypothesis
4. $(\Delta = \varepsilon) \vee (\Delta \neq \varepsilon)$	tautology
5. $\Delta = \varepsilon$	assumption
6. $\Gamma = \Gamma', x{:}^{\mathsf{r}}A$	by 2, 5
7. $\Gamma^\circ = \Gamma'^\circ, x{:}^{\mathsf{r}}A$	by 6, def. of $\circ$
8. $\Gamma'^\circ \vdash A :^{\mathsf{c}} s$	ind. hyp. on 3 with $\Delta := \varepsilon$
9. $\Gamma'^\circ, x{:}^{\mathsf{r}}A \vdash x :^{\mathsf{r}} A$	AXIOM, 8
10. $\Gamma^\circ, \Delta \vdash x :^{\mathsf{r}} A$	by 9, 7, 5
11. $\Delta \neq \varepsilon$	assumption

Step		Justification
	12. $\Delta = \Delta', x{:}^{\mathsf{r}} A$	by 2, 11 (for some Δ')
	13. $\Gamma' = \Gamma, \Delta'$	
	14. $\Gamma, \Delta' \vdash A :^{\mathsf{c}} s$	by 3, 13
	15. $\Gamma^\circ, \Delta' \vdash A :^{\mathsf{c}} s$	ind. hyp. on 14 with $\Delta := \Delta'$
	16. $\Gamma^\circ, \Delta', x{:}^{\mathsf{r}} A \vdash x :^{\mathsf{r}} A$	Axiom, 15
	17. $\Gamma^\circ, \Delta \vdash x :^{\mathsf{r}} A$	by 16, 12
18.	$\Gamma^\circ, \Delta \vdash x :^{\mathsf{r}} A$	$\vee$-elim, 4, 5–10, 11–17
19.	$\Gamma^\circ, \Delta \vdash M :^{\mathsf{r}} A$	by 18, 1

Weak case

$$\left(\frac{\Gamma' \vdash B :^{\mathsf{c}} s \qquad \Gamma' \vdash M :^{\mathsf{r}} A}{\Gamma', x{:}^{\tau} B \vdash M :^{\mathsf{r}} A}\right)$$

1.	$\Gamma, \Delta = \Gamma', x{:}^{\tau} B$	hypothesis
2.	$\Gamma' \vdash B :^{\mathsf{c}} s$	hypothesis
3.	$\Gamma' \vdash M :^{\mathsf{r}} A$	hypothesis
4.	$(\Delta = \varepsilon) \vee (\Delta \neq \varepsilon)$	tautology
	5. $\Delta = \varepsilon$	assumption
	6. $\Gamma = \Gamma', x{:}^{\tau} B$	by 1, 5
	7. $\Gamma^\circ = \Gamma'^\circ, x{:}^{\mathsf{r}} B$	by 6, def. of $\circ$
	8. $\Gamma'^\circ \vdash B :^{\mathsf{c}} s$	ind. hyp. on 2 with $\Delta := \varepsilon$
	9. $\Gamma'^\circ \vdash M :^{\mathsf{r}} A$	ind. hyp. on 3 with $\Delta := \varepsilon$
	10. $\Gamma'^\circ, x{:}^{\mathsf{r}} B \vdash M :^{\mathsf{r}} A$	Weak, 8, 9
	11. $\Gamma^\circ, \Delta \vdash M :^{\mathsf{r}} A$	by 10, 7, 5
	12. $\Delta \neq \varepsilon$	assumption
	13. $\Delta = \Delta', x{:}^{\tau} B$	by 1, 12 (for some Δ')
	14. $\Gamma' = \Gamma, \Delta'$	
	15. $\Gamma, \Delta' \vdash B :^{\mathsf{c}} s$	by 2, 14

Step		Justification
	16. $\Gamma, \Delta' \vdash M :^{\mathsf{r}} A$	by 3, 14
	17. $\Gamma^\circ, \Delta' \vdash B :^{\mathsf{c}} s$	ind. hyp. on 15 with $\Delta := \Delta'$
	18. $\Gamma^\circ, \Delta' \vdash M :^{\mathsf{r}} A$	ind. hyp. on 16 with $\Delta := \Delta'$
	19. $\Gamma^\circ, \Delta', x{:}^{\tau} B \vdash M :^{\mathsf{r}} A$	Weak, 17, 18
	20. $\Gamma^\circ, \Delta \vdash M :^{\mathsf{r}} A$	by 19, 13
21.	$\Gamma^\circ, \Delta \vdash M :^{\mathsf{r}} A$	$\vee$-elim, 4, 5–11, 12–20

Π-Form case

$$\left(\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma, \Delta \vdash B :^{\mathsf{r}} s_1 \qquad \Gamma, \Delta, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2}{\Gamma, \Delta \vdash \Pi^{\tau} x{:}B.\, C :^{\mathsf{r}} s_3}\right)$$

1.	$M = \Pi^{\tau} x{:}B.\, C$	hypothesis
2.	$A = s_3$	hypothesis
3.	$(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
4.	$\Gamma, \Delta \vdash B :^{\mathsf{r}} s_1$	hypothesis
5.	$\Gamma, \Delta, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2$	hypothesis
6.	$\Gamma^\circ, \Delta \vdash B :^{\mathsf{r}} s_1$	ind. hyp. on 4
7.	$\Gamma^\circ, \Delta, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2$	ind. hyp. on 5
8.	$\Gamma^\circ, \Delta \vdash \Pi^{\tau} x{:}B.\, C :^{\mathsf{r}} s_3$	Π-Form on 3, 6, 7
9.	$\Gamma^\circ, \Delta \vdash M :^{\mathsf{r}} A$	by 8, 1, 2

Π-Intro case (similar to the Π-Form case)

Π-Elim case (similar to the Π-Form case)

Conv case (similar to the Π-Form case)

Reset case

$$\left(\frac{(\Gamma, \Delta)^\circ \vdash M :^{\mathsf{r}} A}{\Gamma, \Delta \vdash M :^{\mathsf{c}} A}\right)$$

1.	$(\Gamma, \Delta)^\circ \vdash M :^{\mathsf{r}} A$	hypothesis
2.	$(\Gamma, \Delta)^\circ = \Gamma^\circ, \Delta^\circ$	def. of $\circ$ (almost)

Step	Justification
3. $\Gamma^\circ, \Delta^\circ \vdash M :^{\mathsf{r}} A$	by 1, 2
4. $\Gamma^\circ = \Gamma^{\circ\circ}$	Lemma A.1.1
5. $\Gamma^{\circ\circ}, \Delta^\circ \vdash M :^{\mathsf{r}} A$	by 3, 4
6. $(\Gamma^\circ, \Delta)^\circ = \Gamma^{\circ\circ}, \Delta^\circ$	def. of $\circ$ (almost)
7. $(\Gamma^\circ, \Delta)^\circ \vdash M :^{\mathsf{r}} A$	by 5, 6
5. $\Gamma^\circ, \Delta \vdash M :^{\mathsf{c}} A$	RESET, 7

□

Corollary A.1.3 (Phase Weakening)

$$\frac{\Gamma \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\mathsf{c}} A}$$

Proof. Assuming $\Gamma \vdash M :^{\mathsf{r}} A$, we obtain $\Gamma^\circ \vdash M :^{\mathsf{r}} A$ from Lemma A.1.2 (by setting $\Delta := \varepsilon$). Then RESET yields $\Gamma \vdash M :^{\mathsf{c}} A$. □

Lemma A.1.4 (Substitution Lemma)

$$\frac{\Gamma, x{:}^{\tau_1} A, \Delta \vdash M :^{\tau_2} B \qquad \Gamma \vdash N :^{\tau_1} A}{\Gamma, \Delta[N/x] \vdash M[N/x] :^{\tau_2} B[N/x]}$$

Proof. By induction on the derivation of $\Gamma, x{:}^{\tau_1} A, \Delta \vdash M :^{\tau_2} B$.

Step	Justification
AXIOM case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$	
this case is impossible, as it requires $\Gamma, x{:}^{\tau_1} A, \Delta = \varepsilon$	

Step	Justification
VAR case	
$\left(\dfrac{\Gamma' \vdash B :^{\mathsf{c}} s}{\Gamma', y{:}^{\mathsf{r}} B \vdash y :^{\mathsf{r}} B}\right)$	
1. $\Gamma \vdash N :^{\tau_1} A$	assumption
2. $M = y$	hypothesis
3. $\Gamma, x{:}^{\tau_1} A, \Delta = \Gamma', y{:}^{\mathsf{r}} B$	hypothesis
4. $\Gamma' \vdash B :^{\mathsf{c}} s$	hypothesis
5. $(\Delta = \varepsilon) \vee (\Delta \neq \varepsilon)$	tautology
6. $\Delta = \varepsilon$	assumption
7. $\Gamma = \Gamma'$	by 3, 6
8. $x = y$	by 3, 6
9. $\tau_1 = \mathsf{r}$	by 3, 6
10. $A = B$	by 3, 6
11. $\Gamma \vdash N :^{\mathsf{r}} B$	by 1, 8, 10
12. $\Delta[N/x] = \varepsilon$	by 6, def. of subst.
13. $M[N/x] = N$	by 2, 8, def. of subst.
14. $x \notin FV(B)$	b/c x $(= y)$ does not appear in Γ'
15. $B[N/x] = B$	by 14
16. $\Gamma, \Delta[N/x] \vdash M[N/x] :^{\mathsf{r}} B[N/x]$	by 11, 12, 13, 15
17. $\Delta \neq \varepsilon$	assumption
18. $\Delta = \Delta', y{:}^{\mathsf{r}} B$	by 3, 17 (for some Δ')
19. $\Gamma' = \Gamma, x{:}^{\tau_1} A, \Delta'$	by 3, 17 (for some Δ')
20. $\Gamma, x{:}^{\tau_1} A, \Delta' \vdash B :^{\mathsf{c}} s$	by 3, 19
21. $\Gamma, \Delta'[N/x] \vdash B[N/x] :^{\mathsf{c}} s[N/x]$	ind. hyp. on 20, 1
22. $s[N/x] = s$	def. of subst.
23. $\Gamma, \Delta'[N/x] \vdash B[N/x] :^{\mathsf{c}} s$	by 21, 22

Step		Justification
	24. $\Gamma, \Delta'[N/x], y{:}^{\mathsf{r}} B[N/x] \vdash y :^{\mathsf{r}} B[N/x]$	VAR, 23
	25. $\Gamma, \Delta[N/x] \vdash y :^{\mathsf{r}} B[N/x]$	by 24, 18, def. of subst.
	26. $x \neq y$	b/c they appear in the same context
	27. $M[N/x] = y$	by 2, 26, def. of subst.
	28. $\Gamma, \Delta[N/x] \vdash M[N/x] :^{\mathsf{r}} B[N/x]$	by 24, 26
28.	$\Gamma, \Delta[N/x] \vdash M[N/x] :^{\mathsf{r}} B[N/x]$	$\vee$-elim, 5, 6–16, 17–28

WEAK case

$$\left(\frac{\Gamma' \vdash C :^{\mathsf{c}} s \qquad \Gamma' \vdash M :^{\mathsf{r}} B}{\Gamma', y{:}^{\tau} C \vdash M :^{\mathsf{r}} B}\right)$$

1.	$\Gamma \vdash N :^{\tau_1} A$	assumption
2.	$\Gamma, x{:}^{\tau_1} A, \Delta = \Gamma', y{:}^{\tau} B$	hypothesis
3.	$\Gamma' \vdash C :^{\mathsf{c}} s$	hypothesis
4.	$\Gamma' \vdash M :^{\mathsf{r}} B$	hypothesis
5.	$(\Delta = \varepsilon) \vee (\Delta \neq \varepsilon)$	tautology
	6. $\Delta = \varepsilon$	assumption
	7. $\Gamma = \Gamma'$	by 2, 6
	8. $x = y$	by 2, 6
	9. $\tau_1 = \tau$	by 2, 6
	10. $A = B$	by 2, 6
	11. $y \notin FV(M)$	y is not bound in Γ'
	12. $y \notin FV(B)$	y is not bound in Γ'
	13. $M[N/x] = M$	by 11, 8
	14. $B[N/x] = B$	by 12, 8
	15. $\Delta[N/x] = \varepsilon$	by 6, def. of subst.
	16. $\Gamma, \Delta[N/x] \vdash M[N/x] :^{\mathsf{r}} B[N/x]$	by 4, 7, 15, 13, 14

Step		Justification
	17. $\Delta \neq \varepsilon$	assumption
	18. $\Delta = \Delta', y{:}^{\tau} C$	} by 2, 17 (for some Δ')
	19. $\Gamma' = \Gamma, x{:}^{\tau_1} A, \Delta'$	
	20. $\Gamma, x{:}^{\tau_1} A, \Delta' \vdash C :^{\mathsf{c}} s$	hypothesis
	21. $\Gamma, x{:}^{\tau_1} A, \Delta' \vdash M :^{\mathsf{r}} B$	hypothesis
	22. $\Gamma, \Delta'[N/x] \vdash C[N/x] :^{\mathsf{c}} s[N/x]$	ind. hyp. on 20, 1
	23. $\Gamma, \Delta'[N/x] \vdash C[N/x] :^{\mathsf{c}} s$	by 22, def. of subst.
	24. $\Gamma, \Delta'[N/x] \vdash M[N/x] :^{\mathsf{r}} B[N/x]$	ind. hyp. on 21, 1
25.	$\Gamma, \Delta'[N/x] \vdash M[N/x] :^{\mathsf{r}} B[N/x]$	$\vee$-elim, 5, 6–16, 17–24

Π-Form case

$$\left(\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma, x{:}^{\tau_1} A, \Delta \vdash C :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\tau_1} A, \Delta, x{:}^{\mathsf{r}} C \vdash D :^{\mathsf{r}} s_2}{\Gamma, x{:}^{\tau_1} A, \Delta \vdash \Pi^{\tau} x{:}C.\, D :^{\mathsf{r}} s_3}\right)$$

1.	$\Gamma \vdash N :^{\tau_1} A$	assumption
2.	$M = \Pi^{\tau} x{:}C.\, D$	hypothesis
3.	$B = s_3$	hypothesis
4.	$(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
5.	$\Gamma, x{:}^{\tau_1} A, \Delta \vdash C :^{\mathsf{r}} s_1$	hypothesis
6.	$\Gamma, x{:}^{\tau_1} A, \Delta, x{:}^{\mathsf{r}} C \vdash D :^{\mathsf{r}} s_2$	hypothesis
7.	$\Gamma, \Delta[N/x] \vdash C[N/x] :^{\mathsf{r}} s_1[N/x]$	ind. hyp. on 5, 1
8.	$\Gamma, \Delta[N/x] \vdash C[N/x] :^{\mathsf{r}} s_1$	by 7, def. of subst.
9.	$\Gamma, \Delta[N/x], x{:}^{\mathsf{r}} C[N/x] \vdash D[N/x] :^{\mathsf{r}} s_2[N/x]$	ind. hyp. on 6, 1
10.	$\Gamma, \Delta[N/x], x{:}^{\mathsf{r}} C[N/x] \vdash D[N/x] :^{\mathsf{r}} s_2$	by 9, def. of subst.
11.	$\Gamma, \Delta[N/x] \vdash (\Pi^{\tau} x{:}C.\, D)[N/x] :^{\mathsf{r}} s_3$	Π-Form, 10
12.	$\Gamma, \Delta[N/x] \vdash M[N/x] :^{\mathsf{r}} B[N/x]$	by 11, 2, 3, def. of subst.

Π-Intro case (similar to the Π-Form case.)

Π-Elim case (similar to the Π-Form case, but

Step	Justification
requires the identity $(B[N/x])[M[N/x]/y] = B[M/y][N/x]$.)	
Conv case (similar to the Π-Form case, but requires the lemma that $A =_\beta B$ implies $A[N/x] = B[N/x]$.)	
Reset case (similar to the Π-Form case, but requires the identity $\Delta^\circ[N/x] = (\Delta[N/x])^\circ$.)	

□

Lemma A.1.5 (Context Conversion)

$$\frac{\Gamma, x{:}^{\tau_1} A, \Delta \vdash M :^{\tau_2} C \qquad \Gamma \vdash B :^{\mathsf{c}} s \qquad A =_\beta B}{\Gamma, x{:}^{\tau_1} B, \Delta \vdash M :^{\tau_2} C}$$

Proof. By induction on the derivation of $\Gamma, x{:}^{\tau_1} A, \Delta \vdash M :^{\tau_2} C$.

Step	Justification
Axiom case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$	
1. $\varepsilon = \Gamma, x{:}^{\tau_1} A, \Delta$	hypothesis
2. contradiction!	1 is impossible
Var case $\left(\dfrac{\Gamma' \vdash C :^{\mathsf{c}} s}{\Gamma', y{:}^{\mathsf{r}} C \vdash y :^{\mathsf{r}} C}\right)$	
1. $\Gamma \vdash B :^{\mathsf{c}} s$	assumption
2. $A =_\beta B$	assumption
3. $\Gamma' \vdash C :^{\mathsf{c}} s$	hypothesis
4. $\Gamma', y{:}^{\mathsf{r}} C = \Gamma, x{:}^{\tau_1} A, \Delta$	hypothesis
5. $\Delta = \varepsilon \vee \Delta \neq \varepsilon$	tautology

Step		Justification
	6. $\Delta = \varepsilon$	assumption
	7. $\Gamma = \Gamma'$	by 4, 6
	8. $\tau_1 = \mathsf{r}$	
	9. $x = y$	
	10. $A = C$	
	11. $\Gamma', x{:}^{\mathsf{r}} B \vdash x :^{\mathsf{r}} B$	by 1, 7, VAR
	12. $B =_\beta C$	by 2, 10
	13. $\Gamma', x{:}^{\mathsf{r}} B \vdash x :^{\mathsf{r}} C$	CONV, 11, 3, 12
	14. $\Gamma, x{:}^{\tau_1} B, \Delta \vdash y :^{\mathsf{r}} C$	by 13, 7, 8, 6, 9
	15. $\Delta \neq \varepsilon$	assumption
	16. $\Delta = \Delta', y{:}^{\mathsf{r}} C$	by 4, 15 (for some Δ')
	17. $\Gamma' = \Gamma, x{:}^{\tau_1} A, \Delta'$	
	18. $\Gamma, x{:}^{\tau_1} A, \Delta' \vdash C :^{\mathsf{c}} s$	by 3, 17
	19. $\Gamma, x{:}^{\tau_1} B, \Delta' \vdash C :^{\mathsf{c}} s$	ind. hyp. on 18, 1, 2
	20. $\Gamma, x{:}^{\tau_1} B, \Delta', y{:}^{\mathsf{r}} C \vdash y :^{\mathsf{r}} C$	VAR, 19
	21. $\Gamma, x{:}^{\tau_1} B, \Delta \vdash y :^{\mathsf{r}} C$	by 20, 16
22.	$\Gamma, x{:}^{\tau_1} B, \Delta \vdash y :^{\mathsf{r}} C$	$\vee$-elim, 5, 6–14, 15–21

WEAK case

$$\left(\frac{\Gamma' \vdash D :^{\mathsf{c}} s \qquad \Gamma' \vdash M :^{\mathsf{r}} C}{\Gamma', y{:}^{\tau} D \vdash M :^{\mathsf{r}} C}\right)$$

1.	$\Gamma \vdash B :^{\mathsf{c}} s$	assumption
2.	$A =_\beta B$	assumption
3.	$\Gamma' \vdash D :^{\mathsf{c}} s$	hypothesis
4.	$\Gamma' \vdash M :^{\mathsf{r}} C$	hypothesis
5.	$\Gamma', y{:}^{\tau} D = \Gamma, x{:}^{\tau_1} A, \Delta$	hypothesis
6.	$\Delta = \varepsilon \vee \Delta \neq \varepsilon$	tautology

Step		Justification
7.	$\Delta = \varepsilon$	assumption
8.	$\Gamma = \Gamma'$	by 5, 7
9.	$\tau_1 = \tau$	by 5, 7
10.	$x = y$	by 5, 7
11.	$A = D$	by 5, 7
12.	$\Gamma', x{:}^{\tau_1} B \vdash M :^{\mathsf{r}} C$	Weak, 1, 8, 4
13.	$\Gamma, x{:}^{\tau_1} B, \Delta \vdash M :^{\mathsf{r}} C$	by 12, 8, 7
14.	$\Delta \neq \varepsilon$	assumption
15.	$\Delta = \Delta', y{:}^{\tau} D$	by 5, 14 (for some Δ')
16.	$\Gamma' = \Gamma, x{:}^{\tau_1} A, \Delta'$	by 5, 14 (for some Δ')
17.	$\Gamma, x{:}^{\tau_1} A, \Delta' \vdash D :^{\mathsf{c}} s$	by 3, 16
18.	$\Gamma, x{:}^{\tau_1} B, \Delta' \vdash D :^{\mathsf{c}} s$	ind. hyp. on 17, 1, 2
19.	$\Gamma, x{:}^{\tau_1} A, \Delta' \vdash M :^{\mathsf{r}} C$	by 4, 16
20.	$\Gamma, x{:}^{\tau_1} B, \Delta' \vdash M :^{\mathsf{r}} C$	ind. hyp. on 19, 1, 2
21.	$\Gamma, x{:}^{\tau_1} B, \Delta', y{:}^{\tau} D \vdash M :^{\mathsf{r}} C$	Weak, 18, 20
22.	$\Gamma, x{:}^{\tau_1} B, \Delta \vdash M :^{\mathsf{r}} C$	by 21, 15
23.	$\Gamma, x{:}^{\tau_1} B, \Delta \vdash M :^{\mathsf{r}} C$	$\vee$-elim, 6, 7–13, 14–22

Π-Form case

$$\left(\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma, x{:}^{\tau_1} A, \Delta \vdash C :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\tau_1} A, \Delta, y{:}^{\mathsf{r}} C \vdash D :^{\mathsf{r}} s_2}{\Gamma, x{:}^{\tau_1} A, \Delta \vdash \Pi^{\tau} y{:}C.\, D :^{\mathsf{r}} s_3} \right)$$

1.	$\Gamma \vdash B :^{\mathsf{c}} s$	assumption
2.	$A =_\beta B$	assumption
3.	$(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
4.	$\Gamma, x{:}^{\tau_1} A, \Delta \vdash C :^{\mathsf{r}} s_1$	hypothesis
5.	$\Gamma, x{:}^{\tau_1} A, \Delta, y{:}^{\mathsf{r}} C \vdash D :^{\mathsf{r}} s_2$	hypothesis
6.	$\Gamma, x{:}^{\tau_1} B, \Delta \vdash C :^{\mathsf{r}} s_1$	ind. hyp. on 4, 1, 2

Step	Justification
7. $\Gamma, x{:}^{\tau_1} B, \Delta, y{:}^{\mathsf{r}} C \vdash D :^{\mathsf{r}} s_2$	ind. hyp. on 5, 1, 2
8. $\Gamma, x{:}^{\tau_1} B, \Delta \vdash \Pi^{\tau} y{:}C.\, D :^{\mathsf{r}} s_3$	Π-Form on 3, 6, 7
Π-Intro, Π-Elim, and Conv cases are similar to the Π-Form case	
Reset case $\left(\dfrac{(\Gamma, x{:}^{\tau_1} A, \Delta)^\circ \vdash M :^{\mathsf{r}} C}{\Gamma, x{:}^{\tau_1} A, \Delta \vdash M :^{\mathsf{c}} C}\right)$	
1. $\Gamma^\circ \vdash B :^{\mathsf{c}} s$	assumption
2. $A =_\beta B$	assumption
3. $(\Gamma, x{:}^{\tau_1} A, \Delta)^\circ \vdash M :^{\mathsf{r}} C$	hypothesis
4. $(\Gamma, x{:}^{\tau_1} A, \Delta)^\circ = \Gamma^\circ, x{:}^{\mathsf{r}} A, \Delta^\circ$	def. of $\circ$
5. $\Gamma^\circ, x{:}^{\mathsf{r}} A, \Delta^\circ \vdash M :^{\mathsf{r}} C$	by 3, 4
6. $\Gamma^\circ, x{:}^{\mathsf{r}} B, \Delta^\circ \vdash M :^{\mathsf{r}} C$	ind. hyp. on 5, 1, 2
7. $\Gamma^\circ, x{:}^{\mathsf{r}} B, \Delta^\circ = (\Gamma, x{:}^{\tau_1} B, \Delta)^\circ$	def. of $\circ$
8. $(\Gamma, x{:}^{\tau_1} B, \Delta)^\circ \vdash M :^{\mathsf{r}} C$	by 6, 7
9. $\Gamma, x{:}^{\tau_1} B, \Delta \vdash M :^{\mathsf{c}} C$	Reset, 8

□

Lemma A.1.6 (Generalized Weakening)

$$\frac{\Gamma \vdash A :^{\mathsf{c}} s \qquad \Gamma, \Delta \vdash M :^{\tau'} B}{\Gamma, x{:}^{\tau} A, \Delta \vdash M :^{\tau'} B}$$

Proof. By induction on the derivation $\Gamma, \Delta \vdash M :^{\tau'} B$.

Step	Justification
Axiom case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$	

Step	Justification
1. $\Gamma \vdash A :^{\mathsf{c}} s$	assumption
2. $\Gamma = \Delta = \varepsilon$	hypothesis
3. $(s_1, s_2) \in \mathcal{A}$	hypothesis
4. $\vdash (s_1 :^{\mathsf{r}} s_2$	Axiom, 3
5. $x{:}^{\tau} A \vdash s_1 :^{\mathsf{r}} s_2$	Weak, 1, 4, 2

Var case

$$\left(\frac{\Gamma' \vdash B :^{\mathsf{c}} s}{\Gamma', y{:}^{\mathsf{r}} B \vdash y :^{\mathsf{r}} B}\right)$$

1.	$\Gamma \vdash A :^{\mathsf{c}} s$	assumption
2.	$\Gamma, \Delta = \Gamma', y{:}^{\mathsf{r}} B$	hypothesis
3.	$\Gamma' \vdash B :^{\mathsf{c}} s$	hypothesis
4.	$\Delta = \varepsilon \vee \Delta \neq \varepsilon$	tautology
5.	$\Delta = \varepsilon$	assumption
6.	$\Gamma = \Gamma', y{:}^{\mathsf{r}} B$	by 2, 5
7.	$\Gamma', y{:}^{\mathsf{r}} B \vdash y :^{\mathsf{r}} B$	Var, 3
8.	$\Gamma', y{:}^{\mathsf{r}} B \vdash A :^{\mathsf{c}} s$	by 1, 5, 6
9.	$\Gamma', y{:}^{\mathsf{r}} B, x{:}^{\tau} A \vdash y :^{\mathsf{r}} B$	Weak, 8, 7
10.	$\Gamma, x{:}^{\tau} A, \Delta \vdash y :^{\mathsf{r}} B$	by 9, 5, 6
11.	$\Delta \neq \varepsilon$	assumption
12.	$\Delta = \Delta', y{:}^{\mathsf{r}} B$	by 2, 11 (for some Δ')
13.	$\Gamma' = \Gamma, \Delta'$	by 2, 11 (for some Δ')
14.	$\Gamma, \Delta' \vdash B :^{\mathsf{c}} s$	by 3, 13
15.	$\Gamma, x{:}^{\tau} A, \Delta' \vdash B :^{\mathsf{c}} s$	ind. hyp. on 14, 1
16.	$\Gamma, x{:}^{\tau} A, \Delta', y{:}^{\mathsf{r}} B \vdash y :^{\mathsf{c}} B$	Var, 15
17.	$\Gamma, x{:}^{\tau} A, \Delta \vdash y :^{\mathsf{c}} B$	by 16, 12
18.	$\Gamma, x{:}^{\tau} A, \Delta \vdash y :^{\mathsf{c}} B$	$\vee$-elim, 4, 5–10, 11–17

Step	Justification
WEAK case	
$\left(\dfrac{\Gamma' \vdash B :^{\mathsf{c}} s \qquad \Gamma' \vdash M :^{\mathsf{r}} C}{\Gamma', y{:}^{\tau'} B \vdash M :^{\mathsf{r}} C}\right)$	
1. $\Gamma \vdash A :^{\mathsf{c}} s$	assumption
2. $\Gamma, \Delta = \Gamma', y{:}^{\tau'} B$	hypothesis
3. $\Gamma' \vdash B :^{\mathsf{c}} s$	hypothesis
4. $\Gamma' \vdash M :^{\mathsf{r}} C$	hypothesis
5. $\Delta = \varepsilon \vee \Delta \neq \varepsilon$	tautology
6. $\Delta = \varepsilon$	assumption
7. $\Gamma = \Gamma', y{:}^{\tau'} B$	by 2, 6
8. $\Gamma', y{:}^{\tau'} B \vdash A :^{\mathsf{c}} s$	by 1, 7
9. $\Gamma', y{:}^{\tau'} B \vdash M :^{\mathsf{r}} C$	WEAK, 3, 4
10. $\Gamma', y{:}^{\tau'} B, x{:}^{\tau} A \vdash M :^{\mathsf{r}} C$	WEAK, 8, 9
11. $\Gamma, x{:}^{\tau} A, \Delta \vdash M :^{\mathsf{r}} C$	by 10, 6, 7
12. $\Delta \neq \varepsilon$	assumption
13. $\Delta = \Delta', y{:}^{\tau'} B$	by 5, 14 (for some Δ')
14. $\Gamma' = \Gamma, \Delta'$	by 5, 14 (for some Δ')
15. $\Gamma, \Delta' \vdash B :^{\mathsf{c}} s$	by 14, 3
16. $\Gamma, \Delta' \vdash M :^{\mathsf{r}} C$	by 14, 4
17. $\Gamma, x{:}^{\tau} A, \Delta' \vdash B :^{\mathsf{c}} s$	ind. hyp. on 15, 1
18. $\Gamma, x{:}^{\tau} A, \Delta' \vdash M :^{\mathsf{r}} C$	ind. hyp. on 16, 1
19. $\Gamma, x{:}^{\tau} A, \Delta', y{:}^{\tau'} B \vdash M :^{\mathsf{r}} C$	WEAK, 17, 18
20. $\Gamma, x{:}^{\tau} A, \Delta \vdash M :^{\mathsf{r}} C$	by 13, 19
23. $\Gamma, x{:}^{\tau} B, \Delta \vdash M :^{\mathsf{r}} C$	$\vee$-elim, 5, 6–11, 12–20

Step	**Justification**
Π-Form case	
$\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma, \Delta \vdash B :^{\mathsf{r}} s_1 \qquad \Gamma, \Delta, y{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2}{\Gamma, \Delta \vdash \Pi^{\tau'} y{:}B.\, C :^{\mathsf{r}} s_3}\right)$	
1. $\Gamma \vdash A :^{\mathsf{c}} s$	assumption
2. $(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
3. $\Gamma, \Delta \vdash B :^{\mathsf{r}} s_1$	hypothesis
4. $\Gamma, \Delta, y{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2$	hypothesis
5. $\Gamma, x{:}^{\tau} A, \Delta \vdash B :^{\mathsf{r}} s_1$	ind. hyp. on 3, 1
6. $\Gamma, x{:}^{\tau} A, \Delta, y{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2$	ind. hyp. on 4, 1
7. $\Gamma, x{:}^{\tau} A, \Delta \vdash \Pi^{\tau'} y{:}B.\, C :^{\mathsf{r}} s_3$	Π-Form, 2, 5, 6
Π-Intro, Π-Elim, and Conv cases are similar to the Π-Form case	
Reset case	
$\left(\dfrac{(\Gamma, \Delta)^{\circ} \vdash M :^{\mathsf{r}} B}{\Gamma, \Delta \vdash M :^{\mathsf{c}} B}\right)$	
1. $\Gamma \vdash A :^{\mathsf{c}} s$	assumption
2. $(\Gamma, \Delta)^{\circ} \vdash M :^{\mathsf{r}} B$	hypothesis
3. $(\Gamma, \Delta)^{\circ} = \Gamma^{\circ}, \Delta^{\circ}$	
4. $\Gamma^{\circ}, \Delta^{\circ} \vdash M :^{\mathsf{r}} B$	by 2, 3
5. $\Gamma^{\circ}, x{:}^{\mathsf{r}} A, \Delta^{\circ} \vdash M :^{\mathsf{r}} B$	ind. hyp. on 4, 1
6. $(\Gamma, x{:}^{\tau} A, \Delta)^{\circ} = \Gamma^{\circ}, x{:}^{\mathsf{r}} A, \Delta^{\circ}$	def. of $\circ$
7. $(\Gamma, x{:}^{\tau} A, \Delta)^{\circ} \vdash M :^{\mathsf{r}} B$	by 5, 6
8. $\Gamma, x{:}^{\tau} A, \Delta \vdash M :^{\mathsf{c}} B$	Reset, 7

□

Lemma A.1.7 (Π-Inversion)

$$\frac{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{r} C}{(\exists\, (s_1, s_2, s_3) \in \mathcal{R}) \quad C =_{\beta} s_3 \;\wedge\; \Gamma \vdash A :^{r} s_1 \;\wedge\; \Gamma, x{:}^{r}A \vdash B :^{r} s_2}$$

Proof. By induction on the derivation of $\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} C$. The only rules by which this judgment can possibly be derived are Π-Form, Weak, and Conv, so we omit all other (trivial) cases of the proof.

Step	**Justification**
Weak case $\left(\dfrac{\Gamma \vdash D :^{\mathsf{c}} s \qquad \Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} C}{\Gamma, y{:}^{\tau'} D \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} C}\right)$	
1. $\Gamma \vdash D :^{\mathsf{c}} s$	hypothesis
2. $\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} C$	hypothesis
3. $(s_1, s_2, s_3) \in \mathcal{R}$	ind. hyp. on 2
4. $C =_{\beta} s_3$	ind. hyp. on 2
5. $\Gamma \vdash A :^{\mathsf{r}} s_1$	ind. hyp. on 2
6. $\Gamma, x{:}^{\mathsf{r}}A \vdash B :^{\mathsf{r}} s_2$	ind. hyp. on 2
7. $\Gamma, y{:}^{\tau'} D \vdash A :^{\mathsf{r}} s_1$	Lemma A.1.6, 1, 5
8. $\Gamma, y{:}^{\tau'} D, x{:}^{\mathsf{r}}A \vdash B :^{\mathsf{r}} s_2$	Lemma A.1.6, 1, 6
9. $(\exists\, (s_1, s_2, s_3) \in \mathcal{R})$ $C =_{\beta} s_3 \;\wedge\; \Gamma, y{:}^{\tau'} D \vdash A :^{\mathsf{r}} s_1$ $\wedge\; \Gamma, y{:}^{\tau'} D, x{:}^{\mathsf{r}}A \vdash B :^{\mathsf{r}} s_2$	by 3, 4, 7, 8
Π-Form case $\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}}A \vdash B :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} s_3}\right)$	
1. $C = s_3$	hypothesis
2. $(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis

Step	**Justification**
3. $\Gamma \vdash A :^{\mathsf{r}} s_1$	hypothesis
4. $\Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	hypothesis
5. $C =_\beta s_3$	by 1
6. $(\exists\, (s_1, s_2, s_3) \in \mathcal{R})$	
$\quad C =_\beta s_3 \ \wedge\ \Gamma \vdash A :^{\mathsf{r}} s_1$	
$\quad \wedge\ \Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	by 2, 5, 3, 4

CONV case

$$\left(\frac{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} D \qquad \Gamma \vdash C :^{\mathsf{c}} s \qquad D =_\beta C}{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} C}\right)$$

1. $\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} D$	hypothesis
2. $\Gamma \vdash C :^{\mathsf{c}} s$	hypothesis
3. $D =_\beta C$	hypothesis
4. $(s_1, s_2, s_3) \in \mathcal{R}$	ind. hyp. on 1
5. $D =_\beta s_3$	
6. $\Gamma \vdash A :^{\mathsf{r}} s_1$	
7. $\Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	
8. $C =_\beta s_3$	
9. $(\exists\, (s_1, s_2, s_3) \in \mathcal{R})$	
$\quad C =_\beta s_3 \ \wedge\ \Gamma \vdash A :^{\mathsf{r}} s_1$	
$\quad \wedge\ \Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	by 4, 8, 6, 7

□

Corollary A.1.8 (Π-Inversion in c-mode)

$$\frac{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} C}{(\exists\, (s_1, s_2, s_3) \in \mathcal{R}) \quad C =_\beta s_3 \ \wedge\ \Gamma \vdash A :^{\mathsf{c}} s_1 \ \wedge\ \Gamma, x{:}^{\tau} A \vdash B :^{\mathsf{c}} s_2}$$

Proof.

Step		Justification
1.	$\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} C$	assumption
2.	$\Gamma^{\circ} \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} C$	inversion on 1
3.	$(s_1, s_2, s_3) \in \mathcal{R}$	by 2 (Lemma A.1.7)
4.	$C =_{\beta} s_3$	by 2 (Lemma A.1.7)
5.	$\Gamma^{\circ} \vdash A :^{\mathsf{r}} s_1$	by 2 (Lemma A.1.7)
6.	$\Gamma^{\circ}, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	by 2 (Lemma A.1.7)
7.	$\Gamma \vdash A :^{\mathsf{c}} s_1$	Reset, 5
8.	$(\Gamma, x{:}^{\tau} A)^{\circ} = \Gamma^{\circ}, x{:}^{\mathsf{r}} A$	def. of $\circ$
9.	$(\Gamma, x{:}^{\tau} A)^{\circ} \vdash B :^{\mathsf{r}} s_2$	by 6, 8
10.	$\Gamma, x{:}^{\tau} A \vdash B :^{\mathsf{c}} s_2$	Reset, 9
11.	$(\exists\, (s_1, s_2, s_3) \in \mathcal{R})$ $C =_{\beta} s_3 \ \wedge\ \Gamma \vdash A :^{\mathsf{c}} s_1$ $\wedge\ \Gamma, x{:}^{\tau} A \vdash B :^{\mathsf{c}} s_2$	by 3, 4, 7, 10

□

Lemma A.1.9 (λ-Inversion)

$$\frac{\Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{r} C}{(\exists\, B, s) \quad C =_{\beta} \Pi^{\tau} x{:}A.\, B \ \wedge\ \Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s \ \wedge\ \Gamma, x{:}^{\tau} A \vdash M :^{r} B}$$

Proof. By induction on the derivation of $\Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} C$. The only rules by which this judgment can possibly be derived are Π-Intro, Weak, and Conv, so we omit all other (trivial) cases of the proof.

Step	Justification
Weak case $\left(\dfrac{\Gamma \vdash D :^{\mathsf{c}} s \qquad \Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} C}{\Gamma, y{:}^{\tau'} D \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} C}\right)$	
1. $\Gamma \vdash D :^{\mathsf{c}} s$	hypothesis

Step	Justification
2. $\Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} C$	hypothesis
3. $C =_{\beta} \Pi^{\tau} x{:}A.\, B$	ind. hyp. on 2
4. $\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s$	ind. hyp. on 2
5. $\Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B$	ind. hyp. on 2
6. $\Gamma, y{:}^{\tau'} D, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B$	Lemma A.1.6, 1, 5
7. $(\exists\, B, s)\ C =_{\beta} \Pi^{\tau} x{:}A.\, B$ $\wedge\ \Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s$ $\wedge\ \Gamma, y{:}^{\tau'} D, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B$	by 3, 4, 6

Π-Intro case

$$\left(\frac{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s \qquad \Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B}{\Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B}\right)$$

1. $C = \Pi^{\tau} x{:}A.\, B$	hypothesis
2. $\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s$	hypothesis
3. $\Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B$	hypothesis
4. $C =_{\beta} \Pi^{\tau} x{:}A.\, B$	by 1
5. $(\exists\, B, s)\ C =_{\beta} \Pi^{\tau} x{:}A.\, B$ $\wedge\ \Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s$ $\wedge\ \Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B$	by 4, 2, 3

Conv case

$$\left(\frac{\Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} D \qquad \Gamma \vdash C :^{\mathsf{c}} s \qquad D =_{\beta} C}{\Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} C}\right)$$

1. $\Gamma \vdash \lambda^{\tau} x{:}A.\, M :^{\mathsf{r}} D$	hypothesis
2. $\Gamma \vdash C :^{\mathsf{c}} s$	hypothesis
3. $D =_{\beta} C$	hypothesis
4. $D =_{\beta} \Pi^{\tau} x{:}A.\, B$	ind. hyp. on 1
5. $\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s$	ind. hyp. on 1

Step	Justification
6. $\Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B$	(also) ind. hyp. on 1
7. $C =_\beta \Pi^{\tau} x{:}A.\, B$	by 3, 4
8. $(\exists\, B, s)\ C =_\beta \Pi^{\tau} x{:}A.\, B$	
$\wedge\ \Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s$	
$\wedge\ \Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B$	by 7, 5, 6

□

Lemma A.1.10 (@-Inversion)

$$\frac{\Gamma \vdash M@^{\tau} N :^{r} C}{(\exists\, x, A, B) \quad C =_\beta B[N/x] \ \wedge\ \Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B \ \wedge\ \Gamma \vdash N :^{\tau} A}$$

Proof. By induction on the derivation of $\Gamma \vdash M@^{\tau} N :^{\mathsf{r}} C$. The only rules by which this judgment can possibly be derived are Π-Elim, Weak, and Conv, so we omit all other (trivial) cases of the proof.

Step	Justification
Weak case $\left(\dfrac{\Gamma \vdash D :^{\mathsf{c}} s \qquad \Gamma \vdash M@^{\tau} N :^{\mathsf{r}} C}{\Gamma, y{:}^{\tau'} D \vdash M@^{\tau} N :^{\mathsf{r}} C}\right)$	
1. $\Gamma \vdash D :^{\mathsf{c}} s$	hypothesis
2. $\Gamma \vdash M@^{\tau} N :^{\mathsf{r}} C$	hypothesis
3. $C =_\beta B[N/x]$	ind. hyp. on 2 (for some x, A, B)
4. $\Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	ind. hyp. on 2 (for some x, A, B)
5. $\Gamma \vdash N :^{\tau} A$	ind. hyp. on 2 (for some x, A, B)
6. $\Gamma, y{:}^{\tau'} D \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	Weak, 1, 4
7. $\Gamma, y{:}^{\tau'} D \vdash N :^{\tau} A$	Lemma A.1.6, 1, 5
8. $(\exists\, x, A, B)\ C =_\beta B[N/x]$	
$\wedge\ \Gamma, y{:}^{\tau'} D \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	

Step	**Justification**
$\wedge\ \Gamma, y{:}^{\tau'} D \vdash N :^{\tau} A$	by 3, 6, 7

Π-Elim case

$$\left(\frac{\Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B \qquad \Gamma \vdash N :^{\tau} A}{\Gamma \vdash M@^{\tau} N :^{\mathsf{r}} B[N/x]}\right)$$

1.	$C = B[N/x]$	hypothesis
2.	$\Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	hypothesis
3.	$\Gamma \vdash N :^{\tau} A$	hypothesis
4.	$C =_{\beta} B[N/x]$	by 1
5.	$(\exists\, x, A, B)\ C =_{\beta} B[N/x]$ $\wedge\ \Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$ $\wedge\ \Gamma \vdash N :^{\tau} A$	by 4, 2, 3

Conv case

$$\left(\frac{\Gamma \vdash M@^{\tau} N :^{\mathsf{r}} D \qquad \Gamma \vdash C :^{\mathsf{c}} s \qquad D =_{\beta} C}{\Gamma \vdash M@^{\tau} N :^{\mathsf{r}} C}\right)$$

1.	$\Gamma \vdash M@^{\tau} N :^{\mathsf{r}} D$	hypothesis
2.	$\Gamma \vdash C :^{\mathsf{c}} s$	hypothesis
3.	$D =_{\beta} C$	hypothesis
4.	$D =_{\beta} B[N/x]$	ind. hyp. on 1
5.	$\Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	
6.	$\Gamma \vdash N :^{\tau} A$	
7.	$C =_{\beta} B[N/x]$	by 3, 4
8.	$(\exists\, x, A, B)\ C =_{\beta} B[N/x]$ $\wedge\ \Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$ $\wedge\ \Gamma \vdash N :^{\tau} A$	by 7, 5, 6

□

Lemma A.1.11 (Coherence Lemma)

$$\frac{\Gamma \vdash M :^{\tau} A}{(\exists s) \quad A = s \quad \vee \quad \Gamma \vdash A :^{\mathsf{c}} s}$$

Proof. By induction on the derivation of $\Gamma \vdash M :^{\tau} A$.

Step	**Justification**
AXIOM case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$	
1. $A = s_2$	hypothesis
VAR case $\left(\dfrac{\Gamma \vdash A :^{\mathsf{c}} s}{\Gamma, x{:}^{\mathsf{r}}A \vdash x :^{\mathsf{r}} A}\right)$	
1. $\Gamma \vdash A :^{\mathsf{c}} s$	hypothesis
2. $\Gamma^{\circ} \vdash A :^{\mathsf{r}} s$	inversion on 1
3. $\Gamma^{\circ} \vdash A :^{\mathsf{c}} s$	Corollary A.1.3 on 2
4. $\Gamma^{\circ}, x{:}^{\mathsf{r}}A \vdash A :^{\mathsf{r}} s$	WEAK, 3, 2
5. $\Gamma^{\circ}, x{:}^{\mathsf{r}}A = (\Gamma, x{:}^{\mathsf{r}}A)^{\circ}$	def. of $\circ$
6. $\Gamma, x{:}^{\mathsf{r}}A \vdash A :^{\mathsf{c}} s$	by 4, 5, RESET
WEAK case $\left(\dfrac{\Gamma \vdash B :^{\mathsf{c}} s \qquad \Gamma \vdash M :^{\mathsf{r}} A}{\Gamma, x{:}^{\tau}B \vdash M :^{\mathsf{r}} A}\right)$	
1. $\Gamma \vdash B :^{\mathsf{c}} s$	hypothesis
2. $\Gamma \vdash M :^{\mathsf{r}} A$	hypothesis
3. $A = s' \vee \Gamma \vdash A :^{\mathsf{c}} s'$	ind. hyp. on 2 (for some s')
4. $A = s'$	assumption
5. $(\exists s')\ A = s' \vee \Gamma, x{:}^{\tau}B \vdash A :^{\mathsf{c}} s'$	by 4

Step		Justification
	6. $\Gamma \vdash A :^{\mathsf{c}} s'$	assumption
	7. $\Gamma^\circ \vdash A :^{\mathsf{r}} s'$	inversion on 6
	8. $\Gamma^\circ \vdash A :^{\mathsf{c}} s'$	Corollary A.1.3 on 7
	9. $\Gamma^\circ, x{:}^{\mathsf{r}}A \vdash A :^{\mathsf{r}} s'$	Weak on 8, 7
	10. $\Gamma^\circ, x{:}^{\mathsf{r}}A = (\Gamma, x{:}^{\tau}A)^\circ$	def. of $\circ$
	11. $\Gamma, x{:}^{\tau}A \vdash A :^{\mathsf{c}} s'$	Reset, 9, 10
	12. $(\exists s')\ A = s' \vee \Gamma, x{:}^{\tau}B \vdash A :^{\mathsf{c}} s'$	by 11
13.	$(\exists s')\ A = s' \vee \Gamma, x{:}^{\tau}B \vdash A :^{\mathsf{c}} s'$	$\vee$-elim, 3, 4–5, 6–12

Π-Form case

$$\left(\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash B :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}}B \vdash C :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\tau}x{:}B.\,C :^{\mathsf{r}} s_3}\right)$$

1.	$A = s_3$	hypothesis

Π-Intro case

$$\left(\frac{\Gamma \vdash \Pi^{\tau}x{:}B.\,C :^{\mathsf{c}} s \qquad \Gamma, x{:}^{\tau}B \vdash M :^{\mathsf{r}} C}{\Gamma \vdash \lambda^{\tau}x{:}B.\,M :^{\mathsf{r}} \Pi^{\tau}x{:}B.\,C}\right)$$

1.	$A = \Pi^{\tau}x{:}B.\,C$	hypothesis
2.	$\Gamma \vdash \Pi^{\tau}x{:}B.\,C :^{\mathsf{c}} s$	hypothesis
3.	$\Gamma \vdash A :^{\mathsf{c}} s$	by 1, 2

Π-Elim case

$$\left(\frac{\Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau}x{:}B.\,C \qquad \Gamma \vdash N :^{\tau} B}{\Gamma \vdash M@^{\tau}N :^{\mathsf{r}} C[N/x]}\right)$$

1.	$A = C[N/x]$	hypothesis
2.	$\Gamma \vdash M :^{\mathsf{r}} \Pi^{\tau}x{:}B.\,C$	hypothesis
3.	$\Gamma \vdash N :^{\tau} B$	hypothesis
4.	$\Pi^{\tau}x{:}B.\,C = s \vee \Gamma \vdash \Pi^{\tau}x{:}B.\,C :^{\mathsf{c}} s$	ind. hyp. on 2 (for some s)
5.	$\Gamma \vdash \Pi^{\tau}x{:}B.\,C :^{\mathsf{c}} s$	by 4

Step	Justification
6. $\Gamma, x{:}^{\tau} B \vdash C :^{\mathsf{c}} s'$	Corollary A.1.8 on 5 (for some s')
7. $\Gamma \vdash C[N/x] :^{\mathsf{c}} s'[N/x]$	Lemma A.1.4, 6, 3
8. $s'[N/x] = s'$	def. of subst.
9. $\Gamma \vdash A :^{\mathsf{c}} s'$	by 1, 7, 8
CONV case	
$\left(\dfrac{\Gamma \vdash M :^{\mathsf{r}} B \qquad \Gamma \vdash A :^{\mathsf{c}} s \qquad B =_\beta A}{\Gamma \vdash M :^{\mathsf{r}} A}\right)$	
1. $\Gamma \vdash A :^{\mathsf{c}} s$	hypothesis
RESET case	
$\left(\dfrac{\Gamma^\circ \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\mathsf{c}} A}\right)$	
1. $\Gamma^\circ \vdash M :^{\mathsf{r}} A$	hypothesis
2. $A = s \vee \Gamma^\circ \vdash A :^{\mathsf{c}} s$	ind. hyp. on (for some s)
3. $A = s$	assumption
4. $(\exists s)\ A = s \vee \Gamma \vdash A :^{\mathsf{c}} s$	by 3
5. $\Gamma^\circ \vdash A :^{\mathsf{c}} s$	assumption
6. $\Gamma^{\circ\circ} \vdash A :^{\mathsf{r}} s$	by 5
7. $\Gamma^\circ \vdash A :^{\mathsf{r}} s$	by 6, Lemma A.1.1
8. $\Gamma \vdash A :^{\mathsf{c}} s$	by 7, RESET
9. $(\exists s)\ A = s \vee \Gamma \vdash A :^{\mathsf{c}} s$	by 8
10 $(\exists s)\ A = s \vee \Gamma \vdash A :^{\mathsf{c}} s$	$\vee$-elim, 3–4, 5–9

□

Corollary A.1.12 ($\lambda\Pi$-Inversion)

$$\frac{\Gamma \vdash \lambda^{\tau'} x{:}A'.\, M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B}{(\exists\, B) \quad \tau' = \tau \ \wedge\ \Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B}$$

Proof.

Step		Justification
1.	$\Gamma \vdash \lambda^{\tau'} x{:}A'.\, M :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	assumption
2.	$\Pi^{\tau} x{:}A.\, B =_\beta \Pi^{\tau'} x{:}A'.\, B'$	by 1
3.	$\Gamma \vdash \Pi^{\tau'} x{:}A'.\, B' :^{\mathsf{c}} s$	by 1
4.	$\Gamma, x{:}^{\tau'} A' \vdash M :^{\mathsf{r}} B'$	by 1 (Lemma A.1.9)
6.	$\tau' = \tau$	by 2
7.	$A' =_\beta A$	by 2
8.	$B' =_\beta B$	by 2
9.	$\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{c}} s_3$	Lemma A.1.11, 1
10.	$\Gamma \vdash A :^{\mathsf{c}} s_1$	by 9
11.	$\Gamma, x{:}^{\tau} A \vdash B :^{\mathsf{c}} s_2$	by 9 (Lemma A.1.8)
12.	$\Gamma, x{:}^{\tau'} A \vdash M :^{\mathsf{r}} B'$	Lemma A.1.5, 4, 10, 7
13.	$\Gamma, x{:}^{\tau'} A \vdash M :^{\mathsf{r}} B$	Conv, 12, 11, 8
14.	$(\exists\, B)\ \tau' = \tau\ \wedge\ \Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B$	by 6, 13

□

Lemma A.1.13 (Subject Reduction)

$$\frac{\Gamma \vdash M :^{\tau} A \qquad M \to_\beta N}{\Gamma \vdash N :^{\tau} A}$$

Proof. By induction on the derivation of $\Gamma \vdash M :^{\tau} A$.

Step		Justification
Axiom case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$		
1.	$M \to_\beta N$	assumption
2.	$M = s_1$	hypothesis

Step	**Justification**
3. contradiction!	by 1, 2, def. of $\rightarrow_\beta$
VAR case $\left(\dfrac{\Gamma \vdash A :^{\mathsf{c}} s}{\Gamma, x{:}^{\mathsf{r}} A \vdash x :^{\mathsf{r}} A}\right)$	
1. $M \rightarrow_\beta N$	assumption
2. $M = x$	hypothesis
3. contradiction!	by 1, 2, def. of $\rightarrow_\beta$
WEAK case $\left(\dfrac{\Gamma \vdash B :^{\mathsf{c}} s \qquad \Gamma \vdash M :^{\mathsf{r}} A}{\Gamma, x{:}^{\tau} B \vdash M :^{\mathsf{r}} A}\right)$	
1. $M \rightarrow_\beta N$	assumption
2. $\Gamma \vdash B :^{\mathsf{c}} s$	hypothesis
3. $\Gamma \vdash M :^{\mathsf{r}} A$	hypothesis
4. $\Gamma \vdash N :^{\mathsf{r}} A$	ind. hyp. on 3, 1
5. $\Gamma, x{:}^{\tau} B \vdash N :^{\mathsf{r}} A$	WEAK, 2, 4
Π-FORM case $\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash B :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\tau} x{:}B.\, C :^{\mathsf{r}} s_3}\right)$	
1. $M \rightarrow_\beta N$	assumption
2. $M = \Pi^{\tau} x{:}B.\, C$	hypothesis
3. $(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
4. $\Gamma \vdash B :^{\mathsf{r}} s_1$	hypothesis
5. $\Gamma, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2$	hypothesis
6. $(\exists\, B'.\, B \rightarrow_\beta B' \wedge N = \Pi^{\tau} x{:}B'.\, C) \vee$ $(\exists\, C'.\, C \rightarrow_\beta C' \wedge N = \Pi^{\tau} x{:}B.\, C')$	by 1, 2, def. of $\rightarrow_\beta$
$\Big[$ 7. $B \rightarrow_\beta B'$	$\Big\}$ assumption ...

Step		Justification
	8. $N = \Pi^{\tau} x{:}B'.\, C$	} ... (for some B')
	9. $\Gamma \vdash B' :^{\mathsf{r}} s_1$	ind. hyp. on 4, 7
	10. $B =_{\beta} B'$	by 7
	11. $\Gamma, x{:}^{\mathsf{r}} B' \vdash C :^{\mathsf{r}} s_2$	Lemma A.1.5 on 5, 9, 10
	12. $\Gamma \vdash \Pi^{\tau} x{:}B'.\, C :^{\mathsf{r}} s_3$	Π-Form on 3, 9, 11
	13. $\Gamma \vdash N :^{\mathsf{r}} s_3$	by 12, 8
	14. $C \to_{\beta} C'$	} assumption
	15. $N = \Pi^{\tau} x{:}B.\, C'$	} (for some C')
	16. $\Gamma, x{:}^{\mathsf{r}} B \vdash C' :^{\mathsf{r}} s_2$	ind. hyp. on 5, 14
	17. $\Gamma \vdash \Pi^{\tau} x{:}B.\, C' :^{\mathsf{r}} s_3$	Π-Form on 3, 4, 16
	18. $\Gamma \vdash N :^{\mathsf{r}} s_3$	by 17, 15
19.	$\Gamma \vdash N :^{\mathsf{r}} s_3$	∨-elim, 6, 7–13, 14–18

Π-Intro case is similar to the Π-Form case.

Π-Elim case

$$\left(\frac{\Gamma \vdash P :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B \qquad \Gamma \vdash Q :^{\tau} A}{\Gamma \vdash P@^{\tau}Q :^{\mathsf{r}} B[Q/x]}\right)$$

1.	$M \to_{\beta} N$	assumption
2.	$M = P@^{\tau}Q$	hypothesis
3.	$\Gamma \vdash P :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	hypothesis
4.	$\Gamma \vdash Q :^{\tau} A$	hypothesis
5.	$((\exists\, P')\ P \to_{\beta} P' \wedge N = P'@^{\tau}Q) \vee$ $((\exists\, Q')\ Q \to_{\beta} Q' \wedge N = P@^{\tau}Q') \vee$ $((\exists\, A'\ P')\ P = \lambda^{\tau'} x{:}A'.\, P'$ $\wedge\ N = P'[Q/x])$	by 1, def. of $\to_{\beta}$
	6. $P \to_{\beta} P'$	} assumption
	7. $N = P'@^{\tau}Q$	} (for some P')

Step		Justification
8.	$\Gamma \vdash P' :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	ind. hyp. on 3
9.	$\Gamma \vdash P'@^{\tau}Q :^{\mathsf{r}} B[Q/x]$	Π-Elim, 8, 4
10.	$\Gamma \vdash N :^{\mathsf{r}} B[Q/x]$	by 9, 7
11.	$Q \to_{\beta} Q'$	assumption (for some Q')
12.	$N = P@^{\tau}Q'$	assumption (for some Q')
13.	$\Gamma \vdash Q' :^{\tau} A$	ind. hyp. on 4, 11
14.	$\Gamma \vdash P@^{\tau}Q' :^{\mathsf{r}} B[Q/x]$	Π-Elim, 3, 13
15.	$\Gamma \vdash N :^{\mathsf{r}} B[Q/x]$	by 14, 12
16.	$P = \lambda^{\tau'} x{:}A'.\, P'$	assumption (for some A', P')
17.	$N = P'[Q/x]$	assumption (for some A', P')
18.	$\Gamma \vdash \lambda^{\tau'} x{:}A'.\, P' :^{\mathsf{r}} \Pi^{\tau} x{:}A.\, B$	by 3, 16
19.	$\Gamma, x{:}^{\tau} A \vdash P' :^{\mathsf{r}} B$	by 18, Corollary A.1.12
20.	$\Gamma \vdash P'[Q/x] :^{\mathsf{r}} B[Q/x]$	Lemma A.1.4, 19, 4
21.	$\Gamma \vdash N :^{\mathsf{r}} B[Q/x]$	by 20, 17
22.	$\Gamma \vdash N :^{\mathsf{r}} B[Q/x]$	$\vee$-elim, 5, 6–10, 11–15, 16–21

Conv case

$$\left(\frac{\Gamma \vdash M :^{\mathsf{r}} B \qquad \Gamma \vdash A :^{\mathsf{c}} s \qquad B =_{\beta} A}{\Gamma \vdash M :^{\mathsf{r}} A}\right)$$

1.	$M \to_{\beta} N$	assumption
2.	$\Gamma \vdash M :^{\mathsf{r}} B$	hypothesis
3.	$\Gamma \vdash A :^{\mathsf{c}} s$	hypothesis
4.	$B =_{\beta} A$	hypothesis
5.	$\Gamma \vdash N :^{\mathsf{r}} B$	ind. hyp. on 2, 1
6.	$\Gamma \vdash N :^{\mathsf{r}} A$	Conv on 5, 3, 4

Reset case is similar to the Conv case

Step	Justification

□

A.2 META-THEORY OF ERASURE

The RV operation gathers all run-time variables in a context Γ into a set. It was defined in Definition 3.3.2 as follows:

$$RV(\varepsilon) = \emptyset \qquad RV(\Gamma, x{:}^{\mathsf{r}} A) = RV(\Gamma) \cup \{x\} \qquad RV(\Gamma, x{:}^{\mathsf{c}} A) = RV(\Gamma)$$

Lemma A.2.1 (Variable Survival)

$$\frac{\Gamma \vdash M :^{r} A}{FV(M^{\bullet}) \subseteq RV(\Gamma)}$$

Proof. By induction on the derivation of $\Gamma \vdash M :^{\mathsf{r}} A$.

Step	Justification
AXIOM case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$	
1. $M = s_1$	hypothesis
2. ${s_1}^{\bullet} = s_1$	def. of $\bullet$
3. $FV(s_1) = \emptyset$	def. of FV
4. $\emptyset \subseteq RV(\Gamma)$	def. of $\subseteq$
5. $FV(M^{\bullet}) \subseteq RV(\Gamma)$	by 1, 2, 3, 4
VAR case $\left(\dfrac{\Delta \vdash A :^{\mathsf{c}} s}{\Delta, x{:}^{\mathsf{r}} A \vdash x :^{\mathsf{r}} A}\right)$	
1. $M = x$	hypothesis
2. $\Gamma = \Delta, x{:}^{\mathsf{r}} A$	hypothesis

Step	Justification
3. $x^\bullet = x$	def. of $\bullet$
4. $FV(x) = \{x\}$	def. of FV
5. $RV(\Delta, x{:}^{\mathsf{r}} A) = RV(\Delta) \cup \{x\}$	def. of RV
6. $\{x\} \subseteq RV(\Delta) \cup \{x\}$	def. of $\subseteq$
7. $FV(M^\bullet) \subseteq RV(\Gamma)$	by 1, 3, 4, 6, 5, 2

Weak case

$$\left(\frac{\Delta \vdash B :^{\mathsf{c}} s \qquad \Delta \vdash M :^{\mathsf{r}} A}{\Delta, x{:}^{\tau} B \vdash M :^{\mathsf{r}} A}\right)$$

1. $\Gamma = \Delta, x{:}^{\tau} B$	hypothesis
2. $\Delta \vdash M :^{\mathsf{r}} A$	hypothesis
3. $FV(M^\bullet) \subseteq RV(\Delta)$	ind. hyp. on 2
4. $RV(\Delta) \subseteq RV(\Delta, x{:}^{\tau} B)$	def. of RV, $\subseteq$
5. $FV(M^\bullet) \subseteq RV(\Delta, x{:}^{\tau} B)$	by 3, 4

Π-Form case

$$\left(\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash B :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\tau} x{:}B.\, C :^{\mathsf{r}} s_3}\right)$$

1. $M = \Pi^{\tau} x{:}B.\, C$	hypothesis
2. $\Gamma \vdash B :^{\mathsf{r}} s_1$	hypothesis
3. $\Gamma, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2$	hypothesis
4. $FV(B^\bullet) \subseteq RV(\Gamma)$	ind. hyp. on 2
5. $FV(C^\bullet) \subseteq RV(\Gamma, x{:}^{\mathsf{r}} B)$	ind. hyp. on 3
6. $RV(\Gamma, x{:}^{\mathsf{r}} B) = RV(\Gamma) \cup \{x\}$	def. of RV
7. $FV(C^\bullet) \subseteq RV(\Gamma) \cup \{x\}$	by 5, 6
8. $FV(C^\bullet) - \{x\} \subseteq RV(\Gamma)$	by 7
9. $FV(B^\bullet) \cup (FV(C^\bullet) - \{x\}) \subseteq RV(\Gamma)$	by 4, 8
10. $\tau = \mathsf{r} \vee \tau = \mathsf{c}$	tautology

Step		Justification
	11. $\tau = \mathsf{r}$	assumption
	12. $(\Pi^{\mathsf{r}} x{:}B.\, C)^\bullet = \Pi x{:}B^\bullet.\, C^\bullet$	def. of $\bullet$
	13. $FV(\Pi x{:}B^\bullet.\, C^\bullet) =$ $FV(B^\bullet) \cup (FV(C^\bullet) - \{x\})$	def. of FV
	14. $FV(M^\bullet) \subseteq RV(\Gamma)$	by 1, 11, 12, 13, 9
	15. $\tau = \mathsf{c}$	assumption
	16. $(\Pi^{\mathsf{r}} x{:}B.\, C)^\bullet = \forall x{:}B^\bullet.\, C^\bullet$	def. of $\bullet$
	17. $FV(\forall x{:}B^\bullet.\, C^\bullet) =$ $FV(B^\bullet) \cup (FV(C^\bullet) - \{x\})$	def. of FV
	18. $FV(M^\bullet) \subseteq RV(\Gamma)$	by 1, 15, 16, 17, 9
19.	$FV(M^\bullet) \subseteq RV(\Gamma)$	$\vee$-elim, 10, 11–14, 15–18

Π-Intro case

$$\left(\frac{\Gamma \vdash \Pi^\tau x{:}A.\, B :^{\mathsf{c}} s \qquad \Gamma, x{:}^\tau A \vdash N :^{\mathsf{r}} B}{\Gamma \vdash \lambda^\tau x{:}A.\, N :^{\mathsf{r}} \Pi^\tau x{:}A.\, B}\right)$$

1.	$M = \lambda^\tau x{:}A.\, N$	hypothesis
2.	$\Gamma, x{:}^\tau A \vdash N :^{\mathsf{r}} B$	hypothesis
3.	$FV(N^\bullet) \subseteq RV(\Gamma, x{:}^\tau A)$	ind. hyp. on 2
4.	$\tau = \mathsf{r} \vee \tau = \mathsf{c}$	tautology
	5. $\tau = \mathsf{r}$	assumption
	6. $(\lambda^{\mathsf{r}} x{:}A.\, N)^\bullet = \lambda x.\, N^\bullet$	def. of $\bullet$
	7. $FV(\lambda x.\, N^\bullet) = FV(N^\bullet) - \{x\}$	def. of FV
	8. $RV(\Gamma, x{:}^{\mathsf{r}} A) = RV(\Gamma) \cup \{x\}$	def. of RV
	9. $FV(N^\bullet) \subseteq RV(\Gamma) \cup \{x\}$	by 3, 5, 8
	10. $FV(N^\bullet) - \{x\} \subseteq RV(\Gamma)$	by 9
	11. $FV(M^\bullet) \subseteq RV(\Gamma)$	by 1, 5, 6, 7, 10
	12. $\tau = \mathsf{c}$	assumption

Step		Justification
	13. $(\lambda^{\mathsf{c}} x{:}A.\, N)^\bullet = N^\bullet$	def. of $\bullet$
	14. $RV(\Gamma, x{:}^{\mathsf{c}} A) = RV(\Gamma)$	def. of RV
	15. $FV(M^\bullet) \subseteq RV(\Gamma)$	by 1, 12, 13, 3, 14
16.	$FV(M^\bullet) \subseteq RV(\Gamma)$	$\vee$-elim, 4, 5–11, 12–15

Π-Elim case

$$\left(\frac{\Gamma \vdash P :^{\mathsf{r}} \Pi^\tau x{:}B.\, C \qquad \Gamma \vdash N :^\tau B}{\Gamma \vdash P@^\tau N :^{\mathsf{r}} C[N/x]}\right)$$

1.	$M = P@^\tau N$	hypothesis
2.	$\Gamma \vdash P :^{\mathsf{r}} \Pi^\tau x{:}B.\, C$	hypothesis
3.	$\Gamma \vdash N :^\tau B$	hypothesis
4.	$FV(P^\bullet) \subseteq RV(\Gamma)$	ind. hyp. on 2
5.	$\tau = \mathsf{r} \vee \tau = \mathsf{c}$	tautology
	6. $\tau = \mathsf{r}$	assumption
	7. $(P@^{\mathsf{r}} N)^\bullet = P^\bullet\ N^\bullet$	def. of $\bullet$
	8. $FV(P^\bullet\ N^\bullet) = FV(P^\bullet) \cup FV(N^\bullet)$	def. of FV
	9. $FV(N^\bullet) \subseteq RV(\Gamma)$	by 6, ind. hyp. on 3
	10. $FV(P^\bullet) \cup FV(N^\bullet) \subseteq RV(\Gamma)$	by 4, 9
	11. $FV(M^\bullet) \subseteq RV(\Gamma)$	by 1, 6, 7, 8, 10
	12. $\tau = \mathsf{c}$	assumption
	13. $(P@^{\mathsf{c}} N)^\bullet = P^\bullet$	def. of $\bullet$
	14. $FV(M^\bullet) \subseteq RV(\Gamma)$	by 1, 12, 13, 4
15.	$FV(M^\bullet) \subseteq RV(\Gamma)$	$\vee$-elim, 5, 6–11, 12–14

Conv case

$$\left(\frac{\Gamma \vdash M :^{\mathsf{r}} B \qquad \Gamma \vdash A :^{\mathsf{c}} s \qquad B =_\beta A}{\Gamma \vdash M :^{\mathsf{r}} A}\right)$$

1.	$\Gamma \vdash M :^{\mathsf{r}} B$	assumption

Step	Justification
2. $FV(M^\bullet) \subseteq RV(\Gamma)$	ind. hyp. on 1
RESET case $\left(\dfrac{\Gamma^\circ \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\mathsf{c}} A}\right)$	
1. $\mathsf{r} = \mathsf{c}$	hypothesis
2. contradiction!	by 1

□

Lemma A.2.2 (Erasure Commutes with Substitution)

$$(M[N/x])^\bullet = M^\bullet[N^\bullet/x]$$

Proof. By induction on M.

Case	Step			Justification
case $M = x$				
	$(x[N/x])^\bullet$	$=$	$N^\bullet$	def. of subst.
		$=$	$x[N^\bullet/x]$	def. of subst.
		$=$	$x^\bullet[N^\bullet/x]$	def. of $\bullet$
case $M = y \neq x$				
	$(y[N/x])^\bullet$	$=$	$y^\bullet$	def. of subst.
		$=$	y	def. of $\bullet$
		$=$	$y[N^\bullet/x]$	def. of subst.
		$=$	$y^\bullet[N^\bullet/x]$	def. of $\bullet$
case $M = \lambda^{\mathsf{r}} y{:}A.\, M_0$				
	$((\lambda^{\mathsf{r}} y{:}A.\, M_0)[N/x])^\bullet$	$=$	$(\lambda^{\mathsf{r}} y{:}A[N/x].\, M_0[N/x])^\bullet$	def. of subst.
		$=$	$\lambda y.\, (M_0[N/x])^\bullet$	def. of $\bullet$
		$=$	$\lambda y.\, M_0^\bullet[N^\bullet/x]$	ind. hyp. on M_0

Case	Step		Justification
	$=$	$(\lambda y.\, M_0{}^\bullet)[N^\bullet/x]$	def. of subst.
	$=$	$(\lambda^{\mathsf{r}} y{:}A.\, M_0)^\bullet[N^\bullet/x]$	def. of $\bullet$
case $M = \lambda^{\mathsf{c}} y{:}A.\, M_0$			
$((\lambda^{\mathsf{c}} y{:}A.\, M_0)[N/x])^\bullet$	$=$	$(\lambda^{\mathsf{c}} y{:}A[N/x].\, M_0[N/x])^\bullet$	def. of subst.
	$=$	$(M_0[N/x])^\bullet$	def. of $\bullet$
	$=$	$M_0{}^\bullet[N^\bullet/x]$	ind. hyp. on M_0
	$=$	$(\lambda^{\mathsf{c}} y{:}A.\, M_0)^\bullet[N^\bullet/x]$	def. of $\bullet$
case $M = M_0 @^{\mathsf{r}} N_0$			
$((M_0 @^{\mathsf{r}} N_0)[N/x])^\bullet$	$=$	$(M_0[N/x] @^{\mathsf{r}} N_0[N/x])^\bullet$	def. of subst.
	$=$	$(M_0[N/x])^\bullet\ (N_0[N/x])^\bullet$	def. of $\bullet$
	$=$	$(M_0{}^\bullet[N^\bullet/x])\ (N_0{}^\bullet[N^\bullet/x])$	ind. hyp. on M_0
	$=$	$(M_0{}^\bullet\ N_0{}^\bullet)[N^\bullet/x]$	def. of subst.
	$=$	$(M_0 @^{\mathsf{r}} N_0)^\bullet[N^\bullet/x]$	def. of $\bullet$
case $M = M_0 @^{\mathsf{c}} N_0$			
$((M_0 @^{\mathsf{c}} N_0)[N/x])^\bullet$	$=$	$(M_0[N/x] @^{\mathsf{c}} N_0[N/x])^\bullet$	def. of subst.
	$=$	$(M_0[N/x])^\bullet$	def. of $\bullet$
	$=$	$M_0{}^\bullet[N^\bullet/x]$	ind. hyp. on M_0
	$=$	$(M_0 @^{\mathsf{c}} N_0)^\bullet[N^\bullet/x]$	def. of $\bullet$
case $M = \Pi^{\mathsf{r}} y{:}A.\, B$			
$((\Pi^{\mathsf{r}} y{:}A.\, B)[N/x])^\bullet$	$=$	$(\Pi^{\mathsf{r}} y{:}A[N/x].\, B[N/x])^\bullet$	def. of subst.
	$=$	$\Pi y{:}(A[N/x])^\bullet.\, (B[N/x])^\bullet$	def. of $\bullet$
	$=$	$\Pi y{:}A^\bullet[N^\bullet/x].\, B^\bullet[N^\bullet/x]$	ind. hyp. on A, B
	$=$	$(\Pi y{:}A^\bullet.\, B^\bullet)[N^\bullet/x]$	def. of subst.
	$=$	$(\Pi^{\mathsf{r}} y{:}A.\, B)^\bullet[N^\bullet/x]$	def. of $\bullet$
case $M = \Pi^{\mathsf{c}} y{:}A.\, B$			
$((\Pi^{\mathsf{c}} y{:}A.\, B)[N/x])^\bullet$	$=$	$(\Pi^{\mathsf{c}} y{:}A[N/x].\, B[N/x])^\bullet$	def. of subst.
	$=$	$\forall y{:}(A[N/x])^\bullet.\, (B[N/x])^\bullet$	def. of $\bullet$

Case	Step	Justification
	$= \forall y{:}A^\bullet[N^\bullet/x].\, B^\bullet[N^\bullet/x]$	ind. hyp. on A, B
	$= (\forall y{:}A^\bullet.\, B^\bullet)[N^\bullet/x]$	def. of subst.
	$= (\Pi^{\mathsf{c}} y{:}A.\, B)^\bullet[N^\bullet/x]$	def. of $\bullet$
case $M = s$		
$(s[N/x])^\bullet$	$= s^\bullet$	def. of subst.
	$= s$	def. of $\bullet$
	$= s[N^\bullet/x]$	def. of subst.
	$= s^\bullet[N^\bullet/x]$	def. of $\bullet$

□

Theorem A.2.3 (Erasure Respects Reduction)

$$\frac{\Gamma \vdash M :^\tau A \qquad M \to_\beta N}{M^\bullet \to_\beta N^\bullet \;\vee\; M^\bullet = N^\bullet}$$

Note. During this proof, it will be convenient to have a notation for $(\Pi^\tau x{:}A.\, B)^\bullet$ when τ is unknown We define $\Pi^{\bullet\tau} x{:}A.\, B$ as follows

$$\Pi^{\bullet\mathsf{r}} x{:}A.\, B = \Pi x{:}A.\, B \qquad\qquad \Pi^{\bullet\mathsf{c}} x{:}A.\, B = \forall x{:}A.\, B$$

so that $(\Pi^\tau x{:}A.\, B)^\bullet = \Pi^{\bullet\tau} x{:}A^\bullet.\, B^\bullet$.

Proof. By induction on the derivation of $\Gamma \vdash M :^\tau A$.

Step	Justification
AXIOM case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$	
1. $M \to_\beta N$	assumption
2. $M = s_1$	hypothesis

Step	Justification
3. contradiction!	by 1, 2

VAR case

$$\left(\frac{\Gamma \vdash A :^{\mathsf{c}} s}{\Gamma, x{:}^{\mathsf{r}} A \vdash x :^{\mathsf{r}} A}\right)$$

1. $M \to_\beta N$	assumption
2. $M = x$	hypothesis
3. contradiction!	by 1, 2

WEAK case

$$\left(\frac{\Gamma \vdash B :^{\mathsf{c}} s \qquad \Gamma \vdash M :^{\mathsf{r}} A}{\Gamma, x{:}^{\tau} B \vdash M :^{\mathsf{r}} A}\right)$$

1. $M \to_\beta N$	assumption
2. $\Gamma \vdash M :^{\mathsf{r}} A$	hypothesis
3. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	ind. hyp. on 2, 1

Π-FORM case

$$\left(\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash B :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\tau} x{:}B.\, C :^{\mathsf{r}} s_3}\right)$$

1. $M \to_\beta N$	assumption
2. $M = \Pi^{\tau} x{:}B.\, C$	hypothesis
3. $\Gamma \vdash B :^{\mathsf{r}} s_1$	hypothesis
4. $\Gamma, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2$	hypothesis
5. $M^\bullet = \Pi^{\bullet\tau} x{:}B^\bullet.\, C^\bullet$	by 2, def. of $\bullet$
6. $((\exists\, B')\ B \to_\beta B' \wedge N = \Pi^{\tau} x{:}B'.\, C) \vee$ $((\exists\, C')\ C \to_\beta C' \wedge N = \Pi^{\tau} x{:}B.\, C')$	by 1, 2, def. of $\to_\beta$
7. $B \to_\beta B'$	assumption (for some B')
8. $N = \Pi^{\tau} x{:}B'.\, C$	assumption
9. $N^\bullet = \Pi^{\bullet\tau} x{:}B'^\bullet.\, C^\bullet$	by 8, def. of $\bullet$

Step		Justification
10.	$B^\bullet \to_\beta B'^\bullet \vee B^\bullet = B'^\bullet$	ind. hyp. on 3, 7
11.	$B^\bullet \to_\beta B'^\bullet$	assumption
12.	$M^\bullet \to_\beta N^\bullet$	by 5, 8, 11
13.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 12
14.	$B^\bullet = B'^\bullet$	assumption
15.	$M^\bullet = N^\bullet$	by 5, 8, 14
16.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 15
17.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 10, 11–13, 14–16
18.	$C \to_\beta C'$	assumption (for some C')
19.	$N = \Pi^\tau x{:}B.\, C'$	assumption
20.	$N^\bullet = \Pi^{\bullet\tau} x{:}B^\bullet.\, C'^\bullet$	by 19, def. of $\bullet$
21.	$C^\bullet \to_\beta C'^\bullet \vee C^\bullet = C'^\bullet$	ind. hyp. on 4, 18
22.	$C^\bullet \to_\beta C'^\bullet$	assumption
23.	$M^\bullet \to_\beta N^\bullet$	by 5, 20, 22
24.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 23
25.	$C^\bullet = C'^\bullet$	assumption
26.	$M^\bullet = N^\bullet$	by 5, 20, 25
27.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 26
28.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 21, 22–24, 25–27
29.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 6, 7–17, 18–28

Π^{r}-Intro case

$$\left(\frac{\Gamma \vdash \Pi^{\mathsf{r}} x{:}A.\, B :^{\mathsf{c}} s \qquad \Gamma, x{:}^{\mathsf{r}} A \vdash P :^{\mathsf{r}} B}{\Gamma \vdash \lambda^{\mathsf{r}} x{:}A.\, P :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}A.\, B}\right)$$

1.	$M \to_\beta N$	assumption
2.	$M = \lambda^{\mathsf{r}} x{:}A.\, P$	hypothesis
3.	$\Gamma, x{:}^{\mathsf{r}} A \vdash P :^{\mathsf{r}} B$	hypothesis

Step		Justification
4.	$M^\bullet = \lambda x.\, P^\bullet$	by 2, def. of $\bullet$
5.	$((\exists\, A')\ A \to_\beta A' \wedge N = \lambda^{\mathsf{r}} x{:}A.\, P) \vee$ $((\exists\, P')\ P \to_\beta P' \wedge N = \lambda^{\mathsf{r}} x{:}A.\, P')$	by 1, 2, def. of $\to_\beta$
6.	$A \to_\beta A'$	assumption (for some A')
7.	$N = \lambda^{\mathsf{r}} x{:}A'.\, P$	assumption
8.	$N^\bullet = \lambda x.\, P^\bullet$	by 7, def. of $\bullet$
9.	$M^\bullet = N^\bullet$	by 4, 8
10.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 9
11.	$P \to_\beta P'$	assumption (for some P')
12.	$N = \lambda^{\mathsf{r}} x{:}A.\, P'$	assumption
13.	$N^\bullet = \lambda x.\, P'^\bullet$	by 12, def. of $\bullet$
14.	$P^\bullet \to_\beta P'^\bullet \vee P^\bullet = P'^\bullet$	ind. hyp. on 3, 11
15.	$P^\bullet \to_\beta P'^\bullet$	assumption
16.	$M^\bullet \to_\beta N^\bullet$	by 4, 13, 15
17.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 16
18.	$P^\bullet = P'^\bullet$	assumption
19.	$M^\bullet = N^\bullet$	by 4, 13, 18
20.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 19
21.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 14, 15–17, 18–20
22.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 5, 6–10, 11–21

Π^{c}-Intro case

$$\left(\frac{\Gamma \vdash \Pi^{\mathsf{c}} x{:}A.\, B :^{\mathsf{c}} s \qquad \Gamma, x{:}^{\mathsf{c}} A \vdash P :^{\mathsf{r}} B}{\Gamma \vdash \lambda^{\mathsf{c}} x{:}A.\, P :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}A.\, B}\right)$$

1.	$M \to_\beta N$	assumption
2.	$M = \lambda^{\mathsf{c}} x{:}A.\, P$	hypothesis
3.	$\Gamma, x{:}^{\mathsf{c}} A \vdash P :^{\mathsf{r}} B$	hypothesis

Step		Justification
4.	$M^\bullet = P^\bullet$	by 2, def. of $\bullet$
5.	$((\exists\, A')\ A \to_\beta A' \wedge N = \lambda^{\mathsf{c}} x{:}A'.\, P) \vee$ $((\exists\, P')\ P \to_\beta P' \wedge N = \lambda^{\mathsf{c}} x{:}A.\, P')$	by 1, 2, def. of $\to_\beta$
6.	$A \to_\beta A'$	assumption (for some A')
7.	$N = \lambda^{\mathsf{c}} x{:}A'.\, P$	assumption
8.	$N^\bullet = P^\bullet$	by 7, def. of $\bullet$
9.	$M^\bullet = N^\bullet$	by 4, 8
10.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 9
11.	$P \to_\beta P'$	assumption (for some P')
12.	$N = \lambda^{\mathsf{c}} x{:}A.\, P'$	assumption
13.	$N^\bullet = P'^\bullet$	by 12, def. of $\bullet$
14.	$P^\bullet \to_\beta P'^\bullet \vee P^\bullet = P'^\bullet$	ind. hyp. on 3, 11
15.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	by 4, 13, 14
16.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 5, 6–10, 11–15

Π^{r}-Elim case

$$\left(\frac{\Gamma \vdash P :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}B.\, C \qquad \Gamma \vdash Q :^{\mathsf{r}} B}{\Gamma \vdash P@^{\mathsf{r}} Q :^{\mathsf{r}} C[Q/x]}\right)$$

1.	$M \to_\beta N$	assumption
2.	$M = P@^{\mathsf{r}} Q$	hypothesis
3.	$\Gamma \vdash P :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}B.\, C$	hypothesis
4.	$\Gamma \vdash Q :^{\mathsf{r}} B$	hypothesis
5.	$M^\bullet = P^\bullet\ Q^\bullet$	by 3, def. of $\bullet$
6.	$((\exists\, P')\ P \to_\beta P' \wedge N = P'@^{\mathsf{r}} Q) \vee$ $((\exists\, Q')\ Q \to_\beta Q' \wedge N = P@^{\mathsf{r}} Q') \vee$ $((\exists\, A'\ M')\ P = \lambda^{\tau} x{:}A'.\, M'$ $\wedge\ N = M'[Q/x])$	by 1, 2, def. of $\to_\beta$

Step		Justification
7.	$P \to_\beta P'$	assumption (for some P')
8.	$N = P'@^{\mathsf{r}}Q$	assumption
9.	$N^\bullet = P'^\bullet\ Q^\bullet$	by 8, def. of $\bullet$
10.	$P^\bullet \to_\beta P'^\bullet \vee P^\bullet = P'^\bullet$	ind. hyp. on 3, 7
11.	$P^\bullet \to_\beta P'^\bullet$	assumption
12.	$M^\bullet \to_\beta N^\bullet$	by 5, 9, 11
13.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 12
14.	$P^\bullet = P'^\bullet$	assumption
15.	$M^\bullet = N^\bullet$	by 5, 9, 14
16.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 15
17.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 10, 11–13, 14–16
18.	$Q \to_\beta Q'$	assumption (for some Q')
19.	$N = P@^{\mathsf{r}}Q'$	assumption
20.	$N^\bullet = P^\bullet\ Q'^\bullet$	by 19, def. of $\bullet$
21.	$Q^\bullet \to_\beta Q'^\bullet \vee Q^\bullet = Q'^\bullet$	ind. hyp. on 4, 18
22.	$Q^\bullet \to_\beta Q'^\bullet$	assumption
23.	$M^\bullet \to_\beta N^\bullet$	by 5, 20, 22
24.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 23
25.	$Q^\bullet = Q'^\bullet$	assumption
26.	$M^\bullet = N^\bullet$	by 5, 20, 25
27.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 26
28.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 21, 22–24, 25–27
29.	$P = \lambda^\tau x{:}A'.\, M'$	assumption (for some A', M')
30.	$N = M'[Q/x]$	assumption
31.	$N^\bullet = M'^\bullet[Q^\bullet/x]$	by 30, Lemma A.2.2
32.	$\Gamma \vdash \lambda^\tau x{:}A'.\, M' :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}B.\, C$	by 29, 3
33.	$\tau = \mathsf{r}$	Corollary A.1.12 on 32

Step		Justification
	34. $P^\bullet = \lambda x.\, M'^\bullet$	by 33, 29, def. of $\bullet$
	35. $(\lambda x.\, M'^\bullet)\ Q^\bullet \to_\beta M'^\bullet[Q^\bullet/x]$	def. of $\to_\beta$
	36. $M^\bullet \to_\beta N^\bullet$	by 35, 31, 34, 5
	37. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 36
38.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 6, 7–17, 18–28, 29–36

Π^{c}-Elim case

$$\left(\frac{\Gamma \vdash P :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}B.\, C \qquad \Gamma \vdash Q :^{\mathsf{c}} B}{\Gamma \vdash P@^{\mathsf{c}}Q :^{\mathsf{r}} C[Q/x]}\right)$$

1.	$M \to_\beta N$	assumption
2.	$M = P@^{\mathsf{c}}Q$	hypothesis
3.	$\Gamma \vdash P :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}B.\, C$	hypothesis
4.	$\Gamma \vdash Q :^{\mathsf{c}} B$	hypothesis
5.	$M^\bullet = P^\bullet$	by 3, def. of $\bullet$
6.	$((\exists\, P')\ P \to_\beta P' \wedge N = P'@^{\mathsf{c}}Q) \vee$ $((\exists\, Q')\ Q \to_\beta Q' \wedge N = P@^{\mathsf{c}}Q') \vee$ $((\exists\, A'\ M')\ P = \lambda^\tau x{:}A'.\, M'$ $\wedge\ N = M'[Q/x])$	by 1, 2, def. of $\to_\beta$
	7. $P \to_\beta P'$	assumption (for some P')
	8. $N = P'@^{\mathsf{c}}Q$	assumption
	9. $N^\bullet = P'^\bullet$	by 8, def. of $\bullet$
	10. $P^\bullet \to_\beta P'^\bullet \vee P^\bullet = P'^\bullet$	ind. hyp. on 3, 7
	11. $P^\bullet \to_\beta P'^\bullet$	assumption
	12. $M^\bullet \to_\beta N^\bullet$	by 5, 9, 11
	13. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 12
	14. $P^\bullet = P'^\bullet$	assumption
	15. $M^\bullet = N^\bullet$	by 5, 9, 14

Step		Justification
	16. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 15
	17. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 10, 11–13, 14–16
	18. $Q \to_\beta Q'$	assumption (for some Q')
	19. $N = P@^{\mathsf{c}}Q'$	assumption
	20. $N^\bullet = P^\bullet$	by 19, def. of $\bullet$
	26. $M^\bullet = N^\bullet$	by 5, 20
	28. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 26
	29. $P = \lambda^\tau x{:}A'.\, M'$	assumption (for some A', M')
	30. $N = M'[Q/x]$	assumption
	31. $N^\bullet = M'^\bullet[Q^\bullet/x]$	by 30, Lemma A.2.2
	32. $\Gamma \vdash \lambda^\tau x{:}A'.\, M' :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}B.\, C$	by 29, 3
	33. $\tau = \mathsf{c}$	Corollary A.1.12 on 32
	34. $\Gamma, x{:}^{\mathsf{c}}B \vdash M' :^{\mathsf{r}} C$	Corollary A.1.12 on 32
	35. $P^\bullet = M'^\bullet$	by 33, 29, def. of $\bullet$
	36. $FV(M'^\bullet) \subseteq RV(\Gamma, x{:}^{\mathsf{c}}B)$	Lemma A.2.1, 34
	37. $x \notin RV(\Gamma, x{:}^{\mathsf{c}}B)$	def. of RV
	38. $x \notin FV(M'^\bullet)$	by 36, 37
	39. $M'^\bullet[Q^\bullet/x] = M'^\bullet$	by 38
	40. $M^\bullet = N^\bullet$	by 5, 35, 39, 31
	41. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-intro, 40
42.	$M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	$\vee$-elim, 6, 7–17, 18–28, 29–41

Conv case

$$\left(\frac{\Gamma \vdash M :^{\mathsf{r}} B \qquad \Gamma \vdash A :^{\mathsf{c}} s \qquad B =_\beta A}{\Gamma \vdash M :^{\mathsf{r}} A}\right)$$

1.	$M \to_\beta N$	assumption
2.	$\Gamma \vdash M :^{\mathsf{r}} B$	hypothesis

Step	Justification
3. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	ind. hyp. on 2, 1

RESET case

$$\left(\frac{\Gamma^\circ \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\mathsf{c}} A}\right)$$

1. $M \to_\beta N$	assumption
2. $\Gamma^\circ \vdash M :^{\mathsf{r}} A$	hypothesis
3. $M^\bullet \to_\beta N^\bullet \vee M^\bullet = N^\bullet$	ind. hyp. on 2, 1

□

Corollary A.2.4 (Erasure Respects Reductions)

$$\frac{\Gamma \vdash M :^{\tau} A \qquad M \to_\beta^* N}{M^\bullet \to_\beta^* N^\bullet}$$

Proof. Follows immediately from Theorem A.2.3 and Lemma A.1.13 by induction on the length of the reduction path for $M \to_\beta^* N$. □

Corollary A.2.5 (Erasure Respects β-Conversion)

$$\frac{\Gamma \vdash M :^{\tau_1} A \qquad \Delta \vdash N :^{\tau_2} B \qquad M =_\beta N}{M^\bullet =_\beta N^\bullet}$$

Proof. Since $M =_\beta N$, there is a term P to which both M, and N reduce, that is, $M \to_\beta^* P$ and $M \to_\beta^* N$. Since M and N are both well formed, we may apply Corollary A.2.4 to obtain $M^\bullet \to_\beta^* P^\bullet$ and $M^\bullet \to_\beta^* N^\bullet$. □

Theorem A.2.6 (Reflection Lemma)

$$\frac{\Gamma \vdash M :^{r} \Pi^{r} x{:}A.\, B \qquad M^\bullet = \lambda x.\, P^\bullet}{(\exists\, A', P') \quad M \to_\beta^* \lambda^{r} x{:}A'.\, P' \quad \wedge \quad P'^\bullet = P^\bullet}$$

Proof. We recount the discussion from Section 3.3.1 that proves this result at a high level of abstraction (a low level of detail). At present, I do not see how to formalize this proof any more without doing violence to the core argument.

The only way $M^\bullet$ can be $\lambda x.\, P^\bullet$ is if M consists of a subterm $\lambda^{\mathsf{r}} x{:}C.\, P$ nested under some (perhaps zero) "frames" of the form $\lambda^{\mathsf{c}} y{:}C.\, [\,]$ or $[\,]@^{\mathsf{c}} N$. Because the type of M is $\Pi^{\mathsf{r}} x{:}A.\, B$, we know the top-most (outer-most) frame cannot be a λ^{c}. Similarly, for typing reasons, the bottom-most (inner-most) frame cannot be a $@^{\mathsf{c}}$, because it would be applied to a λ^{r}. Therefore, if there are any frames at all on top of $\lambda^{\mathsf{r}} x{:}C.\, P$, then there are at least two, and at some point there is a λ^{c} frame just underneath a $@^{\mathsf{c}}$ one, forming a redex. If we reduce this redex, the rest of the frame structure remains in tact, and the number of frames decreases by two. We may repeat this process until no intermediate frames are left (formally, the proof is by strong induction on the length of the frame stack). Then $M \to_\beta^* \lambda^{\mathsf{r}} x{:}C[\theta].\, P[\theta]$ where θ is the sequence of substitutions effected by the sequence of reductions. Because θ is comprised solely of substitutions for λ^{c}-bound variables, the Variable Survival Lemma (A.2.1) tells us there will be no occurrences of these variables inside $P^\bullet$. Therefore $P[\theta]^\bullet = P^\bullet[\theta^\bullet] = P^\bullet$. We conclude by setting $A' = C[\theta]$ and $P' = P[\theta]$. □

Theorem A.2.7 (Erasure Reflects Reductions)

$$\frac{\Gamma \vdash M :^{\tau} A \qquad M^\bullet \to_\beta E}{(\exists N) \quad N^\bullet = E \quad \wedge \quad M \to_\beta^+ N}$$

Proof. By induction on the derivation of $\Gamma \vdash M :^{\tau} A$.

Step	**Justification**
AXIOM case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\vdash s_1 :^{\mathsf{r}} s_2}\right)$	

Step	**Justification**
1. $M^\bullet \rightarrow_\beta E$	assumption
2. $M = s_1$	hypothesis
3. ${s_1}^\bullet = s_1$	def. of $\bullet$
4. $s_1 \not\rightarrow_\beta$	def. of $\rightarrow_\beta$
5. contradiction!	by 1, 2, 3, 4

VAR case

$$\left(\frac{\Gamma \vdash A :^{\mathsf{c}} s}{\Gamma, x{:}^{\mathsf{r}} A \vdash x :^{\mathsf{r}} A}\right)$$

1. $M^\bullet \rightarrow_\beta E$	assumption
2. $M = x$	hypothesis
3. $x^\bullet = x$	def. of $\bullet$
4. $x \not\rightarrow_\beta$	def. of $\rightarrow_\beta$
5. contradiction!	by 1, 2, 3, 4

WEAK case

$$\left(\frac{\Gamma \vdash A :^{\mathsf{c}} s \qquad \Gamma \vdash M :^{\mathsf{r}} B}{\Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B}\right)$$

1. $M^\bullet \rightarrow_\beta E$	assumption
2. $\Gamma \vdash M :^{\mathsf{r}} B$	hypothesis
3. $(\exists\, N)\ N^\bullet = E \wedge M \rightarrow_\beta^+ N$	ind. hyp. on 2, 1

Π^{r}-FORM case

$$\left(\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\mathsf{r}} x{:}A.\, B :^{\mathsf{r}} s_3}\right)$$

1. $M^\bullet \rightarrow_\beta E$	assumption
2. $M = \Pi^{\mathsf{r}} x{:}A.\, B$	hypothesis
3. $\Gamma \vdash A :^{\mathsf{r}} s_1$	hypothesis
4. $\Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	hypothesis

Step		Justification
5.	$(\Pi^{\mathsf{r}} x{:}A.\, B)^\bullet = \Pi x{:}A^\bullet.\, B^\bullet$	def. of $\bullet$
6.	$\Pi x{:}A^\bullet.\, B^\bullet \to_\beta E$	by 1, 2, 5
7.	$((\exists\, A')\ A^\bullet \to_\beta A' \wedge E = \Pi x{:}A'.\, B^\bullet) \vee$	
	$((\exists\, B')\ B^\bullet \to_\beta B' \wedge E = \Pi x{:}A^\bullet.\, B')$	by 6
	8. $A^\bullet \to_\beta A'$	assumption
	9. $E = \Pi x{:}A'.\, B^\bullet$	(for some A')
	10. $P^\bullet = A'$	ind. hyp. on 3, 8
	11. $A \to_\beta^+ P$	(for some P)
	12. let $N = \Pi^{\mathsf{r}} x{:}P.\, B$	definition
	13. $N^\bullet = \Pi x{:}P^\bullet.\, B^\bullet$	def. of $\bullet$, 12
	14. $N^\bullet = E$	by 13, 9, 10
	15. $M \to_\beta^+ N$	by 2, 12, 11
	16. $(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 12, 14, 15
	17. $B^\bullet \to_\beta B'$	assumption
	18. $E = \Pi x{:}A^\bullet.\, B'$	(for some B')
	19. $P^\bullet = B'$	ind. hyp. on 4, 17
	20. $B \to_\beta^+ P$	(for some P)
	21. let $N = \Pi^{\mathsf{r}} x{:}A.\, P$	definition
	22. $N^\bullet = \Pi x{:}A^\bullet.\, P^\bullet$	def. of $\bullet$, 21
	23. $N^\bullet = E$	by 22, 18, 19
	24. $M \to_\beta^+ N$	by 2, 21, 20
	25. $(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 21, 23, 24
26.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	$\vee$-elim, 7, 8–16, 17–25

Π^{c}-Form case

$$\left(\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\mathsf{c}} x{:}A.\, B :^{\mathsf{r}} s_3}\right)$$

Step		Justification
1.	$M^\bullet \to_\beta E$	assumption
2.	$M = \Pi^{\mathsf{c}} x{:}A.\, B$	hypothesis
3.	$\Gamma \vdash A :^{\mathsf{r}} s_1$	hypothesis
4.	$\Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	hypothesis
5.	$(\Pi^{\mathsf{c}} x{:}A.\, B)^\bullet = \forall x{:}A^\bullet.\, B^\bullet$	def. of $\bullet$
6.	$\forall x{:}A^\bullet.\, B^\bullet \to_\beta E$	by 1, 2, 5
7.	$((\exists\, A')\ A^\bullet \to_\beta A' \wedge E = \forall x{:}A'.\, B^\bullet) \vee$	
	$((\exists\, B')\ B^\bullet \to_\beta B' \wedge E = \forall x{:}A^\bullet.\, B')$	by 6
8.	$A^\bullet \to_\beta A'$	assumption
9.	$E = \forall x{:}A'.\, B^\bullet$	(for some A')
10.	$P^\bullet = A'$	ind. hyp. on 3, 8
11.	$A \to_\beta^+ P$	(for some P)
12.	let $N = \Pi^{\mathsf{c}} x{:}P.\, B$	definition
13.	$N^\bullet = \forall x{:}P^\bullet.\, B^\bullet$	def. of $\bullet$, 12
14.	$N^\bullet = E$	by 13, 9, 10
15.	$M \to_\beta^+ N$	by 2, 12, 11
16.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 12, 14, 15
17.	$B^\bullet \to_\beta B'$	assumption
18.	$E = \forall x{:}A^\bullet.\, B'$	(for some B')
19.	$P^\bullet = B'$	ind. hyp. on 4, 17
20.	$B \to_\beta^+ P$	(for some P)
21.	let $N = \Pi^{\mathsf{c}} x{:}A.\, P$	definition
22.	$N^\bullet = \forall x{:}A^\bullet.\, P^\bullet$	def. of $\bullet$, 21
23.	$N^\bullet = E$	by 22, 18, 19
24.	$M \to_\beta^+ N$	by 2, 21, 20
25.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 21, 23, 24

Step	Justification
26. $(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	$\vee$-elim, 7, 8–16, 17–25

Π^{r}-Intro case

$$\left(\frac{\Gamma \vdash \Pi^{\mathsf{r}} x{:}A.\, B :^{\mathsf{c}} s \qquad \Gamma, x{:}^{\mathsf{r}} A \vdash P :^{\mathsf{r}} B}{\Gamma \vdash \lambda^{\mathsf{r}} x{:}A.\, P :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}A.\, B}\right)$$

1.	$M^\bullet \to_\beta E$	assumption
2.	$M = \lambda^{\mathsf{r}} x{:}A.\, P$	hypothesis
3.	$\Gamma \vdash \Pi^{\mathsf{r}} x{:}A.\, B :^{\mathsf{c}} s$	hypothesis
4.	$\Gamma, x{:}^{\mathsf{r}} A \vdash P :^{\mathsf{r}} B$	hypothesis
5.	$M^\bullet = \lambda x.\, P^\bullet$	by 2, def. of $\bullet$
6.	$P^\bullet \to_\beta F$	def. of $\to_\beta$, 1, 5
7.	$E = \lambda x.\, F$	(for some F)
8.	$P'^\bullet = F$	ind. hyp. on 4, 6
9.	$P \to_\beta^+ P'$	(for some P')
10.	let $N = \lambda^{\mathsf{r}} x{:}A.\, P'$	definition
11.	$N^\bullet = \lambda x.\, P'^\bullet$	def. of $\bullet$
12.	$N^\bullet = E$	by 11, 8, 7
13.	$M \to_\beta^+ N$	by 2, 10, 9
14.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 10, 12, 13

Π^{c}-Intro case

$$\left(\frac{\Gamma \vdash \Pi^{\mathsf{c}} x{:}A.\, B :^{\mathsf{c}} s \qquad \Gamma, x{:}^{\mathsf{c}} A \vdash P :^{\mathsf{r}} B}{\Gamma \vdash \lambda^{\mathsf{c}} x{:}A.\, P :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}A.\, B}\right)$$

1.	$M^\bullet \to_\beta E$	assumption
2.	$M = \lambda^{\mathsf{c}} x{:}A.\, P$	hypothesis
3.	$\Gamma \vdash \Pi^{\mathsf{c}} x{:}A.\, B :^{\mathsf{c}} s$	hypothesis
4.	$\Gamma, x{:}^{\mathsf{c}} A \vdash P :^{\mathsf{r}} B$	hypothesis
5.	$M^\bullet = P^\bullet$	by 2, def. of $\bullet$

Step		Justification
6.	$P^\bullet \to_\beta E$	by 1, 5
7.	$P'^\bullet = E$	ind. hyp. on 4, 6
8.	$P \to_\beta^+ P'$	(for some P')
9.	let $N = \lambda^{\mathsf{c}} x{:}A.\, P'$	definition
10.	$N^\bullet = P'^\bullet$	def. of $\bullet$
11.	$N^\bullet = E$	by 10, 7
12.	$M \to_\beta^+ N$	by 2, 9, 8
13.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 9, 11, 12

Π^{r}-Elim case

$$\left(\frac{\Gamma \vdash P :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}A.\, B \qquad \Gamma \vdash Q :^{\mathsf{r}} A}{\Gamma \vdash P @^{\mathsf{r}} Q :^{\mathsf{r}} B[Q/x]}\right)$$

1.	$M^\bullet \to_\beta E$	assumption
2.	$M = P @^{\mathsf{r}} Q$	hypothesis
3.	$\Gamma \vdash P :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}A.\, B$	hypothesis
4.	$\Gamma \vdash Q :^{\mathsf{r}} A$	hypothesis
5.	$M^\bullet = P^\bullet\ Q^\bullet$	def. of $\bullet$, 2
6.	$((\exists\, F)\ P^\bullet \to_\beta F \wedge E = F\ Q^\bullet) \vee$	
	$((\exists\, F)\ Q^\bullet \to_\beta F \wedge E = P^\bullet\ F) \vee$	
	$((\exists\, R)\ P^\bullet = \lambda x.\, R^\bullet \wedge E = R^\bullet[Q^\bullet/x])$	def. of $\to_\beta$, 1, 5
7.	$P^\bullet \to_\beta F$	assumption
8.	$E = F\ Q^\bullet$	(for some F)
9.	$P'^\bullet = F$	ind. hyp. on 3, 7
10.	$P \to_\beta^+ P'$	(for some P')
11.	let $N = P' @^{\mathsf{r}} Q$	definition
12.	$N^\bullet = P'^\bullet\ Q^\bullet$	def. of $\bullet$, 11
13.	$N^\bullet = E$	by 12, 9, 8

Step		Justification
14.	$M \to_\beta^+ N$	by 2, 11, 10
15.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 11, 13, 14
16.	$Q^\bullet \to_\beta F$	assumption
17.	$E = P^\bullet\ F$	(for some F)
18.	$Q'^\bullet = F$	ind. hyp. on 4, 16
19.	$Q \to_\beta^+ Q'$	(for some Q')
20.	let $N = P@^{\mathsf{r}}Q'$	definition
21.	$N^\bullet = P^\bullet\ Q'^\bullet$	def. of $\bullet$, 20
22.	$N^\bullet = E$	by 21, 18, 17
23.	$M \to_\beta^+ N$	by 2, 20, 19
24.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 20, 22, 23
25.	$P^\bullet = \lambda x.\, R^\bullet$	assumption
26.	$E = R^\bullet[Q^\bullet/x]$	(for some R)
27.	$P \to_\beta^* \lambda^{\mathsf{r}} x{:}A'.\, R'$	Lemma A.2.6, 3, 25
28.	$R'^\bullet = R^\bullet$	(for some A', R')
29.	let $N = R'[Q/x]$	definition
30.	$N^\bullet = (R'[Q/x])^\bullet$	def. of $\bullet$, 29
31.	$(R'[Q/x])^\bullet = R'^\bullet[Q^\bullet/x]$	Lemma A.2.2
32.	$N^\bullet = E$	by 30, 31, 28, 26
33.	$(\lambda^{\mathsf{r}} x{:}A'.\, R')@^{\mathsf{r}}Q \to_\beta R'[Q/x]$	def. of $\to_\beta$
34.	$M \to_\beta^+ N$	by 2, 27, 33, 29
35.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 29, 32, 34
36.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	$\vee$-elim, 6, 7–15, 16–24, 25–35

Π^{c}-Elim case

$$\left(\frac{\Gamma \vdash P :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}A.\, B \qquad \Gamma \vdash Q :^{\mathsf{c}} A}{\Gamma \vdash P@^{\mathsf{c}}Q :^{\mathsf{r}} B[Q/x]}\right)$$

Step		Justification
1.	$M^\bullet \to_\beta E$	assumption
2.	$M = P@^{\mathsf{c}}Q$	hypothesis
3.	$\Gamma \vdash P :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}A.\, B$	hypothesis
4.	$\Gamma \vdash Q :^{\mathsf{c}} A$	hypothesis
5.	$M^\bullet = P^\bullet$	def. of $\bullet$, 2
6.	$P^\bullet \to_\beta E$	by 1, 5
7.	$P'^\bullet = E$	ind. hyp. of 3, 6 (for some P')
8.	$P \to_\beta^+ P'$	
9.	let $N = P'@^{\mathsf{c}}Q$	definition
10.	$N^\bullet = P'^\bullet$	def. of $\bullet$, 9
11.	$N^\bullet = E$	by 10, 7
12.	$M \to_\beta^+ N$	by 2, 9, 8
13.	$(\exists\, N)\ N^\bullet = E \wedge M \to_\beta^+ N$	by 9, 11, 12
Conv and Reset cases are similar to the Weak case		

□

Lemma A.2.8 (Erasure Annihilates Context Reset)

$$\Gamma^{\circ\bullet} = \Gamma^\bullet$$

Proof. By induction on the structure of Γ.

Case	Step			Justification
ε	$\varepsilon^{\circ\bullet}$	$=$	$\varepsilon^\bullet$	def. of $\circ$
$\Gamma, x{:}^\tau A$	$(\Gamma, x{:}^\tau A)^{\circ\bullet}$	$=$	$(\Gamma^\circ, x{:}^{\mathsf{r}} A)^\bullet$	def. of $\circ$
		$=$	$\Gamma^{\circ\bullet}, x{:}A$	def. of $\bullet$
		$=$	$\Gamma^\bullet, x{:}A$	ind. hyp. on Γ
		$=$	$(\Gamma, x{:}^\tau A)^\bullet$	def. of $\bullet$

□

Theorem A.2.9 (Erasure Respects Types)

$$\frac{\Gamma \vdash M :^{\tau} A}{\Gamma^{\bullet} \vdash M^{\bullet} : A^{\bullet}}$$

Note. In the following proof, we often apply the induction hypothesis to a judgment of the form $\Gamma \vdash A :^{\tau} s$. In this case, the conclusion is equivalent to $\Gamma^{\bullet} \vdash A^{\bullet} : s$ since $s^{\bullet} = s$. For this reason, we implicitly consider the judgments $\Gamma^{\bullet} \vdash A^{\bullet} : s^{\bullet}$ and $\Gamma^{\bullet} \vdash A^{\bullet} : s$ to be equivalent in the following proof.

Proof. By induction on the derivation of $\Gamma \vdash M :^{\tau} A$.

Step	**Justification**
AXIOM case $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\varepsilon \vdash s_1 :^{\mathsf{r}} s_2}\right)$	
1. $(s_1, s_2) \in \mathcal{A}$	hypothesis
2. $\varepsilon^{\bullet} = \varepsilon$	def. of $\bullet$
3. ${s_1}^{\bullet} = s_1$	def. of $\bullet$
4. $\vdash s_1 : s_2$	AXIOM, 1
5. $\varepsilon^{\bullet} \vdash {s_1}^{\bullet} : s_2$	by 2, 3, 4
VAR case $\left(\dfrac{\Gamma \vdash A :^{\mathsf{c}} s}{\Gamma, x{:}^{\mathsf{r}}A \vdash x :^{\mathsf{r}} A}\right)$	
1. $\Gamma \vdash A :^{\mathsf{c}} s$	hypothesis
2. $\Gamma^{\bullet} \vdash A^{\bullet} : s$	ind. hyp. on
3. $\Gamma^{\bullet}, x{:}A^{\bullet} \vdash x : A^{\bullet}$	VAR, 2
4. $(\Gamma, x{:}^{\mathsf{r}}A)^{\bullet} = \Gamma^{\bullet}, x{:}A^{\bullet}$	def. of $\bullet$
5. $x^{\bullet} = x$	def. of $\bullet$
6. $(\Gamma, x{:}^{\mathsf{r}}A)^{\bullet} \vdash x^{\bullet} : A^{\bullet}$	by 3, 4, 5

Step	Justification
WEAK case	
$\left(\dfrac{\Gamma \vdash A :^{\mathsf{c}} s \qquad \Gamma \vdash M :^{\mathsf{r}} B}{\Gamma, x{:}^{\tau} A \vdash M :^{\mathsf{r}} B}\right)$	
1. $\Gamma \vdash A :^{\mathsf{c}} s$	hypothesis
2. $\Gamma \vdash M :^{\mathsf{r}} B$	hypothesis
3. $\Gamma^{\bullet} \vdash A^{\bullet} : s$	ind. hyp. on 1
4. $\Gamma^{\bullet} \vdash M^{\bullet} : B^{\bullet}$	ind. hyp. on 2
5. $\Gamma^{\bullet}, x{:}A^{\bullet} \vdash M^{\bullet} : B^{\bullet}$	WEAK, 3, 4
6. $(\Gamma, x{:}^{\tau} A)^{\bullet} = \Gamma^{\bullet}, x{:}A^{\bullet}$	def. of $\bullet$
7. $(\Gamma, x{:}^{\tau} A)^{\bullet} \vdash M^{\bullet} : B^{\bullet}$	by 5, 6
Π-FORM case	
$\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A :^{\mathsf{r}} s_1 \qquad \Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2}{\Gamma \vdash \Pi^{\tau} x{:}A.\, B :^{\mathsf{r}} s_3}\right)$	
1. $(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
2. $\Gamma \vdash A :^{\mathsf{r}} s_1$	hypothesis
3. $\Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	hypothesis
4. $\Gamma^{\bullet} \vdash A^{\bullet} : s_1$	ind. hyp. on 2
5. $(\Gamma, x{:}^{\mathsf{r}} A)^{\bullet} \vdash B^{\bullet} : s_2$	ind. hyp. on 3
6. $(\Gamma, x{:}^{\mathsf{r}} A)^{\bullet} = \Gamma^{\bullet}, x{:}A^{\bullet}$	def. of $\bullet$
7. $\Gamma^{\bullet}, x{:}A^{\bullet} \vdash B^{\bullet} : s_2$	by 5, 6
8. $\tau = \mathsf{r} \ \vee \ \tau = \mathsf{c}$	tautology
9. $\tau = \mathsf{r}$	assumption
10. $\Gamma^{\bullet} \vdash \Pi x{:}A^{\bullet}.\, B^{\bullet} : s_3$	Π-FORM, 1, 4, 7
11. $(\Pi^{\tau} x{:}A.\, B)^{\bullet} = \Pi x{:}A^{\bullet}.\, B^{\bullet}$	by 9, def. of $\bullet$
12. $\Gamma^{\bullet} \vdash (\Pi^{\tau} x{:}A.\, B)^{\bullet} : s_3$	by 10, 11
13. $\tau = \mathsf{c}$	assumption

Step		Justification
14.	$\Gamma^\bullet \vdash \forall x{:}A^\bullet.\, B^\bullet : s_3$	$\forall$-Form, 1, 4, 7
15.	$(\Pi^\tau x{:}A.\, B)^\bullet = \forall x{:}A^\bullet.\, B^\bullet$	by 13, def. of $\bullet$
16.	$\Gamma^\bullet \vdash (\Pi^\tau x{:}A.\, B)^\bullet : s_3$	by 14, 15
17.	$\Gamma^\bullet \vdash (\Pi^\tau x{:}A.\, B)^\bullet : s_3$	$\vee$-elim 8, 9–12, 13–16

Π-Intro case

$$\left(\frac{\Gamma \vdash \Pi^\tau x{:}A.\, B :^{\mathsf{c}} s \qquad \Gamma, x{:}^\tau A \vdash M :^{\mathsf{r}} B}{\Gamma \vdash \lambda^\tau x{:}A.\, M :^{\mathsf{r}} \Pi^\tau x{:}A.\, B}\right)$$

1.	$\Gamma \vdash \Pi^\tau x{:}A.\, B :^{\mathsf{c}} s$	hypothesis
2.	$\Gamma, x{:}^\tau A \vdash M :^{\mathsf{r}} B$	hypothesis
3.	$\Gamma^\bullet \vdash (\Pi^\tau x{:}A.\, B)^\bullet : s$	ind. hyp. on 1
4.	$(\Gamma, x{:}^\tau A)^\bullet \vdash M^\bullet : B^\bullet$	ind. hyp. on 2
5.	$(\Gamma, x{:}^\tau A)^\bullet = \Gamma^\bullet, x{:}A^\bullet$	def. of $\bullet$
6.	$\Gamma^\bullet, x{:}A^\bullet \vdash M^\bullet : B^\bullet$	by 4, 5
7.	$\tau = \mathsf{r} \ \vee \ \tau = \mathsf{c}$	tautology
8.	$\tau = \mathsf{r}$	assumption
9.	$(\Pi^\tau x{:}A.\, B)^\bullet = \Pi x{:}A^\bullet.\, B^\bullet$	by 8, def. of $\bullet$
10.	$\Gamma^\bullet \vdash \Pi x{:}A^\bullet.\, B^\bullet : s$	by 3, 9
11.	$\Gamma^\bullet \vdash \lambda x.\, M^\bullet : \Pi x{:}A^\bullet.\, B^\bullet$	Π-Intro, 10, 6
12.	$(\lambda^\tau x{:}A.\, M)^\bullet = \lambda x.\, M^\bullet$	by 8, def. of $\bullet$
13.	$\Gamma^\bullet \vdash (\lambda^\tau x{:}A.\, M)^\bullet : (\Pi^\tau x{:}A.\, B)^\bullet$	by 11, 12, 9
14.	$\tau = \mathsf{c}$	assumption
15.	$(\Pi^\tau x{:}A.\, B)^\bullet = \forall x{:}A^\bullet.\, B^\bullet$	by 8, def. of $\bullet$
16.	$\Gamma^\bullet \vdash \forall x{:}A^\bullet.\, B^\bullet : s$	by 3, 15
17.	$\Gamma, x{:}^{\mathsf{c}} A \vdash M :^{\mathsf{r}} B$	by 2, 14
18.	$FV(M^\bullet) \subseteq RV(\Gamma, x{:}^{\mathsf{c}} A)$	Lemma A.2.1, 17
19.	$x \notin RV(\Gamma, x{:}^{\mathsf{c}} A)$	def. of RV

Step		Justification
	20. $x \notin FV(M^\bullet)$	by 18, 19
	21. $\Gamma^\bullet \vdash M^\bullet : \forall x{:}A^\bullet.\, B^\bullet$	$\forall$-Intro, 16, 6, 20
	22. $(\lambda^\tau x{:}A.\, M)^\bullet = M^\bullet$	by 14, def. of $\bullet$
	23. $\Gamma^\bullet \vdash (\lambda^\tau x{:}A.\, M)^\bullet : (\Pi^\tau x{:}A.\, B)^\bullet$	by 21, 22, 15
24.	$\Gamma^\bullet \vdash (\lambda^\tau x{:}A.\, M)^\bullet : (\Pi^\tau x{:}A.\, B)^\bullet$	$\vee$-elim, 7, 8–13, 14–23

Π-Elim case

$$\left(\frac{\Gamma \vdash M :^{\mathsf{r}} \Pi^\tau x{:}A.\, B \qquad \Gamma \vdash N :^\tau A}{\Gamma \vdash M@^\tau N :^{\mathsf{r}} B[N/x]}\right)$$

1.	$\Gamma \vdash M :^{\mathsf{r}} \Pi^\tau x{:}A.\, B$	hypothesis
2.	$\Gamma \vdash N :^\tau A$	hypothesis
3.	$\Gamma^\bullet \vdash M^\bullet : (\Pi^\tau x{:}A.\, B)^\bullet$	ind. hyp. on 1
4.	$\Gamma^\bullet \vdash N^\bullet : A^\bullet$	ind. hyp. on 2
5.	$(B[N/x])^\bullet = B^\bullet[N^\bullet/x]$	Lemma A.2.2
6.	$\tau = \mathsf{r} \ \vee \ \tau = \mathsf{c}$	tautology
	7. $\tau = \mathsf{r}$	assumption
	8. $(\Pi^\tau x{:}A.\, B)^\bullet = \Pi x{:}A^\bullet.\, B^\bullet$	by 7, def. of $\bullet$
	9. $(M@^\tau N)^\bullet = M^\bullet\ N^\bullet$	by 7, def. of $\bullet$
	10. $\Gamma^\bullet \vdash M^\bullet : \Pi x{:}A^\bullet.\, B^\bullet$	by 3, 8
	11. $\Gamma^\bullet \vdash M^\bullet\ N^\bullet : B^\bullet[N^\bullet/x]$	Π-Elim, 10, 4
	12. $\Gamma^\bullet \vdash (M@^\tau N)^\bullet : (B[N/x])^\bullet$	by 11, 9, 5
	13. $\tau = \mathsf{c}$	assumption
	14. $(\Pi^\tau x{:}A.\, B)^\bullet = \forall x{:}A^\bullet.\, B^\bullet$	by 11, def. of $\bullet$
	15. $(M@^\tau N)^\bullet = M^\bullet$	by 13, def. of $\bullet$
	16. $\Gamma^\bullet \vdash M^\bullet : \forall x{:}A^\bullet.\, B^\bullet$	by 3, 14
	17. $\Gamma^\bullet \vdash M^\bullet : B^\bullet[N^\bullet/x]$	$\forall$-Intro, 16, 4
	18. $\Gamma^\bullet \vdash (M@^\tau N)^\bullet : (B[N/x])^\bullet$	by 17, 15, 5

Step	Justification
19. $\Gamma^\bullet \vdash (M@^\tau N)^\bullet : (B[N/x])^\bullet$	$\vee$-elim, 6, 7–12, 13–18
Conv case	
$\left(\dfrac{\Gamma \vdash M :^{\mathsf{r}} A \qquad \Gamma \vdash B :^{\mathsf{c}} s \qquad A =_\beta B}{\Gamma \vdash M :^{\mathsf{r}} B}\right)$	
1. $\Gamma \vdash M :^{\mathsf{r}} A$	hypothesis
2. $\Gamma \vdash B :^{\mathsf{c}} s$	hypothesis
3. $A =_\beta B$	hypothesis
4. $\Gamma^\bullet \vdash M^\bullet : A^\bullet$	ind. hyp. on 1
5. $\Gamma^\bullet \vdash B^\bullet : s$	ind. hyp. on 2
6. $A = s' \ \vee \ \Gamma \vdash A :^{\mathsf{c}} s'$	Lemma A.1.11, 1
7. $A = s'$	assumption
8. $B \to_\beta^* s'$	by 7, 3
9. $B^\bullet \to_\beta^* s'^\bullet$	Corollary A.2.4, 2, 8
10. $A^\bullet =_\beta B^\bullet$	by 7, 9
11. $\Gamma \vdash A :^{\mathsf{c}} s'$	assumption
12. $A^\bullet =_\beta B^\bullet$	Corollary A.2.5, 11, 2, 3
13. $A^\bullet =_\beta B^\bullet$	$\vee$-elim, 6, 7–10, 11–12
14. $\Gamma^\bullet \vdash M^\bullet : B^\bullet$	Conv, 4, 5, 13
Reset case	
$\left(\dfrac{\Gamma^\circ \vdash M :^{\mathsf{r}} A}{\Gamma \vdash M :^{\mathsf{c}} A}\right)$	
1. $\Gamma^\circ \vdash M :^{\mathsf{r}} A$	hypothesis
2. $\Gamma^{\circ\bullet} \vdash M^\bullet : A^\bullet$	ind. hyp. on 1
3. $\Gamma^{\circ\bullet} = \Gamma^\bullet$	Lemma A.2.8
4. $\Gamma^\bullet \vdash M^\bullet : A^\bullet$	by 2, 3

□

A.3 IMPLEMENTATION OF ERASURE CONTEXTS

Typing Context Operations are Well-Defined on $\cong$-Equivalence Classes of Cleverly Represented Typing Contexts

Lemma A.3.1 *If* $\Gamma \cong \Delta$ *then* $\Gamma, x{:}^{\tau} A \cong \Delta, x{:}^{\tau} A$.

Proof. Let $\Gamma = [\![\hat{\Gamma}]\!]_i$ and $\Delta = [\![\hat{\Delta}]\!]_j$. The assumption $\Gamma \cong \Delta$, is then logically equivalent to $[\![\hat{\Gamma}]\!]_i \cong [\![\hat{\Delta}]\!]_j$ and $[\![\hat{\Gamma}]\!]_i^\flat = [\![\hat{\Delta}]\!]_j^\flat$. We use this final form to prove the conjecture.

We want to prove $\Gamma, x{:}^{\tau} A \cong \Delta, x{:}^{\tau} A$. We proceed by handling the two cases $\tau = \mathsf{c}$ and $\tau = \mathsf{r}$ separately. Both of the following columns consist of a progression of logically equivalent equations starting with the equation we seek to prove and finishing with an equation that follows immediately from $[\![\hat{\Gamma}]\!]_i^\flat = [\![\hat{\Delta}]\!]_j^\flat$.

$$\begin{array}{rcl}
[\![\hat{\Gamma}]\!]_i, x{:}^{\mathsf{c}} A & \cong & [\![\hat{\Delta}]\!]_j, x{:}^{\mathsf{c}} A \\
[\![\hat{\Gamma}, x{:}^{i+1} A]\!]_i & \cong & [\![\hat{\Delta}, x{:}^{j+1} A]\!]_j \\
[\![\hat{\Gamma}, x{:}^{i+1} A]\!]_i^\flat & = & [\![\hat{\Delta}, x{:}^{j+1} A]\!]_j^\flat \\
[\![\hat{\Gamma}]\!]_i^\flat, x{:}^{\mathsf{c}} A & = & [\![\hat{\Delta}]\!]_j^\flat, x{:}^{\mathsf{c}} A
\end{array}
\qquad
\begin{array}{rcl}
[\![\hat{\Gamma}]\!]_i, x{:}^{\mathsf{r}} A & \cong & [\![\hat{\Delta}]\!]_j, x{:}^{\mathsf{r}} A \\
[\![\hat{\Gamma}, x{:}^{i} A]\!]_i & \cong & [\![\hat{\Delta}, x{:}^{j} A]\!]_j \\
[\![\hat{\Gamma}, x{:}^{i} A]\!]_i^\flat & = & [\![\hat{\Delta}, x{:}^{j} A]\!]_j^\flat \\
[\![\hat{\Gamma}]\!]_i^\flat, x{:}^{\mathsf{r}} A & = & [\![\hat{\Delta}]\!]_j^\flat, x{:}^{\mathsf{r}} A
\end{array}$$

□

Lemma A.3.2 *If* $\Gamma \cong \Delta$ *then* $\Gamma^\circ \cong \Delta^\circ$

Proof. Let $\Gamma = [\![\hat{\Gamma}]\!]_i$ and $\Delta = [\![\hat{\Delta}]\!]_j$. After unfolding some definitions, we see that our goal is really to prove $[\![\hat{\Gamma}]\!]_{i+1}^\flat = [\![\hat{\Delta}]\!]_{j+1}^\flat$ given the assumption $[\![\hat{\Gamma}]\!]_i^\flat = [\![\hat{\Delta}]\!]_j^\flat$. We proceed by induction on the length of $[\![\hat{\Gamma}]\!]_i^\flat$ ($=[\![\hat{\Delta}]\!]_j^\flat$). In each case, we reason backwards from the goal to some obviously true statement.

In the case that $[\![\hat{\Gamma}]\!]_i^\flat = [\![\hat{\Delta}]\!]_j^\flat = \varepsilon$, both $\hat{\Gamma}$ and $\hat{\Delta}$ must equal $\hat{\varepsilon}$. In this case, the reasoning proceeds as follows:

$$\begin{array}{rcl}
[\![\hat{\varepsilon}]\!]_{i+1}^\flat & = & [\![\hat{\varepsilon}]\!]_{j+1}^\flat \\
\varepsilon & = & \varepsilon
\end{array}$$

In the case that $[\![\hat{\Gamma}]\!]^\flat_i = [\![\hat{\Delta}]\!]^\flat_j \neq \varepsilon$, then there must be some $\hat{\Gamma}'$, x, k, A, $\hat{\Delta}'$, y, h, and B such that $\hat{\Gamma} = \hat{\Gamma}', x{:}^k A$ and $\hat{\Delta} = \hat{\Delta}', y{:}^h B$. Since $[\![\hat{\Gamma}]\!]^\flat_i = [\![\hat{\Delta}]\!]^\flat_j$, it must be that $x = y$ and $A = B$ and $[\![\hat{\Gamma}']\!]^\flat_i = [\![\hat{\Delta}']\!]^\flat_j$. In this case, the reasoning proceeds as follows:

$$
\begin{array}{rcl}
[\![\hat{\Gamma}', x{:}^k A]\!]^\flat_{i+1} & = & [\![\hat{\Delta}', x{:}^h A]\!]^\flat_{j+1} \\
\left\{ \begin{array}{ll} [\![\hat{\Gamma}']\!]^\flat_{i+1}, x{:}^{\mathsf{r}} A & \text{if } k \leq i+1 \\ [\![\hat{\Gamma}']\!]^\flat_{i+1}, x{:}^{\mathsf{c}} A & \text{otherwise} \end{array} \right. & = & \left\{ \begin{array}{ll} [\![\hat{\Delta}']\!]^\flat_{j+1}, x{:}^{\mathsf{r}} A & \text{if } h \leq j+1 \\ [\![\hat{\Delta}']\!]^\flat_{j+1}, x{:}^{\mathsf{c}} A & \text{otherwise} \end{array} \right.
\end{array}
$$

At this point, we note that $k \leq i+1$ and $h \leq j+1$ are exactly the invariants that we have been careful to maintain about Γ and Δ.

$$
[\![\hat{\Gamma}']\!]^\flat_{i+1}, x{:}^{\mathsf{r}} A \quad = \quad [\![\hat{\Delta}']\!]^\flat_{j+1}, x{:}^{\mathsf{r}} A
$$

We prove this remaining goal by using the induction hypothesis on $[\![\hat{\Gamma}']\!]^\flat_i = [\![\hat{\Delta}']\!]^\flat_j$ to conclude that $[\![\hat{\Gamma}']\!]^\flat_{i+1} = [\![\hat{\Delta}']\!]^\flat_{j+1}$. □

Lemma A.3.3 *If* $\Gamma \cong \Delta$ *then* $x{:}^{\mathsf{r}} A \in \Gamma$ *iff* $x{:}^{\mathsf{r}} A \in \Delta$

Proof. Let $\Gamma = [\![\hat{\Gamma}]\!]_i$ and $\Delta = [\![\hat{\Delta}]\!]_j$. After unfolding some definitions, we see that our goal is really to prove $x{:}^{\mathsf{r}} A \in [\![\hat{\Gamma}]\!]_i$ iff $x{:}^{\mathsf{r}} A \in [\![\hat{\Delta}]\!]_j$ given the assumption $[\![\hat{\Gamma}]\!]^\flat_i = [\![\hat{\Delta}]\!]^\flat_j$. We proceed by induction on the length of $[\![\hat{\Gamma}]\!]^\flat_i$ (=$[\![\hat{\Delta}]\!]^\flat_j$). In each case, we reason backwards from the goal to some obviously true statement.

In the case that $[\![\hat{\Gamma}]\!]^\flat_i = [\![\hat{\Delta}]\!]^\flat_j = \varepsilon$, both $\hat{\Gamma}$ and $\hat{\Delta}$ must equal $\hat{\varepsilon}$. In this case, the reasoning proceeds as follows:

$$
\begin{array}{rcl}
x{:}^{\mathsf{r}} A \in [\![\hat{\varepsilon}]\!]_i & \text{iff} & x{:}^{\mathsf{r}} A \in [\![\hat{\varepsilon}]\!]_j \\
\text{false} & \text{iff} & \text{false}
\end{array}
$$

In the case that $[\![\hat{\Gamma}]\!]^\flat_i = [\![\hat{\Delta}]\!]^\flat_j \neq \varepsilon$, then there must be some $\hat{\Gamma}'$, y, k, B, $\hat{\Delta}'$, z, h, and C such that $\hat{\Gamma} = \hat{\Gamma}', y{:}^k B$ and $\hat{\Delta} = \hat{\Delta}', z{:}^h C$. Since $[\![\hat{\Gamma}]\!]^\flat_i = [\![\hat{\Delta}]\!]^\flat_j$, it must be

that $y = z$, $B = C$, $[\![\hat{\Gamma}']\!]^\flat_i = [\![\hat{\Delta}']\!]^\flat_j$, and $k \leq i$ iff $h \leq j$. In this case, the following two logical equivalences hold:

$$x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}', y{:}^{k}B]\!]_i \quad \text{iff} \quad x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}']\!]_i \vee (x = y \wedge A = B \wedge k \leq i)$$
$$x{:}^{\mathsf{r}}A \in [\![\hat{\Delta}', y{:}^{h}B]\!]_j \quad \text{iff} \quad x{:}^{\mathsf{r}}A \in [\![\hat{\Delta}']\!]_j \vee (x = y \wedge A = B \wedge h \leq j)$$

The two right-hand sides are equivalent because we know $k \leq i$ iff $h \leq j$ and we know $x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}']\!]_i$ iff $x{:}^{\mathsf{r}}A \in [\![\hat{\Delta}']\!]_j$ by the induction hypothesis on $[\![\hat{\Gamma}']\!]^\flat_i = [\![\hat{\Delta}']\!]^\flat_j$. Therefore, the two left-hand sides are equivalent:

$$x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}', y{:}^{k}B]\!]_i \quad \text{iff} \quad x{:}^{\mathsf{r}}A \in [\![\hat{\Delta}', y{:}^{h}B]\!]_j$$

□

Both $\flat$ and $\sharp$ preserve the structure of typing contexts

Theorem A.3.4 (Soundness) *The following identities hold:*

$$\varepsilon = \varepsilon^\flat \qquad \Gamma^\flat, x{:}^{\tau}A = (\Gamma, x{:}^{\tau}A)^\flat \qquad (\Gamma^\flat)^\circ = (\Gamma^\circ)^\flat \qquad x{:}^{\mathsf{r}}A \in \Gamma \text{ } \textit{iff} \text{ } x{:}^{\mathsf{r}}A \in \Gamma^\flat$$

Proof. The first identity is obvious:

$$\varepsilon^\flat = [\![\hat{\varepsilon}]\!]^\flat_0 = \varepsilon.$$

The second is proved by simple equational reasoning:

$$\Gamma^\flat, x{:}^{\tau}A \quad = \quad [\![\hat{\Gamma}]\!]^\flat_i, x{:}^{\tau}A \quad = \quad \begin{cases} ([\![\hat{\Gamma}]\!]^\flat_i, x{:}^{\mathsf{c}}A) & \text{if } \tau = \mathsf{c} \\ ([\![\hat{\Gamma}]\!]^\flat_i, x{:}^{\mathsf{r}}A) & \text{if } \tau = \mathsf{r} \end{cases} \quad =$$

$$\begin{cases} [\![\hat{\Gamma}, x{:}^{i+1}A]\!]^\flat_i & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}, x{:}^{i}A]\!]^\flat_i & \text{if } \tau = \mathsf{r} \end{cases} \quad = \quad \left(\begin{cases} [\![\hat{\Gamma}, x{:}^{i+1}A]\!]_i & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}, x{:}^{i}A]\!]_i & \text{if } \tau = \mathsf{r} \end{cases} \right)^\flat \quad =$$

$$([\![\hat{\Gamma}]\!]_i, x{:}^{\tau}A)^\flat \quad = \quad (\Gamma, x{:}^{\tau}A)^\flat$$

The third identity simplifies to $[\![\hat{\Gamma}]\!]_i^{\flat\circ} = [\![\hat{\Gamma}]\!]_{i+1}^{\flat}$, which we prove by induction on $\hat{\Gamma}$. In the case that $\hat{\Gamma} = \hat{\varepsilon}$, we immediately conclude $[\![\hat{\varepsilon}]\!]_i^{\flat\circ} = \varepsilon = [\![\hat{\varepsilon}]\!]_{i+1}^{\flat}$. In the case that $\hat{\Gamma} = \hat{\Gamma}', x{:}^j A$, we have

$$\begin{aligned}
[\![\hat{\Gamma}', x{:}^j A]\!]_i^{\flat\circ} &= \left(\begin{cases} [\![\hat{\Gamma}']\!]_i^{\flat}, x{:}^{\mathsf{r}} A & \text{if } j \leq i \\ [\![\hat{\Gamma}']\!]_i^{\flat}, x{:}^{\mathsf{c}} A & \text{if } j = i+1 \end{cases} \right)^{\circ} \\
&= \begin{cases} ([\![\hat{\Gamma}']\!]_i^{\flat}, x{:}^{\mathsf{r}} A)^{\circ} & \text{if } j \leq i \\ ([\![\hat{\Gamma}']\!]_i^{\flat}, x{:}^{\mathsf{c}} A)^{\circ} & \text{if } j = i+1 \end{cases} \\
&= [\![\hat{\Gamma}']\!]_i^{\flat\circ}, x{:}^{\mathsf{r}} A
\end{aligned}$$

and

$$\begin{aligned}
[\![\hat{\Gamma}', x{:}^j A]\!]_{i+1}^{\flat} &= \begin{cases} [\![\hat{\Gamma}']\!]_{i+1}^{\flat}, x{:}^{\mathsf{r}} A & \text{if } j \leq i+1 \\ [\![\hat{\Gamma}']\!]_{i+1}^{\flat}, x{:}^{\mathsf{c}} A & \text{if } j = i+2 \end{cases} \\
&= [\![\hat{\Gamma}']\!]_{i+1}^{\flat}, x{:}^{\mathsf{r}} A
\end{aligned}$$

Where the final step follows from the invariant for cleverly represented typing contexts. In this case, we use the invariant to conclude $j \leq i$ since $[\![\hat{\Gamma}', x{:}^j A]\!]_i$ is well-formed. The final terms in each string of equations are equal to each other by the induction hypothesis on $\hat{\Gamma}'$.

The fourth identity, namely "$x{:}^{\mathsf{r}} A \in [\![\hat{\Gamma}]\!]_i$ iff $x{:}^{\mathsf{r}} A \in [\![\hat{\Gamma}]\!]_i^{\flat}$", proceeds by induction over $\hat{\Gamma}$. In the case that $\hat{\Gamma} = \hat{\varepsilon}$, both the left- and right-hand sides are false. In the case that $\hat{\Gamma} = \hat{\Gamma}', y{:}^j B$, we have

$$x{:}^{\mathsf{r}} A \in [\![\hat{\Gamma}', y{:}^j B]\!]_i \quad \text{iff} \quad x{:}^{\mathsf{r}} A \in [\![\hat{\Gamma}']\!]_i \vee (x = y \wedge j \leq i \wedge A = B)$$

and

$$\begin{aligned}
x{:}^{\mathsf{r}} A \in [\![\hat{\Gamma}', y{:}^j B]\!]_i^{\flat} \quad &\text{iff} \quad x{:}^{\mathsf{r}} A \in \begin{cases} [\![\hat{\Gamma}']\!]_i^{\flat}, y{:}^{\mathsf{r}} B & \text{if } j \leq i \\ [\![\hat{\Gamma}']\!]_i^{\flat}, y{:}^{\mathsf{c}} B & \text{if } j = i+1 \end{cases} \\
&\text{iff} \quad x{:}^{\mathsf{r}} A \in [\![\hat{\Gamma}']\!]_i^{\flat} \vee (x = y \wedge j \leq i \wedge A = B)
\end{aligned}$$

The final formulas in each string of equations are equal to each other by the induction hypothesis on $\hat{\Gamma}'$. □

Theorem A.3.5 (Completeness) *The following identities hold:*

$$\varepsilon \cong \varepsilon^\sharp \qquad \Gamma^\sharp, x{:}^\tau A \cong (\Gamma, x{:}^\tau A)^\sharp \qquad (\Gamma^\sharp)^\circ \cong (\Gamma^\circ)^\sharp \qquad x{:}^{\mathsf{r}} A \in \Gamma \text{ iff } x{:}^{\mathsf{r}} A \in \Gamma^\sharp$$

Proof. The first identity follows from simple equational reasoning. The clever implementation of ε is $[\![\hat{\varepsilon}]\!]_0$, which equals $\varepsilon^\sharp$. Therefore $\varepsilon^\sharp = \varepsilon$ and $\varepsilon^\sharp \cong \varepsilon$, because $\cong$ is reflexive.

The second identity follows by simple equational reasoning. Let $[\![\hat{\Gamma}]\!]_i$ be the value of $\Gamma^\sharp$. Then we simply calculate

$$\Gamma^\sharp, x{:}^\tau A \quad = \quad [\![\hat{\Gamma}]\!]_i, x{:}^\tau A \quad = \quad \begin{cases} [\![\hat{\Gamma}, x{:}^{i+1} A]\!]_i & \text{iff } \tau = \mathsf{c} \\ [\![\hat{\Gamma}, x{:}^{i} A]\!]_i & \text{iff } \tau = \mathsf{r} \end{cases} \quad = \quad (\Gamma, x{:}^\tau A)^\sharp.$$

Therefore $\Gamma^\sharp, x{:}^\tau A \cong (\Gamma, x{:}^\tau A)^\sharp$, again by reflexivity of $\cong$.

The third identity is proved by induction on Γ. In the case where $\Gamma = \varepsilon$, we have $(\varepsilon^\sharp)^\circ = [\![\hat{\varepsilon}]\!]_0^\circ = [\![\hat{\varepsilon}]\!]_1$ and $(\varepsilon^\circ)^\sharp = \varepsilon^\sharp = [\![\hat{\varepsilon}]\!]_0$. The map $\flat$ sends both $[\![\hat{\varepsilon}]\!]_1$ and $[\![\hat{\varepsilon}]\!]_0$ to ε. Therefore $(\varepsilon^\sharp)^\circ \cong (\varepsilon^\circ)^\sharp$. Now consider the case where $\Gamma = \Gamma', x{:}^\tau A$. Let $[\![\hat{\Gamma}']\!]_i$ be the value of $\Gamma'^\sharp$. Then, by the induction hypothesis, we have

$$\Gamma'^{\circ\sharp} = \Gamma'^{\sharp\circ} = [\![\hat{\Gamma}']\!]_i^\circ = [\![\hat{\Gamma}']\!]_{i+1}.$$

Therefore, the left-hand side reduces as follows

$$(\Gamma', x{:}^\tau A)^{\circ\sharp} = (\Gamma'^\circ, x{:}^{\mathsf{r}} A)^\sharp = [\![\hat{\Gamma}', x{:}^{i+1} A)]\!]_{i+1}$$

and, after applying $\flat$, we have

$$(\Gamma', x{:}^\tau A)^{\circ\sharp\flat} = [\![\hat{\Gamma}', x{:}^{i+1} A)]\!]_{i+1}^\flat = [\![\hat{\Gamma}']\!]_{i+1}^\flat, x{:}^{\mathsf{r}} A.$$

Meanwhile, on the right-hand side:

$$(\Gamma', x{:}^\tau A)^{\sharp\circ} = \left(\begin{cases} [\![\hat{\Gamma}', x{:}^{i+1} A]\!]_i & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}', x{:}^{i} A]\!]_i & \text{if } \tau = \mathsf{r} \end{cases} \right)^\circ = \begin{cases} [\![\hat{\Gamma}', x{:}^{i+1} A]\!]_{i+1} & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}', x{:}^{i} A]\!]_{i+1} & \text{if } \tau = \mathsf{r} \end{cases}$$

and, after applying $\flat$,

$$(\Gamma', x{:}^{\tau}A)^{\sharp\circ\flat} = \begin{cases} [\![\hat{\Gamma}', x{:}^{i+1}A]\!]^{\flat}_{i+1} & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}', x{:}^{i}A]\!]^{\flat}_{i+1} & \text{if } \tau = \mathsf{r} \end{cases}$$
$$= \begin{cases} [\![\hat{\Gamma}']\!]^{\flat}_{i+1}, x{:}^{\mathsf{r}}A & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}']\!]^{\flat}_{i+1}, x{:}^{\mathsf{r}}A & \text{if } \tau = \mathsf{r} \end{cases} = [\![\hat{\Gamma}']\!]^{\flat}_{i+1}, x{:}^{\mathsf{r}}A$$

Therefore,

$$(\Gamma', x{:}^{\tau}A)^{\circ\sharp\flat} = (\Gamma', x{:}^{\tau}A)^{\sharp\circ\flat}$$

which is the same as saying

$$(\Gamma', x{:}^{\tau}A)^{\circ\sharp} \cong (\Gamma', x{:}^{\tau}A)^{\sharp\circ}$$

The fourth identity is proved by induction over Γ. In the case that $\Gamma = \varepsilon$, both $x{:}^{\mathsf{r}}A \in \varepsilon$ and $x{:}^{\mathsf{r}}A \in \varepsilon^{\sharp}$ (iff $x{:}^{\mathsf{r}}A \in [\![\hat{\varepsilon}]\!]_0$) are false. In the case that $\Gamma = \Gamma', y{:}^{\tau}B$, we have

$$x{:}^{\mathsf{r}}A \in \Gamma', y{:}^{\tau}B \quad \text{iff} \quad x{:}^{\mathsf{r}}A \in \Gamma' \vee (x = y \wedge \tau = \mathsf{r} \wedge A = B)$$

and, assuming $\Gamma'^{\sharp} = [\![\hat{\Gamma}']\!]_i$,

$$x{:}^{\mathsf{r}}A \in (\Gamma', y{:}^{\tau}B)^{\sharp} \quad \text{iff} \quad \begin{cases} x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}', y{:}^{i+1}B]\!]_i & \text{if } \tau = \mathsf{c} \\ x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}', y{:}^{i}B]\!]_i & \text{if } \tau = \mathsf{r} \end{cases}$$
$$\text{iff} \quad \begin{cases} x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}']\!]_i & \text{if } \tau = \mathsf{c} \\ x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}']\!]_i \vee (x = y \wedge A = B) & \text{if } \tau = \mathsf{r} \end{cases}$$
$$\text{iff} \quad x{:}^{\mathsf{r}}A \in [\![\hat{\Gamma}']\!]_i \vee (x = y \wedge \tau = \mathsf{r} \wedge A = B)$$

The final terms in each string of equations are equal to each other by the induction hypothesis on Γ'. □

The mappings $\sharp$ and $\flat$ are inverses

Lemma A.3.6 ($\flat$ undoes $\sharp$)

$$(\Gamma^{\sharp})^{\flat} = \Gamma$$

Proof. By induction on Γ. In the base case, $\Gamma = \varepsilon$ and we have $(\varepsilon^\sharp)^\flat = [\![\hat{\varepsilon}]\!]_0^\flat = \varepsilon$. The inductive case proceeds as follows:

$$\begin{aligned}
((\Gamma, x{:}^\tau A)^\sharp)^\flat &= \left(\begin{cases} [\![\hat{\Gamma}, x{:}^{j+1} A]\!]_j & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}, x{:}^{j} A]\!]_j & \text{if } \tau = \mathsf{r} \end{cases} \text{ where } \Gamma^\sharp = [\![\hat{\Gamma}]\!]_j \right)^\flat \\
&= \begin{cases} [\![\hat{\Gamma}, x{:}^{j+1} A]\!]_j^\flat & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}, x{:}^{j} A]\!]_j^\flat & \text{if } \tau = \mathsf{r} \end{cases} \text{ where } \Gamma^\sharp = [\![\hat{\Gamma}]\!]_j \\
&= \begin{cases} [\![\hat{\Gamma}]\!]_j^\flat, x{:}^{\mathsf{c}} A & \text{if } \tau = \mathsf{c} \\ [\![\hat{\Gamma}]\!]_j^\flat, x{:}^{\mathsf{r}} A & \text{if } \tau = \mathsf{r} \end{cases} \text{ where } \Gamma^\sharp = [\![\hat{\Gamma}]\!]_j \\
&= [\![\hat{\Gamma}]\!]_j^\flat, x{:}^\tau A \text{ where } \Gamma^\sharp = [\![\hat{\Gamma}]\!]_j \\
&= (\Gamma^\sharp)^\flat, x{:}^\tau A \quad = \quad \Gamma, x{:}^\tau A
\end{aligned}$$

where the last step uses the induction hypothesis. □

Corollary A.3.7 ($\sharp$ undoes $\flat$)

$$\Gamma \cong \Delta \Longrightarrow (\Gamma^\flat)^\sharp \cong \Delta$$

Proof. Assume $\Gamma \cong \Delta$. By definition, this means $\Gamma^\flat = \Delta^\flat$. By the previous lemma, $((\Gamma^\flat)^\sharp)^\flat = \Gamma^\flat$. By transitivity, we have $((\Gamma^\flat)^\sharp)^\flat = \Delta^\flat$, which is equivalent, by definition, to $(\Gamma^\flat)^\sharp \cong \Delta$. □

A.4 META-THEORY OF EPTS$^{\mathcal{C}}$

Lemma A.4.1 (Correctness of Generalized Context Reset)

$$\sigma(\Gamma^{\circ(\rho)}) = \begin{cases} \sigma\Gamma & \textit{if } \sigma(\rho) = \mathsf{r} \\ (\sigma\Gamma)^\circ & \textit{if } \sigma(\rho) = \mathsf{c} \end{cases}$$

Proof. By induction on Γ. In each case we proceed by cases on $\sigma(\rho)$.

The ε case:

Assuming $\sigma(\rho) = \mathsf{r}$:	Assuming $\sigma(\rho) = \mathsf{c}$:
$\sigma(\varepsilon^{\circ\rho}) = \sigma\varepsilon$	$\sigma(\varepsilon^{\circ\rho}) = \sigma\varepsilon = \varepsilon = \varepsilon^{\circ} = (\sigma\varepsilon)^{\circ}$

The $\Gamma, x{:}^{\gamma}A$ case:

Assuming $\sigma(\rho) = \mathsf{r}$:	Assuming $\sigma(\rho) = \mathsf{c}$:
$\sigma((\Delta, x{:}^{\gamma}A))^{\circ(\rho)}$	$\sigma((\Delta, x{:}^{\gamma}A))^{\circ(\rho)}$
$= \sigma(\Delta^{\circ(\rho)}, x{:}^{\neg\rho\wedge\gamma}A)$	$= \sigma(\Delta^{\circ(\rho)}, x{:}^{\neg\rho\wedge\gamma}A)$
$= \sigma(\Delta^{\circ(\rho)}), x{:}^{\neg\sigma(\rho)\wedge\sigma(\gamma)}\sigma A$	$= \sigma(\Delta^{\circ(\rho)}), x{:}^{\neg\sigma(\rho)\wedge\sigma(\gamma)}\sigma A$
$= (\sigma\Delta), x{:}^{\neg\mathsf{r}\wedge\sigma(\gamma)}\sigma A$	$= (\sigma\Delta)^{\circ}, x{:}^{\neg\mathsf{c}\wedge\sigma(\gamma)}\sigma A$
$= (\sigma\Delta), x{:}^{\sigma(\gamma)}\sigma A$	$= (\sigma\Delta)^{\circ}, x{:}^{\mathsf{r}}\sigma A$
$= (\sigma(\Delta, x{:}^{\gamma}A))$	$= (\sigma\Delta, x{:}^{\sigma\gamma}\sigma A)^{\circ}$
	$= (\sigma(\Delta, x{:}^{\gamma}A))^{\circ}$

since $\mathsf{c} = \mathsf{true}$ and $\mathsf{r} = \mathsf{false}$. □

Lemma A.4.2 $\sigma(M[N/x]) = \sigma M[\sigma N/x]$

Proof. By induction on M.

Step	**Justification**	
Case x	$\sigma(x[N/x])$	
	$= \sigma N$	def. of subst.
	$= x[\sigma N/x]$	def. of subst.
	$= \sigma x[\sigma N/x]$	def. of subst.
Case $y(\neq x)$	$\sigma(y[N/x])$	
	$= \sigma y$	def. of subst.
	$= y[\sigma N/x]$	def. of subst.
	$= \sigma y[\sigma N/x]$	def. of subst.

Step	Justification	
Case $\Pi^\alpha y{:}A.\, B$	$\sigma((\Pi^\alpha y{:}A.\, B)[N/x])$	
	$= \sigma(\Pi^\alpha y{:}A[N/x].\, B[N/x])$	def. of subst.
	$= \Pi^{\sigma\alpha} y{:}\sigma(A[N/x]).\, \sigma(B[N/x])$	def. of eval.
	$= \Pi^{\sigma\alpha} y{:}\sigma A[\sigma N/x].\, \sigma B[\sigma N/x]$	ind. hyp.
	$= (\Pi^{\sigma\alpha} y{:}\sigma A.\, \sigma B)[\sigma N/x]$	def. of subst.
	$= (\sigma(\Pi^\alpha y{:}A.\, B))[\sigma N/x]$	def. of eval.
Case $\lambda^\alpha y{:}A.\, M$	similar to the $\Pi^\alpha y{:}A.\, B$ case	
Case $M@^\alpha N$	similar to the $\Pi^\alpha y{:}A.\, B$ case	
Case s	similar to the $y(\neq x)$ case	

Lemma A.4.3 *If $\sigma P = M[\sigma N/x]$, then $M = \sigma M'$ for some M'.*

Proof. By induction on M.

Step	Justification
Case y	
1. $y = \sigma y$	def. of eval.
Case $\Pi^\tau y{:}A.\, B$	
1. $\sigma P = (\Pi^\tau y{:}A.\, B)[\sigma N/x]$	hypothesis
2. $\sigma P = \Pi^\tau y{:}A[\sigma N/x].\, B[\sigma N/x]$	by 1, def. of subst.
3. $P = \Pi^\alpha y{:}Q.\, R$	}
4. $\sigma\alpha = \tau$	} by 2
5. $\sigma Q = A[\sigma N/x]$	} (for some α, Q, R)
6. $\sigma R = B[\sigma N/x]$	}
7. $A = \sigma A'$	ind. hyp. on 5 (for some A')
8. $B = \sigma B'$	ind. hyp. on 6 (for some B')
9. $\Pi^\tau y{:}A.\, B = \sigma(\Pi^\alpha y{:}A'.\, B')$	by 4, 7, 8
Cases $\lambda^\alpha y{:}A.\, M$ and $M@^\alpha N$	

Step	Justification
similar to the $\Pi^{\alpha} y{:}A.\, B$ case	
Case s	
similar to the y case	

□

Lemma A.4.4

$$\frac{\sigma P \to_{\beta} Q}{(\exists\, Q') \quad \sigma Q' = Q \quad \wedge \quad P \to_{\beta} Q'}$$

Proof. By induction on the derivation of $\sigma P \to_{\beta} Q$.

Step		Justification
Case Beta: $(\lambda^{\tau} x{:}A.\, M)@^{\tau'} N \to_{\beta} M[N/x]$		
1.	$\sigma P = (\lambda^{\tau} x{:}A.\, M)@^{\tau'} N$	hypothesis
2.	$Q = M[N/x]$	hypothesis
3.	$P = (\lambda^{\alpha} x{:}A'.\, M')@^{\alpha'} N'$	(for some α, α', A', M', N')
4.	$\sigma\alpha = \tau$	
5.	$\sigma A' = A$	by 1
6.	$\sigma M' = M$	
7.	$\sigma\alpha' = \tau'$	
8.	$\sigma N' = N$	
9.	let $Q' = M'[N'/x]$	definition
10.	$\sigma(M'[N'/x]) = \sigma M'[\sigma N'/x]$	by Lemma A.4.2
11.	$\sigma Q' = Q \ \wedge \ P \to_{\beta} Q'$	by 9, 10, 6, 8, 2, 3, Beta
Case Π-Cong1: $\left(\dfrac{A \to_{\beta} B}{\Pi^{\tau} x{:}A.\, C \to_{\beta} \Pi^{\tau} x{:}B.\, C}\right)$		
1.	$\sigma P = \Pi^{\tau} x{:}A.\, C$	hypothesis
2.	$Q = \Pi^{\tau} x{:}B.\, C$	hypothesis

Step		Justification
3.	$A \to_\beta B$	hypothesis
4.	$P = \Pi^\alpha x{:}R.\, C'$	
5.	$\sigma\alpha = \tau$	by 1
6.	$\sigma R = A$	(for some α, R, B')
7.	$\sigma C' = C$	
8.	$\sigma R' = B$	ind. hyp. on 6, 3
9.	$R \to_\beta R'$	(for some R')
10.	let $Q' = \Pi^\alpha x{:}R'.\, C'$	definition
11.	$\sigma Q' = Q$	by 10, 5, 8, 7, 2
12.	$P \to_\beta Q'$	by 4, 9, 10, Π-Cong1
13.	$\sigma Q' = Q \ \wedge \ P \to_\beta Q'$	by 11, 12
All other Congruence Cases		
	similar to the Π-Cong1 case	

□

Lemma A.4.5 *If $M \to_\beta N$, then $\mathsf{true} \vdash M =_\beta N$.*

Proof. By induction on the derivation of $M \to_\beta N$.

Step		Justification
Case Beta: $(\lambda^\alpha x{:}A.\, M)@^{\alpha'} N \to_\beta M[N/x]$		
1.	$\mathsf{true} \vdash (\lambda^\alpha x{:}A.\, M)@^{\alpha'} N =_\beta M[N/x]$	Beta
Case Π-Cong1: $\left(\dfrac{A \to_\beta B}{\Pi^\alpha x{:}A.\, C \to_\beta \Pi^\alpha x{:}B.\, C}\right)$		
1.	$A \to_\beta B$	hypothesis
2.	$\mathsf{true} \vdash A =_\beta B$	ind. hyp. on 1
3.	$\mathsf{true} \vdash C =_\beta C$	by Refl
4.	$\alpha = \alpha \wedge \mathsf{true} \wedge \mathsf{true} \vdash \Pi^\alpha x{:}A.\, C =_\beta \Pi^\alpha x{:}B.\, C$	by CongPi, 2, 3

Step	Justification
5. $(\alpha = \alpha \wedge \mathsf{true} \wedge \mathsf{true}) = \mathsf{true}$	tautology
6. $\mathsf{true} \vdash \Pi^{\alpha} x{:}A.\, C =_{\beta} \Pi^{\alpha} x{:}B.\, C$	by 4, 5
All other Congruence Cases	
similar to the Π-CONG1 case	

□

Lemma A.4.6 (Pre-Completeness of EPTS$^{\mathcal{C}}$ conversion rules)

$$\frac{\sigma M = \sigma N}{(\exists \mathcal{C}) \quad \mathcal{C} \vdash M =_{\beta} N \quad \wedge \quad \sigma \vDash \mathcal{C}}$$

Proof. By induction on σM.

Step	Justification
Case x	
1. $\sigma M = \sigma N$	assumption
2. $\sigma M = x$	hypothesis
3. $M = x$	by 2
4. $N = x$	by 1, 2
5. let $\mathcal{C} = \mathsf{true}$	definition
6. $\mathcal{C} \vdash M =_{\beta} N$	by 3, 4, REFL
7. $\sigma \vDash \mathsf{true}$	def. of $\vDash$
Case $\Pi^{\tau} x{:}A.\, B$	
1. $\sigma M = \sigma N$	assumption
2. $\sigma M = \Pi^{\tau} x{:}A.\, B$	hypothesis
3. $M = \Pi^{\alpha'} x{:}A'.\, B'$	by 2 (for some α', A', B')
4. $\sigma\alpha' = \tau$	
5. $\sigma A' = A$	
6. $\sigma B' = B$	

Step		Justification
7.	$N = \Pi^{\alpha''} x{:}A''.\, B''$	
8.	$\sigma\alpha'' = \tau$	by 2
9.	$\sigma A'' = A$	(for some α'', A'', B'')
10.	$\sigma B'' = B$	
11.	$A' = A''$	by 5, 9
12.	$\mathcal{C}_1 \vdash A' =_\beta A''$	ind. hyp. on 11
13.	$\sigma \vDash \mathcal{C}_1$	(for some $\mathcal{C}_1$)
14.	$\sigma B' = \sigma B''$	by 6, 10
15.	$\mathcal{C}_2 \vdash A' =_\beta A''$	ind. hyp. on 14
16.	$\sigma \vDash \mathcal{C}_2$	(for some $\mathcal{C}_2$)
17.	let $\mathcal{C} = \alpha' = \alpha'' \wedge \mathcal{C}_1 \wedge \mathcal{C}_2$	definition
18.	$\mathcal{C} \vdash M =_\beta N$	by 17, 3, 7, CongPi, 12, 15
19.	$\mathcal{C} \vDash \alpha' = \alpha''$	by 4, 8
20.	$\sigma \vDash \mathcal{C}$	by 17, 13, 16, 19
21.	$\mathcal{C} \vdash M =_\beta N \wedge \sigma \vDash \mathcal{C}$	by 18, 20
Cases $\lambda^\alpha x{:}A.\, M$ and $M@^\alpha N$		
	similar to the $\Pi^\alpha x{:}A.\, B$ case	
Case s		
	similar to the x case	

□

Lemma A.4.7

$$\frac{\sigma P = M[\sigma N/x]}{(\exists M', \mathcal{C}) \qquad M = \sigma M' \qquad \mathcal{C} \vdash P =_\beta M'[N/x] \qquad \sigma \vDash \mathcal{C}}$$

Proof. By induction on M.

Step	Justification
Case $M = x$	
1. $\sigma P = x[\sigma N/x]$	hypothesis
2. $x[\sigma N/x] = \sigma N$	def. of subst.
3. $\sigma P = \sigma N$	by 1, 2
4. $\mathcal{C} \vdash P =_\beta N$	by Lemma A.4.6 (for some $\mathcal{C}$)
5. $\sigma \vDash \mathcal{C}$	by Lemma A.4.6 (for some $\mathcal{C}$)
6. let $M' = x$	definition
7. $\sigma x = x$	def. of eval.
8. $\sigma M' = M$	by 7, 6
9. $M'[N/x] = N$	by 7, def. of subst.
10. $\mathcal{C} \vdash P =_\beta M'[N/x]$	by 4, 9
conclusions 5, 8, and 10	goal!
Case $M = y(\neq x)$	
1. $\sigma P = y[\sigma N/x]$	hypothesis
2. $y[\sigma N/x] = y$	def. of subst.
3. $\sigma P = y$	by 1, 2
4. $P = y$	by 3
5. let $M' = y$	definition
6. $\mathsf{true} \vdash y =_\beta y$	Refl
7. $M'[N/x] = y$	by 5, def. of subst.
8. $\mathsf{true} \vdash P =_\beta M'[N/x]$	by 6, 4, 7
9. $\sigma \vDash \mathsf{true}$	def. of $\vDash$
10. $\sigma M' = M$	by 5, def. of eval.
conclusions 8, 9, and 10	goal!
Case $M = \Pi^\tau y{:}A.\, B$	
1. $\sigma P = (\Pi^\tau y{:}A.\, B)[\sigma N/x]$	hypothesis
2. $\sigma P = \Pi^\tau y{:}A[\sigma N/x].\, B[\sigma N/x]$	by 1, def. of subst.

Step		Justification
3.	$P = \Pi^\alpha y{:}Q.\, R$	
4.	$\sigma\alpha = \tau$	by 2
5.	$\sigma Q = A[\sigma N/x]$	(for some α, Q, R)
6.	$\sigma R = B[\sigma N/x]$	
7.	$A = \sigma A'$	
8.	$\mathcal{C} \vdash Q =_\beta A'[N/x]$	ind. hyp. on 5
9.	$\sigma \vDash \mathcal{C}$	(for some A')
10.	$B = \sigma B'$	
11.	$\mathcal{D} \vdash R =_\beta B'[N/x]$	ind. hyp. on 6
12.	$\sigma \vDash \mathcal{D}$	(for some B')
13.	$\mathcal{C} \wedge \mathcal{D} \vdash \Pi^\alpha y{:}Q.\, R =_\beta \Pi^\alpha y{:}A'[N/x].\, B'[N/x]$	CongPi on 8, 11
14.	let $M' = \Pi^\alpha y{:}A'.\, B'$	definition
15.	$\mathcal{C} \wedge \mathcal{D} \vdash P =_\beta M'[N/x]$	by 13, 3, 14, def. of subst.
16.	$\sigma \vDash \mathcal{C} \wedge \mathcal{D}$	by 9, 12
17.	$\sigma M' = M$	by 14, 4, 7, 10
Cases $\lambda^\alpha y{:}A.\, M$ and $M@^\alpha N$		
	similar to the $\Pi^\alpha y{:}A.\, B$ case	
Case s		
	similar to the y case	

□

Theorem A.4.8 (Soundness of EPTS$^{\mathcal{C}}$ conversion rules)

$$\frac{\mathcal{C} \vdash M =_\beta N \qquad \sigma \vDash \mathcal{C}}{\sigma M =_\beta \sigma N}$$

Proof. By induction on the derivation of $\mathcal{C} \vdash M =_\beta N$.

Step	**Justification**
Case Refl: $\left(\dfrac{}{\mathsf{true} \vdash M =_\beta M}\right)$	
1. $\sigma M = \sigma M$	reflexivity of $=_\beta$
Case Symm: $\left(\dfrac{\mathcal{C} \vdash M =_\beta N}{\mathcal{C} \vdash N =_\beta M}\right)$	
1. $\sigma \vDash \mathcal{C}$	assumption
2. $\mathcal{C} \vdash M =_\beta N$	hypothesis
3. $\sigma M =_\beta \sigma N$	ind. hyp. on 2, 1
4. $\sigma N =_\beta \sigma M$	by 3, symmetry of $=_\beta$
Case Trans: $\left(\dfrac{\mathcal{C} \vdash M =_\beta M'' \qquad \mathcal{C} \vdash M'' =_\beta M'}{\mathcal{C} \vdash M =_\beta M'}\right)$	
1. $\sigma \vDash \mathcal{C}$	assumption
2. $\mathcal{C} \vdash M =_\beta M''$	hypothesis
3. $\mathcal{C} \vdash M'' =_\beta M'$	hypothesis
3. $\sigma M =_\beta \sigma M''$	ind. hyp. on 2, 1
4. $\sigma M'' =_\beta \sigma M'$	ind. hyp. on 3, 1
5. $\sigma M =_\beta \sigma M'$	by 3, 4, transitivity of $=_\beta$
Case Beta: $\left(\dfrac{}{\mathsf{true} \vdash (\lambda^\alpha x{:}A.\, M)@^{\alpha'} N =_\beta M[N/x]}\right)$	
1. $\sigma \vDash \mathcal{C}$	assumption
2. $\sigma((\lambda^\alpha x{:}A.\, M)@^{\alpha'} N) = (\lambda^{\sigma\alpha} x{:}\sigma A.\, \sigma M)@^{\alpha'} \sigma N$	def. of eval.
3. $(\lambda^{\sigma\alpha} x{:}\sigma A.\, \sigma M)@^{\alpha'} \sigma N \to_\beta \sigma M[\sigma N/x]$	β-reduction
4. $\sigma M[\sigma N/x] = \sigma(M[N/x])$	def. of eval.
5. $\sigma(\lambda^\alpha x{:}A.\, M)@^{\alpha'} N \to_\beta \sigma(M[N/x])$	by 2, 3, 4
Case CongPi: $\left(\dfrac{\mathcal{C} \vdash A =_\beta A' \qquad \mathcal{D} \vdash B =_\beta B'}{\alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D} \vdash \Pi^\alpha x{:}A.\, B =_\beta \Pi^{\alpha'} x{:}A'.\, B'}\right)$	

Step		Justification
1.	$\sigma \vDash \alpha = \alpha' \wedge \mathcal{C} \wedge \mathcal{D}$	assumption
2.	$\mathcal{C} \vdash A =_\beta A'$	hypothesis
3.	$\mathcal{D} \vdash B =_\beta B'$	hypothesis
4.	$\sigma \vDash \alpha = \alpha'$	by 1
5.	$\sigma \vDash \mathcal{C}$	by 1
6.	$\sigma \vDash \mathcal{D}$	by 1
7.	$\sigma A =_\beta \sigma A'$	ind. hyp. on 2, 5
8.	$\sigma B =_\beta \sigma B'$	ind. hyp. on 3, 6
9.	$\sigma\alpha = \sigma\alpha'$	by 4
10.	$\Pi^{\sigma\alpha} x{:}\sigma A.\, \sigma B =_\beta \Pi^{\sigma\alpha'} x{:}\sigma A'.\, \sigma B'$	by 7, 8, 9
11.	$\sigma(\Pi^{\alpha} x{:}A.\, B) =_\beta \sigma(\Pi^{\alpha'} x{:}A'.\, B')$	by 10, def. of eval.
Cases CongLam and CongApp		
similar to the CongPi case		

□

Theorem A.4.9 (Completeness of EPTS$^{\mathcal{C}}$ conversion rules)

$$\frac{\sigma M =_\beta \sigma N}{(\exists \mathcal{C}) \quad \mathcal{C} \vdash M =_\beta N \quad \wedge \quad \sigma \vDash \mathcal{C}}$$

Proof. Since $\sigma M =_\beta \sigma N$, there exists a term $\widehat{P}$ such that $\sigma M \to_\beta^* \widehat{P}$ and $\sigma N \to_\beta^* \widehat{P}$ (by the Church-Rosser Theorem). By repeated applications of Lemma A.4.4, there exists P_1 and P_2 such that $\sigma P_1 = \sigma P_2 = \widehat{P}$ and $M \to_\beta^* P_1$ and $N \to_\beta^* P_2$. By Lemma A.4.6, there is some constraint $\mathcal{C}$ such that $\mathcal{C} \vdash P_1 =_\beta P_2$ and $\sigma \vDash \mathcal{C}$. By repeated applications of Lemma A.4.5, we have $\mathsf{true} \vdash M =_\beta P_1$ and $\mathsf{true} \vdash N =_\beta P_2$. Therefore, by some applications of Symm and Trans, we can derive $\mathcal{C} \vdash M =_\beta N$, and we already know that $\sigma \vDash \mathcal{C}$. □

Theorem A.4.10 (Soundness of EPTS$^{\mathcal{C}}$ typing rules)

$$\frac{\mathcal{C}\,;\,\Gamma \vdash M :^{\rho} A \qquad \sigma \vDash \mathcal{C}}{\sigma\Gamma \vdash \sigma M :^{\sigma\rho} \sigma A}$$

Proof. By induction on the derivation of $\mathcal{C}\,;\,\Gamma \vdash M :^{\rho} A$.

Step	**Justification**
Case Axiom: $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\mathsf{true}\,;\,\varepsilon \vdash s_1 :^{\mathsf{r}} s_2}\right)$	
1. $(s_1, s_2) \in \mathcal{A}$	hypothesis
2. $\sigma\varepsilon = \varepsilon$	def. of eval.
3. $\forall s.\ \sigma s = s$	def. of eval.
4. $\sigma\mathsf{r} = \mathsf{r}$	def. of eval.
5. $\sigma\varepsilon \vdash \sigma s_1 :^{\sigma\mathsf{r}} \sigma s_2$	by 1, 2, 3, 4, Axiom
Case Var: $\left(\dfrac{\mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{c}} s}{\mathcal{C} \wedge \neg\gamma\,;\,\Gamma, x{:}^{\gamma}A \vdash x :^{\mathsf{r}} A}\right)$	
1. $\sigma \vDash \mathcal{C} \wedge \neg\gamma$	assumption
2. $\mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{c}} s$	hypothesis
3. $\sigma \vDash \mathcal{C}$	} by 1
4. $\sigma \vDash \neg\gamma$	} by 1
5. $\sigma\Gamma \vdash \sigma A :^{\mathsf{c}} s$	ind. hyp. on 1, 2
6. $\sigma\gamma = \mathsf{r}$	by 4
7. $\sigma\Gamma, x{:}^{\mathsf{r}}\sigma A \vdash x :^{\mathsf{r}} \sigma A$	by 5, Var
8. $\sigma\Gamma, x{:}^{\mathsf{r}}\sigma A = \sigma(\Gamma, x{:}^{\gamma}A)$	by 6, def. of eval.
9. $\sigma(\Gamma, x{:}^{\gamma}A) \vdash \sigma x :^{\sigma\mathsf{r}} \sigma A$	by 8, def. of eval.
Case Weak: $\left(\dfrac{\mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{c}} s \qquad \mathcal{D}\,;\,\Gamma \vdash M :^{\tau} B}{\mathcal{C} \wedge \mathcal{D}\,;\,\Gamma, x{:}^{\gamma}A \vdash M :^{\tau} B}\right)$	
1. $\sigma \vDash \mathcal{C} \wedge \mathcal{D}$	assumption

Step		Justification
2.	$\mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{c}} s$	hypothesis
3.	$\mathcal{D}\,;\,\Gamma \vdash M :^{\tau} B$	hypothesis
4.	$\sigma \vDash \mathcal{C}$	} by 1
5.	$\sigma \vDash \mathcal{D}$	
6.	$\sigma\Gamma \vdash \sigma A :^{\mathsf{c}} s$	ind. hyp. on 2, 4
7.	$\sigma\Gamma \vdash \sigma M :^{\tau} \sigma B$	ind. hyp. on 3, 5
8.	$\sigma\Gamma, x{:}^{\sigma\gamma}\sigma A \vdash \sigma M :^{\tau} \sigma B$	by 6, 7, Weak[1]
9.	$\sigma(\Gamma, x{:}^{\gamma} A) \vdash \sigma M :^{\sigma\tau} \sigma B$	by 7, def. of eval.

Case Π-Form: $\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{r}} s_1 \qquad \mathcal{D}\,;\,\Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2}{\mathcal{C} \wedge \mathcal{D}\,;\,\Gamma \vdash \Pi^{\alpha} x{:}A.\, B :^{\mathsf{r}} s_3}\right)$

1.	$\sigma \vDash \mathcal{C} \wedge \mathcal{D}$	assumption
2.	$(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
3.	$\mathcal{C}\,;\,\Gamma \vdash A :^{\mathsf{r}} s_1$	hypothesis
4.	$\mathcal{D}\,;\,\Gamma, x{:}^{\mathsf{r}} A \vdash B :^{\mathsf{r}} s_2$	hypothesis
5.	$\sigma \vDash \mathcal{C}$	} by 1
6.	$\sigma \vDash \mathcal{D}$	
7.	$\sigma\Gamma \vdash \sigma A :^{\mathsf{r}} s_1$	by 3, 5, ind. hyp.
8.	$\sigma(\Gamma, x{:}^{\mathsf{r}} A) \vdash \sigma B :^{\mathsf{r}} s_2$	by 4, 6, ind. hyp.
9.	$\sigma\Gamma, x{:}^{\mathsf{r}}\sigma A \vdash \sigma B :^{\mathsf{r}} s_2$	by 8, def. of eval.
10.	$\sigma\Gamma, x{:}^{\mathsf{r}}\sigma A \vdash \Pi^{\sigma\alpha} x{:}\sigma A.\, \sigma B :^{\mathsf{r}} s_3$	by 2, 7, 9, Π-Form
11.	$\sigma(\Gamma, x{:}^{\mathsf{r}} A) \vdash \sigma(\Pi^{\alpha} x{:}A.\, B) :^{\sigma\mathsf{r}} \sigma s_3$	by 10, def. of eval.

Case Π-Intro: $\left(\dfrac{\mathcal{C}\,;\,\Gamma \vdash \Pi^{\alpha'} x{:}A.\, B :^{\mathsf{c}} s \qquad \mathcal{D}\,;\,\Gamma, x{:}^{\alpha} A \vdash M :^{\mathsf{r}} B}{\mathcal{C} \wedge \mathcal{D} \wedge \alpha = \alpha'\,;\,\Gamma \vdash \lambda^{\alpha} x{:}A.\, M :^{\mathsf{r}} \Pi^{\alpha'} x{:}A.\, B}\right)$

1.	$\sigma \vDash \mathcal{C} \wedge \mathcal{D} \wedge \alpha = \alpha'$	assumption

[1] What is actually required is a slight generalization of Weak in that is easy to prove using the Phase Weakening theorem, namely, that $\Gamma, x{:}^{\tau'} A \vdash M :^{\tau} B$ follows from $\Gamma \vdash A :^{\mathsf{c}} s$ and $\Gamma \vdash M :^{\tau} B$.

Step		Justification
2.	$\mathcal{C}\,;\,\Gamma \vdash \Pi^{\alpha'} x{:}A.\,B :^{\mathsf{c}} s$	hypothesis
3.	$\mathcal{D}\,;\,\Gamma, x{:}^{\alpha} A \vdash M :^{\mathsf{r}} B$	hypothesis
4.	$\sigma \vDash \mathcal{C}$	by 1
5.	$\sigma \vDash \mathcal{D}$	by 1
6.	$\sigma \vDash \alpha = \alpha'$	by 1
7.	$\sigma\alpha = \sigma\alpha'$	by 6
8.	$\sigma\Gamma \vdash \sigma(\Pi^{\alpha'} x{:}A.\,B) :^{\mathsf{c}} s$	ind. hyp. on 2, 4
9.	$\sigma(\Gamma, x{:}^{\alpha} A) \vdash \sigma M :^{\mathsf{r}} \sigma B$	ind. hyp. on 2, 5

Case Π-Elim: $\left(\dfrac{\mathcal{C}\,;\,\Gamma \vdash M :^{\mathsf{r}} \Pi^{\alpha_1} x{:}A.\,B \qquad \mathcal{D}\,;\,\Gamma \vdash N :^{\alpha_2} A}{\mathcal{C} \wedge \mathcal{D} \wedge \alpha_1 = \alpha_2\,;\,\Gamma \vdash M@^{\alpha_2} N :^{\mathsf{r}} B[N/x]}\right)$

1.	$\sigma \vDash \mathcal{C} \wedge \mathcal{D} \wedge \alpha_1 = \alpha_2$	assumption
2.	$\mathcal{C}\,;\,\Gamma \vdash M :^{\mathsf{r}} \Pi^{\alpha_1} x{:}A.\,B$	hypothesis
3.	$\mathcal{D}\,;\,\Gamma \vdash N :^{\alpha_2} A$	hypothesis
4.	$\sigma \vDash \mathcal{C}$	by 1
5.	$\sigma \vDash \mathcal{D}$	by 1
6.	$\sigma \vDash \alpha_1 = \alpha_2$	by 1
7.	$\sigma\alpha_1 = \sigma\alpha_2$	by 6
8.	$\sigma\Gamma \vdash \sigma M :^{\mathsf{r}} \Pi^{\sigma(\alpha_1)} x{:}\sigma A.\,\sigma B$	ind. hyp. on 2, 4
9.	$\sigma\Gamma \vdash \sigma N :^{\sigma(\alpha_2)} \sigma A$	ind. hyp. on 3, 5
10.	$\sigma\Gamma \vdash \sigma M@^{\sigma(\alpha_2)} \sigma N :^{\mathsf{r}} \sigma B[\sigma N/x]$	Π-Elim on 8, 9, 7
11.	$\sigma B[\sigma N/x] = \sigma(B[N/x])$	by Lemma A.4.2
12.	$\sigma\mathsf{r} = \mathsf{r}$	def. of eval.
13.	$\sigma\Gamma \vdash \sigma(M@^{\alpha_2} N) :^{\sigma\mathsf{r}} \sigma(B[N/x])$	by 10, 11, 12

Case Conv: $\left(\dfrac{\mathcal{C}\,;\,\Gamma \vdash M :^{\mathsf{r}} A \qquad \mathcal{D}\,;\,\Gamma \vdash B :^{\mathsf{c}} s \qquad \mathcal{E} \vdash A =_{\beta} B}{\mathcal{C} \wedge \mathcal{D} \wedge \mathcal{E}\,;\,\Gamma \vdash M :^{\mathsf{r}} B}\right)$

1.	$\sigma \vDash \mathcal{C} \wedge \mathcal{D} \wedge \mathcal{E}$	assumption

Step		Justification
2.	$\mathcal{C}\,;\,\Gamma \vdash M :^{\mathsf{r}} A$	hypothesis
3.	$\mathcal{D}\,;\,\Gamma \vdash B :^{\mathsf{c}} s$	hypothesis
4.	$\mathcal{E} \vdash A =_\beta B$	hypothesis
5.	$\sigma \vDash \mathcal{C}$	} by 1
6.	$\sigma \vDash \mathcal{D}$	
7.	$\sigma \vDash \mathcal{E}$	
8.	$\sigma\Gamma \vdash \sigma M :^{\mathsf{r}} \sigma A$	ind. hyp. on 2, 5
9.	$\sigma\Gamma \vdash \sigma B :^{\mathsf{c}} s$	ind. hyp. on 3, 6
10.	$\sigma A =_\beta \sigma B$	by Lemma A.4.8, 4, 7
11.	$\sigma\Gamma \vdash \sigma M :^{\mathsf{r}} \sigma B$	by 8, 9, 10, Conv
12.	$\sigma\mathsf{r} = \mathsf{r}$	def. of eval.
13.	$\sigma\Gamma \vdash \sigma M :^{\sigma\mathsf{r}} \sigma B$	by 11, 12

Case Reset: $\left(\dfrac{\mathcal{C}\,;\,\Gamma^{\circ(\rho)} \vdash M :^{\mathsf{r}} A}{\mathcal{C}\,;\,\Gamma \vdash M :^{\rho} A}\right)$

1.	$\sigma \vDash \mathcal{C}$	assumption
2.	$\mathcal{C}\,;\,\Gamma^{\circ(\rho)} \vdash M :^{\mathsf{r}} A$	hypothesis
3.	$\sigma(\Gamma^{\circ(\rho)}) \vdash \sigma M :^{\mathsf{r}} \sigma A$	ind. hyp. on 2, 1
4.	$\sigma(\rho) = \mathsf{r} \vee \sigma(\rho) = \mathsf{c}$	tautology
5.	$\sigma(\rho) = \mathsf{r}$	assumption
6.	$\sigma(\Gamma^{\circ(\rho)}) = \sigma\Gamma$	by Lemma A.4.1, 5
7.	$\sigma\Gamma \vdash \sigma M :^{\sigma(\rho)} \sigma A$	by 3, 6, 5
8.	$\sigma(\rho) = \mathsf{c}$	assumption
9.	$\sigma(\Gamma^{\circ(\rho)}) = (\sigma\Gamma)^\circ$	by Lemma A.4.1, 8
10.	$(\sigma\Gamma)^\circ \vdash \sigma M :^{\mathsf{r}} \sigma A$	by 3, 9
11.	$\sigma\Gamma \vdash \sigma M :^{\mathsf{c}} \sigma A$	by 10, Reset
12.	$\sigma\Gamma \vdash \sigma M :^{\sigma(\rho)} \sigma A$	by 11, 8

Step	Justification
13. $\sigma\Gamma \vdash \sigma M :^{\sigma(\rho)} \sigma A$	by $\vee$-elimination, 4, 5–7, 8–12

□

Definition A.4.11 (Assignment extension $\boxed{\sigma' \rhd \sigma}$)

$$\sigma' \rhd \sigma \quad = \quad dom(\sigma) \subseteq dom(\sigma') \ \wedge \ \forall \alpha \in dom(\sigma).\ \sigma(\alpha) = \sigma'(\alpha)$$

Lemma A.4.12 (Basic Properties of $\rhd$)

The relation $\rhd$ is a pre-order — it is reflexive, transitive, and anti-symmetric.

Proof. Immediate from the definition of $\rhd$.

Lemma A.4.13 *For all σ and M there exist σ' and M' such that $\sigma' \rhd \sigma$ and $\sigma' M' = M$.*

Proof. By induction on M.

Step	Justification
Case x	
1. let $M' = x$	definition
2. let $\sigma' = \sigma$	definition
3. $\sigma' \rhd \sigma$	by 2, reflexivity of $\rhd$
4. $\sigma' M' = x$	by 3, def. of eval.
Case $\Pi^\tau x{:}A.\, B$	
1. $\sigma_1 \rhd \sigma$	ind. hyp. on A
2. $\sigma_1 A' = A$	(for some σ_1, A')
3. $\sigma_2 \rhd \sigma_1$	ind. hyp. on B
4. $\sigma_2 B' = B$	(for some σ_2, B')
5. $\alpha \notin dom(\sigma_2)$	for some fresh α
6. let $\sigma_3 = \sigma_2\{\alpha := \tau\}$	definition

Step		Justification
7.	$\sigma_3 \rhd \sigma_2$	by 5, 6
8.	$\sigma_3(\alpha) = \tau$	by 7
9.	$\sigma_3 \rhd \sigma$	7, 3, 1, transitivity of $\rhd$
10.	$\sigma_3(\Pi^\alpha x{:}A'.\, B') = \Pi^\tau x{:}A.\, B$	by 8, 2, 3, 4, 7
all other cases are similar to these two		

□

Theorem A.4.14 (Completeness of EPTS$^{\mathcal{C}}$ typing rules)

$$\frac{\sigma\Gamma \vdash \sigma M :^{\sigma\rho} \sigma A}{(\exists \mathcal{C}, \sigma') \quad \mathcal{C}\,;\, \Gamma \vdash M :^{\rho} A \quad \wedge \quad \sigma' \vDash \mathcal{C} \quad \wedge \quad \sigma' \rhd \sigma}$$

Note. One possible proof of this theorem is by induction on the derivation of $\sigma\Gamma \vdash \sigma M :^{\sigma\rho} \sigma A$. Each case of this proof proceeds by cases by whether or not ρ is an annotation variable α. If $\rho = \alpha$, some additional wrapper logic is required around the primary reasoning for that case of the proof. Viewing the proof as a functional program that manipulates derivations, it is good practice to abstract out this repeated wrapper logic into an auxiliary function.

This step requires care, however, as the auxiliary function G does sometimes calls the main function F on sub-derivations of its input that are not proper (i.e., a call to $G(\mathcal{X})$ results in a call to $F(\mathcal{X})$ on the same derivation $\mathcal{X}$). For this reason, if F were to call G on something that is not a proper sub-derivation of F's argument, we have the possibility that some calls to F and G may not terminate, meaning that they do not represent valid proofs. Fortunately F only calls G on proper sub-derivations of its argument, so every recursive call from G to G via F happens on a structurally smaller value than G's original argument.

For reasons stated above, we may restate Theorem A.4.14 as the conjunction

of the following two lemmas (named F and G as in the previous paragraph):

$$F :: \left(\frac{\sigma\Gamma \vdash \sigma M :^{\tau} \sigma A}{(\exists \mathcal{C}, \sigma') \quad \mathcal{C}\,;\, \Gamma \vdash M :^{\tau} A \quad \wedge \quad \sigma' \vDash \mathcal{C} \quad \wedge \quad \sigma' \rhd \sigma} \right)$$

$$G :: \left(\frac{\sigma\Gamma \vdash \sigma M :^{\sigma\alpha} \sigma A}{(\exists \mathcal{C}, \sigma') \quad \mathcal{C}\,;\, \Gamma \vdash M :^{\alpha} A \quad \wedge \quad \sigma' \vDash \mathcal{C} \quad \wedge \quad \sigma' \rhd \sigma} \right)$$

The proof of F is by induction on the derivation of the typing judgment above the line. The proof of G is by cases on whether $\sigma(\alpha)$ equals r or c. The proof of F may appeal to G only on proper sub-derivations of its input derivation while G may appeal to F on any (not necessarily proper) sub-derivation of its input derivation.

Proof of G. By cases on $\sigma\alpha$.

Step		**Justification**
Case $\sigma\alpha = \mathsf{c}$		
1.	$\sigma\Gamma \vdash \sigma M :^{\sigma\alpha} \sigma A$	assumption
2.	$\sigma\alpha = \mathsf{c}$	hypothesis
3.	$\sigma\Gamma \vdash \sigma M :^{\mathsf{c}} \sigma A$	by 1, 2
4.	$(\sigma\Gamma)^{\circ} \vdash \sigma M :^{\mathsf{r}} \sigma A$	by 2, inversion
5.	$(\sigma\Gamma)^{\circ} = \sigma(\Gamma^{\circ(\alpha)})$	by 2, Lemma A.4.1
6.	$\sigma(\Gamma^{\circ(\alpha)}) \vdash \sigma M :^{\mathsf{r}} \sigma A$	by 4, 5
7.	$\mathcal{C}\,;\, \Gamma^{\circ(\alpha)} \vdash M :^{\mathsf{r}} A$	by F, 6 (for some σ', $\mathcal{C}$)
8.	$\sigma' \vDash \mathcal{C}$	
9.	$\sigma' \rhd \sigma$	
10.	$\mathcal{C}\,;\, \Gamma \vdash M :^{\alpha} A$	by Reset, 7
Case $\sigma\alpha = \mathsf{r}$		
1.	$\sigma\Gamma \vdash \sigma M :^{\sigma\alpha} \sigma A$	assumption
2.	$\sigma\alpha = \mathsf{c}$	hypothesis
3.	$\sigma\Gamma \vdash \sigma M :^{\mathsf{r}} \sigma A$	assumption

Step		Justification
4.	$\sigma\Gamma = \sigma(\Gamma^{\circ(\alpha)})$	by 2, Lemma A.4.1
5.	$\sigma(\Gamma^{\circ(\alpha)}) \vdash \sigma M :^{\mathsf{r}} \sigma A$	by 3, 4
6.	$\mathcal{C}\,;\,\Gamma^{\circ(\alpha)} \vdash M :^{\mathsf{r}} A$	by F, 5 (for some σ', $\mathcal{C}$)
7.	$\sigma' \vDash \mathcal{C}$	
8.	$\sigma' \rhd \sigma$	
9.	$\mathcal{C}\,;\,\Gamma \vdash M :^{\alpha} A$	by Reset, 6

Proof of F. By induction on the derivation of $\sigma\Gamma \vdash \sigma M :^{\tau} \sigma A$.

Step		Justification
Case Axiom: $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\varepsilon \vdash s_1 :^{\mathsf{r}} s_2}\right)$		
1.	$(s_1, s_2) \in \mathcal{A}$	hypothesis
2.	$\sigma\Gamma = \varepsilon$	hypothesis
3.	$\sigma M = s_1$	hypothesis
4.	$\sigma A = s_2$	hypothesis
5.	$\Gamma = \varepsilon$	by 2
6.	$M = s_1$	by 3
7.	$A = s_2$	by 4
8.	$\mathsf{true}\,;\,\varepsilon \vdash s_1 :^{\mathsf{r}} s_2$	by Axiom, 1
9.	let $\mathcal{C} = \mathsf{true}$	definition
10.	let $\sigma' = \sigma$	definition
11.	$\sigma' \rhd \sigma$	by 10, reflexivity
12.	$\sigma' \vDash \mathcal{C}$	by 8
13.	$\mathcal{C}\,;\,\Gamma \vdash M :^{\mathsf{r}} A$	by 8, 9, 5, 6, 7
Case Var: $\left(\dfrac{\Delta \vdash B :^{\mathsf{c}} s}{\Delta, x{:}^{\mathsf{r}}B \vdash x :^{\mathsf{r}} B}\right)$		

Step		Justification
1.	$\Delta \vdash B :^{\mathsf{c}} s$	hypothesis
2.	$\sigma\Gamma = \Delta, x{:}^{\mathsf{r}}B$	hypothesis
3.	$\sigma M = x$	hypothesis
4.	$\sigma A = B$	hypothesis
5.	$\Gamma = \Delta', x{:}^{\gamma}B'$	} by 2 (for some Δ', γ, B')
6.	$\sigma\Delta' = \Delta$	
7.	$\sigma\gamma = \mathsf{r}$	
8.	$\sigma B' = B$	
9.	$\sigma s = s$	def. of eval.
10.	$\sigma\Delta' \vdash \sigma B' :^{\mathsf{c}} \sigma s$	by 1, 6, 8, 9
11.	$\mathcal{C}\,;\,\Delta' \vdash B' :^{\mathsf{c}} s$	} ind. hyp. on 10 (for some σ', $\mathcal{C}$)
12.	$\sigma' \rhd \sigma$	
13.	$\sigma' \vDash \mathcal{C}$	
14.	$\mathcal{C} \wedge \neg\gamma\,;\,\Delta', x{:}^{\gamma}B' \vdash x :^{\mathsf{r}} B'$	by 11, Var
15.	$\sigma B' = \sigma A$	by 4, 8
16.	$\mathcal{D} \vdash B' =_{\beta} A$	} Lemma A.4.6 on 15 (for some $\mathcal{D}$)
17.	$\sigma \vDash \mathcal{D}$	
18.	$\sigma'\Delta' \vdash \sigma' A :^{\mathsf{c}} \sigma' s$	by 1, 12, 6, 4, 9
19.	$\mathcal{E}\,;\,\Delta' \vdash A :^{\mathsf{c}} s$	} ind. hyp. on 18 (for some σ'', $\mathcal{E}$)
20.	$\sigma'' \rhd \sigma'$	
21.	$\sigma'' \vDash \mathcal{E}$	
22.	$\mathcal{C} \wedge \mathcal{E}\,;\,\Delta', x{:}^{\gamma}B' \vdash A :^{\mathsf{c}} s$	by Weak, 11, 19
23.	$\mathcal{C} \wedge \neg\gamma \wedge \mathcal{D} \wedge \mathcal{E}\,;\,\Delta', x{:}^{\gamma}B' \vdash x :^{\mathsf{r}} A$	by Conv, 14, 22, 16
24.	$M = x$	by 3
25.	$\mathcal{C} \wedge \neg\gamma \wedge \mathcal{D} \wedge \mathcal{E}\,;\,\Gamma \vdash M :^{\mathsf{r}} A$	by 23, 5, 24
26.	$\sigma'' \vDash \mathcal{C} \wedge \neg\gamma \wedge \mathcal{D} \wedge \mathcal{E}$	by 20, 12, 13, 7, 17, 21

Step		Justification
27.	$\sigma'' \rhd \sigma$	by 20, 12, transitivity

Case Weak: $\left(\dfrac{\Delta \vdash B :^{\mathsf{c}} s \qquad \Delta \vdash N :^{\mathsf{r}} C}{\Delta, x{:}^{\tau} B \vdash N :^{\mathsf{r}} C}\right)$

1.	$\sigma\Gamma = \Delta, x{:}^{\tau} B$	hypothesis
2.	$\sigma M = N$	hypothesis
3.	$\sigma A = C$	hypothesis
4.	$\Delta \vdash B :^{\mathsf{c}} s$	hypothesis
5.	$\Delta \vdash N :^{\mathsf{r}} C$	hypothesis
6.	$\Gamma = \Delta', x{:}^{\gamma} B'$	by 1 (for some Δ', γ, B')
7.	$\sigma\Delta' = \Delta$	
8.	$\sigma\gamma = \tau$	
9.	$\sigma B' = B$	
10.	$\sigma s = s$	def. of eval.
11.	$\sigma\Delta' \vdash \sigma B' :^{\mathsf{c}} \sigma s$	by 4, 7, 9, 10
12.	$\mathcal{C}\,;\, \Delta' \vdash B' :^{\mathsf{c}} s$	ind. hyp. on 11 (for some $\mathcal{C}$, σ')
13.	$\sigma' \rhd \sigma$	
14.	$\sigma' \vDash \mathcal{C}$	
15.	$\sigma'\Delta' \vdash \sigma' M :^{\mathsf{r}} \sigma' A$	by 5, 7, 2, 3, 13
16.	$\mathcal{D}\,;\, \Delta' \vdash M :^{\mathsf{r}} A$	ind. hyp. on 11 (for some $\mathcal{D}$, σ'')
17.	$\sigma'' \rhd \sigma'$	
18.	$\sigma'' \vDash \mathcal{D}$	
19.	$\mathcal{C} \wedge \mathcal{D}\,;\, \Delta', x{:}^{\gamma} B' \vdash M :^{\mathsf{r}} A$	by Weak, 12, 16
20.	$\mathcal{C} \wedge \mathcal{D}\,;\, \Gamma \vdash M :^{\mathsf{r}} A$	by 19, 6
21.	$\sigma'' \vDash \mathcal{C} \wedge \mathcal{D}$	by 18, 14, 17
22.	$\sigma'' \rhd \sigma$	by 17, 13, transitivity

Step	Justification

Case Π-Form: $\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Delta \vdash B :^{\mathsf{r}} s_1 \qquad \Delta, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2}{\Delta \vdash \Pi^{\tau} x{:}B.\, C :^{\mathsf{r}} s_3}\right)$

1.	$\sigma\Gamma = \Delta$	hypothesis
2.	$\sigma M = \Pi^{\tau} x{:}B.\, C$	hypothesis
3.	$\sigma A = s_3$	hypothesis
4.	$(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
5.	$\Delta \vdash B :^{\mathsf{r}} s_1$	hypothesis
6.	$\Delta, x{:}^{\mathsf{r}} B \vdash C :^{\mathsf{r}} s_2$	hypothesis
7.	$M = \Pi^{\alpha} x{:}B'.\, C'$	} by 2 (for some α, B', C')
8.	$\sigma\alpha = \tau$	
9.	$\sigma B' = B$	
10.	$\sigma C' = C$	
11.	$\sigma s_1 = s_1$	def. of eval.
12.	$\sigma\Gamma \vdash \sigma B' :^{\mathsf{r}} \sigma s_1$	by 5, 1, 9, 11
13.	$\mathcal{C}\,;\, \Gamma \vdash B' :^{\mathsf{r}} s_1$	} ind. hyp. on 12
14.	$\sigma' \rhd \sigma$	
15.	$\sigma' \vDash \mathcal{C}$	
16.	$\sigma'(\Gamma, x{:}^{\mathsf{r}} B') = \Delta, x{:}^{\mathsf{r}} B$	by 1, 9, 14
18.	$\sigma s_2 = s_2$	def. of eval.
19.	$\sigma'(\Gamma, x{:}^{\mathsf{r}} B') \vdash \sigma' C' :^{\mathsf{r}} \sigma' s_2$	by 6, 16, 10, 18, 14
20.	$\mathcal{D}\,;\, \Gamma, x{:}^{\mathsf{r}} B' \vdash C' :^{\mathsf{r}} s_2$	} ind. hyp. on 19 (for some $\mathcal{D}$, σ'')
21.	$\sigma'' \vDash \mathcal{D}$	
22.	$\sigma'' \rhd \sigma'$	
23.	$\mathcal{C} \wedge \mathcal{D}\,;\, \Gamma \vdash \Pi^{\alpha} x{:}B'.\, C' :^{\mathsf{r}} s_3$	by Π-Form, 4, 13, 20
24.	$A = s_3$	by 3
24.	$\mathcal{C} \wedge \mathcal{D}\,;\, \Gamma \vdash M :^{\mathsf{r}} A$	by 23, 7, 24
25.	$\sigma'' \vDash \mathcal{C} \wedge \mathcal{D}$	by 15, 22, 21

Step		Justification
26.	$\sigma'' \rhd \sigma$	by 22, 14

Case Π-Intro: $\left(\dfrac{\Delta \vdash \Pi^{\tau} x{:}B.\, C :^{\mathsf{c}} s \qquad \Delta, x{:}^{\tau} B \vdash N :^{\mathsf{r}} C}{\Delta \vdash \lambda^{\tau} x{:}B.\, N :^{\mathsf{r}} \Pi^{\tau} x{:}B.\, C}\right)$

1.	$\sigma\Gamma = \Delta$	hypothesis
2.	$\sigma M = \lambda^{\tau} x{:}B.\, N$	hypothesis
3.	$\sigma A = \Pi^{\tau} x{:}B.\, C$	hypothesis
4.	$\Delta \vdash \Pi^{\tau} x{:}B.\, C :^{\mathsf{c}} s$	hypothesis
5.	$\Delta, x{:}^{\tau} B \vdash N :^{\mathsf{r}} C$	hypothesis
6.	$M = \lambda^{\alpha_1} x{:}B'.\, N'$	
7.	$\sigma\alpha_1 = \tau$	by 2
8.	$\sigma B' = B$	(for some α_1, B', N')
9.	$\sigma N' = N$	
10.	$A = \Pi^{\alpha_2} x{:}B''.\, C'$	
11.	$\sigma\alpha_2 = \tau$	by 3
12.	$\sigma B'' = B$	(for some α_2, B'', C')
13.	$\sigma C' = C$	
14.	$\sigma\Gamma \vdash \sigma(\Pi^{\alpha_2} x{:}B'.\, C') :^{\mathsf{c}} \sigma s$	by 4, 1, 11, 8, 13
15.	$\mathcal{C}\,;\, \Gamma \vdash \Pi^{\alpha_2} x{:}B'.\, C' :^{\mathsf{c}} s$	
16.	$\sigma' \vDash \mathcal{C}$	ind. hyp. on 14
17.	$\sigma' \rhd \sigma$	(for some $\mathcal{C}$, σ')
18.	$\sigma(\Gamma, x{:}^{\alpha_1} B') = \Delta, x{:}^{\tau} B$	by 1, 7, 8
19.	$\sigma'(\Gamma, x{:}^{\alpha_1} B') \vdash \sigma' N' :^{\mathsf{r}} \sigma' C'$	by 5, 18, 9, 13, 17
20.	$\mathcal{D}\,;\, \Gamma, x{:}^{\alpha_1} B' \vdash N' :^{\mathsf{r}} C'$	
21.	$\sigma'' \vDash \mathcal{D}$	by ind. hyp. on 19
22.	$\sigma'' \rhd \sigma'$	(for some $\mathcal{D}$, σ'')
23.	$\sigma''\Gamma \vdash \sigma''(\Pi^{\alpha_2} x{:}B''.\, C') :^{\mathsf{c}} \sigma'' s$	by 4, 1, 11, 12, 13, 22, 17

Step		Justification
24.	$\mathcal{E}\,;\,\Gamma \vdash \Pi^{\alpha_2} x{:}B''.\,C' :^{\mathsf{c}} s$	
25.	$\sigma''' \vDash \mathcal{E}$	ind. hyp. on 23
26.	$\sigma''' \rhd \sigma''$	(for some $\mathcal{E}$, σ''')
27.	$\sigma B' = \sigma B''$	by 8, 12
28.	$\sigma(\Pi^{\alpha_2} x{:}B'.\,C') = \sigma(\Pi^{\alpha_2} x{:}B''.\,C')$	by 27
29.	$\mathcal{F} \vdash \Pi^{\alpha_2} x{:}B'.\,C' =_\beta \Pi^{\alpha_2} x{:}B''.\,C'$	Lemma A.4.6 on 27
30.	$\sigma \vDash \mathcal{F}$	(for some $\mathcal{F}$)
31.	$\mathcal{C} \wedge \mathcal{D} \wedge \alpha_1 = \alpha_2; \Gamma \vdash \lambda^{\alpha_1} x{:}B'.\,N' :^{\mathsf{r}} \Pi^{\alpha_2} x{:}B'.\,C'$	Π-Intro on 15, 20
32.	$\mathcal{C} \wedge \mathcal{D} \wedge \alpha_1 = \alpha_2 \wedge \mathcal{E} \wedge \mathcal{F}; \Gamma \vdash \lambda^{\alpha_1} x{:}B'.\,N' :^{\mathsf{r}} \Pi^{\alpha_2} x{:}B''.\,C'$	Conv on 31, 24, 29
33.	let $\mathcal{G} = \mathcal{C} \wedge \mathcal{D} \wedge \alpha_1 = \alpha_2 \wedge \mathcal{E} \wedge \mathcal{F}$	definition
34.	$\mathcal{G}\,;\,\Gamma \vdash M :^{\mathsf{r}} A$	by 32, 33, 6, 10
35.	$\sigma''' \vDash \mathcal{G}$	by 33, 16, 21, 7, 11, 25, 30, 26, 22, 17
36.	$\sigma''' \rhd \sigma$	by 26, 22, 17, transitivity

Case Π-Elim: $\left(\dfrac{\Delta \vdash N :^{\mathsf{r}} \Pi^{\tau} x{:}B.\,C \qquad \Delta \vdash P :^{\tau} B}{\Delta \vdash N@^{\tau} P :^{\mathsf{r}} C[P/x]}\right)$

1.	$\sigma\Gamma = \Delta$	hypothesis
2.	$\sigma M = N@^{\tau} P$	hypothesis
3.	$\sigma A = C[P/x]$	hypothesis
4.	$\Delta \vdash N :^{\mathsf{r}} \Pi^{\tau} x{:}B.\,C$	hypothesis
5.	$\Delta \vdash P :^{\tau} B$	hypothesis
6.	$M = N'@^{\alpha_1} P'$	by 2
7.	$\sigma N' = N$	(for some N', α_1, P')
8.	$\sigma\alpha_1 = \tau$	

Step		Justification
9.	$\sigma P' = P$	} (continued)
10.	$\sigma A = C[\sigma P'/x]$	by 3, 9
11.	$\sigma C' = C$	} Lemma A.4.7 on 10 (for some $\mathcal{C}$, A')
12.	$\mathcal{C} \vdash A =_\beta C'[P'/x]$	
13.	$\sigma \vDash \mathcal{C}$	
14.	$\sigma_1 B' = B$	} Lemma A.4.13 (for some B', σ_1)
15.	$\sigma_1 \rhd \sigma$	
16.	let $\sigma_2 = \sigma_1\{\alpha_2 = \tau\}$	} definition for some fresh α_2
17.	$\sigma_2 \rhd \sigma_1$	
18.	$\sigma_2\Gamma \vdash \sigma_2 N' :^{\mathsf{r}} \sigma_2(\Pi^{\alpha_2} x{:}B'.\, C')$	by 4, 1, 7, 16, 14, 11
19.	$\mathcal{D}\,;\, \Gamma \vdash N' :^{\mathsf{r}} \Pi^{\alpha_2} x{:}B'.\, C'$	} ind. hyp. on 18 (for some $\mathcal{D}$)
20.	$\sigma_3 \vDash \mathcal{D}$	
21.	$\sigma_3 \rhd \sigma_2$	
22.	$\sigma_3\Gamma \vdash \sigma_3 P' :^{\sigma_3\alpha_1} \sigma_3 B'$	by 5, 1, 9, 16, 14, 11
23.	$\mathcal{E}\,;\, \Gamma \vdash P' :^{\alpha_1} B'$	} by G on 22 (for some $\mathcal{E}$, σ_4)
24.	$\sigma_4 \vDash \mathcal{E}$	
25.	$\sigma_4 \rhd \sigma_3$	
26.	$\mathcal{D} \wedge \mathcal{E} \wedge \alpha_1 = \alpha_2; \Gamma \vdash N'@^{\alpha_1} P' :^{\mathsf{r}} C'[P'/x]$	Π-Elim on 19, 23
27.	$\sigma_4\Gamma \vdash \sigma_4 A :^{\mathsf{c}} \sigma_4 s$	by Coherence[2]
28.	$\mathcal{F}\,;\, \Gamma \vdash A :^{\mathsf{c}} s$	} by ind. hyp. on 27 (for some $\mathcal{F}$, σ_5)
29.	$\sigma_5 \vDash \mathcal{F}$	
30.	$\sigma_5 \rhd \sigma_4$	
31.	$\mathcal{D} \wedge \mathcal{E} \wedge \alpha_1 = \alpha_2 \wedge \mathcal{F} \wedge \mathcal{C}; \Gamma \vdash$	

[2]We are actually working with an EPTS variant in which the Π-Elim rule contains the premise $\Gamma \vdash C[P/x] :^{\mathsf{c}} s$. Such a premise is required here in order to validate the appeal to the induction hypothesis in steps 28–30. The coherence theorem shows that this variant of EPTS is no stronger than the original EPTS.

Step		Justification
	$M :^{\mathsf{r}} A$	by Conv on 26, 28, 12, 6
32.	$\sigma_5 \vDash \mathcal{D} \wedge \mathcal{E} \wedge \alpha_1 = \alpha_2 \wedge \mathcal{F} \wedge \mathcal{C}$	by 20, 24, 8, 16, 29, 13, 30, 25, 21, 17, 15
33.	$\sigma_5 \rhd \sigma$	by 30, 25, 21, 17, 15

Case Conv: $\left(\dfrac{\Delta \vdash N :^{\mathsf{r}} B \qquad \Delta \vdash C :^{\mathsf{c}} s \qquad B =_\beta C}{\Delta \vdash N :^{\mathsf{r}} C} \right)$

1.	$\sigma\Gamma = \Delta$	hypothesis
2.	$\sigma M = N$	hypothesis
3.	$\sigma A = C$	hypothesis
4.	$\Delta \vdash N :^{\mathsf{r}} B$	hypothesis
5.	$\Delta \vdash C :^{\mathsf{c}} s$	hypothesis
6.	$B =_\beta C$	hypothesis
7.	$\sigma' B' = B$	Lemma A.4.13 (for some B', σ')
8.	$\sigma' \rhd \sigma$	
9.	$\sigma'\Gamma \vdash \sigma' M :^{\mathsf{r}} \sigma' B'$	by 4, 1, 2, 7, 8
10.	$\mathcal{C}\,;\, \Gamma \vdash M :^{\mathsf{r}} B'$	ind. hyp. on 9 (for some $\mathcal{C}$, σ'')
11.	$\sigma'' \vDash \mathcal{C}$	
12.	$\sigma'' \rhd \sigma'$	
13.	$\sigma'' s = s$	def. of eval.
14.	$\sigma''\Gamma \vdash \sigma'' A :^{\mathsf{c}} \sigma'' s$	by 5, 2, 13, 12, 8
15.	$\mathcal{D}\,;\, \Gamma \vdash A :^{\mathsf{c}} s$	ind. hyp. on 9 (for some $\mathcal{D}$, σ''')
16.	$\sigma''' \vDash \mathcal{D}$	
17.	$\sigma''' \rhd \sigma''$	
18.	$\sigma''' B' =_\beta \sigma''' A$	by 6, 3, 7, 8, 12, 17
19.	$\mathcal{E} \vdash B' =_\beta A$	Theorem A.4.9 on 18 (for some $\mathcal{D}$)
20.	$\sigma''' \vDash \mathcal{E}$	

Step		Justification
21.	$\mathcal{C} \wedge \mathcal{D} \wedge \mathcal{E} \,;\, \Gamma \vdash M :^{\mathsf{r}} A$	by Conv, 10, 15, 19
22.	$\sigma''' \vDash \mathcal{C} \wedge \mathcal{D} \wedge \mathcal{E}$	by 11, 17, 16, 20
23.	$\sigma''' \rhd \sigma$	by 17, 12, 8
Case Reset:	$\left(\dfrac{\Delta^\circ \vdash N :^{\mathsf{r}} B}{\Delta \vdash N :^{\mathsf{c}} B}\right)$	
1.	$\sigma\Gamma = \Delta$	hypothesis
2.	$\sigma M = N$	hypothesis
3.	$\sigma A = B$	hypothesis
4.	$\Delta^\circ \vdash N :^{\mathsf{r}} B$	hypothesis
5.	$(\sigma\Gamma)^\circ = \sigma(\Gamma^{\circ(\mathsf{c})})$	by Lemma A.4.1
6.	$\sigma(\Gamma^{\circ(\mathsf{c})}) \vdash \sigma M :^{\mathsf{r}} \sigma A$	by 4, 1, 5, 2, 3
7.	$\mathcal{C} \,;\, \Gamma^{\circ(\mathsf{c})} \vdash M :^{\mathsf{r}} A$	} ind. hyp. on 6 (for some $\mathcal{C}$, σ')
8.	$\sigma' \vDash \mathcal{C}$	
9.	$\sigma' \rhd \sigma$	
10.	$\mathcal{C} \,;\, \Gamma \vdash M :^{\mathsf{c}} A$	by Reset, 7

□

A.5 META-THEORY OF EPTS•

Lemma A.5.1

$$\frac{\vdash \Gamma \ \mathsf{ctx}}{\vdash \Gamma^\circ \ \mathsf{ctx}}$$

Proof. By induction on Γ.

Case	Step		Justification
ε	1.	$\varepsilon^\circ = \varepsilon$	definition of $\circ$
	2.	$\vdash \varepsilon \ \mathsf{ctx}$	OkNil

	3.	$\vdash \varepsilon^\circ$ ctx	by 1, 2
$\Gamma, x{:}^\tau A$	1.	$\vdash \Gamma$ ctx	hypothesis
	2.	$\Gamma \vdash A :^{\mathsf{c}} s$	hypothesis
	3.	$\vdash \Gamma^\circ$ ctx	induction hypothesis on 1
	4.	$\Gamma^\circ \vdash A :^{\mathsf{c}} s$	by Lemma 3.1.3, 2
	5.	$\vdash \Gamma^\circ, x{:}^{\mathsf{r}} A$ ctx	by OkExt, 3, 4
	6.	$(\Gamma, x{:}^\tau A)^\circ = \Gamma^\circ, x{:}^{\mathsf{r}} A$	definition of $\circ$
	7.	$\vdash (\Gamma, x{:}^\tau A)^\circ$ ctx	by 5, 6

□

Theorem A.5.2 (Elaboration in r mode)

$$\frac{\vdash \Gamma \text{ ctx} \qquad \Gamma^\bullet \vdash M : A \qquad FV(M) \subseteq RV(\Gamma)}{(\exists\, M'\, A') \qquad \Gamma \vdash M' :^{r} A' \qquad M'^\bullet = M \qquad A'^\bullet = A}$$

Note. While proving the main theorem, we often use the following chain of reasoning over and over again after invoking an induction hypothesis. Therefore the proof is streamlined by factoring out the following corollary, which we may use in any place the induction hypothesis is permitted (on structurally smaller sub-derivations).

Corollary A.5.3 (Elaboration in c mode)

$$\frac{\vdash \Gamma\ \mathsf{ctx} \qquad \Gamma^\bullet \vdash M : A}{(\exists\, M'\, A') \qquad \Gamma \vdash M' :^{\mathsf{c}} A' \qquad M'^\bullet = M \qquad A'^\bullet = A}$$

Proof. Because $\Gamma^\bullet = \Gamma^{\circ\bullet}$ (see Lemma A.2.8), we have $\Gamma^{\circ\bullet} \vdash M : A$. Also $\vdash \Gamma^\circ\ \mathsf{ctx}$ follows from $\vdash \Gamma\ \mathsf{ctx}$ by Lemma A.5.1 and $FV(M) \subseteq CV(\Gamma) = RV(\Gamma^\circ)$. Therefore, we can apply the theorem to obtain $\Gamma^\circ \vdash M' :^{\mathsf{r}} A'$ and $M'^\bullet = M$ and $A'^\bullet = A$. for some M' and A'. By RESET, we obtain $\Gamma \vdash M' :^{\mathsf{c}} A'$. □

Proof of Theorem A.5.2. By induction on the derivation of $\Gamma^\bullet \vdash M : A$.

Step	**Justification**
Case AXIOM: $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\varepsilon \vdash s_1 : s_2}\right)$	
1. $\Gamma^\bullet = \varepsilon$	assumption
2. $(s_1, s_2) \in \mathcal{A}$	hypothesis
3. let $M' = s_1$	definition
4. let $A' = s_2$	definition
5. $\Gamma = \varepsilon$	by 1, def. of $\bullet$
6. $\varepsilon \vdash s_1 :^{\mathsf{r}} s_2$	by AXIOM, 1
7. $\Gamma \vdash M' :^{\mathsf{r}} A'$	by 6, 5, 3, 4
8. ${s_1}^\bullet = s_1$	def. of $\bullet$
9. $M'^\bullet = M$	by 8, 3, 4
10. ${s_2}^\bullet = s_2$	def. of $\bullet$
11. $A'^\bullet = A$	by 10, 3, 4
12. $\Gamma \vdash M' :^{\mathsf{r}} A' \ \wedge$ $M'^\bullet = M \ \wedge\ A'^\bullet = A$	by 7, 9, 11
Case VAR: $\left(\dfrac{\Delta \vdash A : s}{\Delta, x{:}A \vdash x : A}\right)$	

Step		Justification
1.	$FV(x) \subseteq RV(\Gamma)$	assumption
2.	$\vdash \Gamma$ ctx	assumption
3.	$\Gamma^\bullet = \Delta, x{:}A$	hypothesis
4.	$\Delta \vdash A : s$	hypothesis
5.	$x \in RV(\Gamma)$	by 1, def. of FV
6.	$\Gamma = \Delta', x{:}^\tau A'$	by 3 (for some Δ', τ, A')
7.	$\Delta'^\bullet = \Delta$	
8.	$A'^\bullet = A$	
9.	$\tau = \mathsf{r}$	by 5, 6, def. of RV
10.	$\vdash \Delta'$ ctx	by 2, 6 (for some s')
11.	$\Delta' \vdash A' :^{\mathsf{c}} s'$	
12.	$\Delta', x{:}^{\mathsf{r}} A' \vdash x :^{\mathsf{r}} A'$	by Var, 11
13.	let $M' = x$	definition
14.	$x^\bullet = x$	def. of $\bullet$
14.	$\Gamma = \Delta', M'{:}^{\mathsf{r}} A' \wedge$ $M'^\bullet = x \ \wedge \ A'^\bullet = A$	by 6, 9, 13, 14, 8

Case Weak: $\left(\dfrac{\Delta \vdash A : s \qquad \Delta \vdash M : B}{\Delta, x{:}A \vdash M : B} \right)$

1.	$\vdash \Gamma$ ctx	assumption
2.	$FV(M) \subseteq RV(\Gamma)$	assumption
3.	$\Gamma^\bullet = \Delta, x{:}A$	hypothesis
4.	$\Delta \vdash A : s$	hypothesis
5.	$\Delta \vdash M : B$	hypothesis
6.	$\Gamma = \Delta', x{:}^\tau A'$	by 3 (for some Δ', τ, A')
7.	$\Delta'^\bullet = \Delta$	
8.	$A'^\bullet = A$	

Step		Justification
9.	$\vdash \Delta'$ ctx	by 1, 6
10.	$\Delta' \vdash A' :^{\mathsf{c}} s'$	(for some s')
11.	$\Delta' \vdash M' :^{\mathsf{r}} B'$	ind. hyp. on 5, 7, 9, 2
12.	$M'^{\bullet} = M$	
13.	$B'^{\bullet} = B$	(for some M', B')
14.	$\Delta', x{:}^{\tau} A' \vdash M' :^{\mathsf{r}} B'$	by Weak, 10, 11
15.	$\Gamma \vdash M' :^{\mathsf{r}} B' \wedge$ $M'^{\bullet} = M \ \wedge \ B'^{\bullet} = B$	by 14, 6, 12, 13

Case Π-Form: $\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Delta \vdash A : s_1 \qquad \Delta, x{:}A \vdash B : s_2}{\Delta \vdash \Pi x{:}A.\, B : s_3} \right)$

1.	$\vdash \Gamma$ ctx	assumption
2.	$FV(\Pi x{:}A.\, B) \subseteq RV(\Gamma)$	assumption
3.	$\Gamma^{\bullet} = \Delta$	hypothesis
4.	$(s_1, s_2, s_3) \in \mathcal{R}$	hypothesis
5.	$\Delta \vdash A : s_1$	hypothesis
6.	$\Delta, x{:}A \vdash B : s_2$	hypothesis
7.	$FV(\Pi x{:}A.\, B)$ $= FV(A) \cup (FV(B) - \{x\})$	def. of FV
8.	$FV(A) \subseteq RV(\Gamma)$	by 2, 7
9.	$\Gamma \vdash A' :^{\mathsf{r}} s_1'$	ind. hyp. on 5, 3, 1, 8
10.	$A'^{\bullet} = A$	
11.	$s_1'^{\bullet} = s_1$	(for some A', s_1')
12.	$\Gamma \vdash A' :^{\mathsf{c}} s_1'$	phase weakening on 9
13.	$\vdash \Gamma, x{:}^{\mathsf{r}} A'$ ctx	by 1, 12
14.	$(\Gamma, x{:}^{\mathsf{r}} A')^{\bullet} = \Delta, x{:}A$	by 3, 10, def. of $\bullet$
15.	$FV(B) - \{x\} \subseteq RV(\Gamma)$	by 7, 2

Step		Justification
16.	$FV(B) \subseteq RV(\Gamma) \cup \{x\}$	by 15
17.	$RV(\Gamma) \cup \{x\} = RV(\Gamma, x{:}^{\mathsf{r}} A')$	by 15
18.	$FV(B) \subseteq RV(\Gamma, x{:}^{\mathsf{r}} A')$	by 16, 17
19.	$\Gamma, x{:}^{\mathsf{r}} A' \vdash B' :^{\mathsf{r}} s_2'$	ind. hyp. on 6, 14, 13, 18 (for some B', s_2')
20.	$B'^{\bullet} = B$	
21.	$s_2'^{\bullet} = s_2$	
22.	$s_1' = s_1$	by 11, def. of $\bullet$
23.	$s_2' = s_2$	by 21, def. of $\bullet$
24.	$\Gamma \vdash \Pi^{\mathsf{r}} x{:}A'.\, B' :^{\mathsf{r}} s_3$	Π-Form on 4, 22, 9, 23, 19
25.	$(\Pi^{\mathsf{r}} x{:}A'.\, B')^{\bullet} = \Pi x{:}A.\, B$	by def. of $\bullet$, 10, 20
26.	$\Gamma \vdash \Pi^{\mathsf{r}} x{:}A'.\, B' :^{\mathsf{r}} s_3$ $\wedge$ $(\Pi^{\mathsf{r}} x{:}A'.\, B')^{\bullet} = \Pi x{:}A.\, B$ $\wedge$ $s_3{}^{\bullet} = s_3$	by 24, 25, def. of $\bullet$

Case $\forall$-Form: $\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \forall x{:}A.\, B : s_3}\right)$

similar to the Π-Form case

Case Π-Intro: $\left(\dfrac{\Delta \vdash \Pi x{:}A.\, B : s \qquad \Delta, x{:}A \vdash M : B}{\Delta \vdash \lambda x.\, M : \Pi x{:}A.\, B}\right)$

1.	$\vdash \Gamma$ ctx	assumption
2.	$FV(\lambda x.\, M) \subseteq RV(\Gamma)$	assumption
3.	$\Gamma^{\bullet} = \Delta$	hypothesis
4.	$\Delta \vdash \Pi x{:}A.\, B : s$	hypothesis
5.	$\Delta, x{:}A \vdash M : B$	hypothesis
6.	$\Gamma \vdash C :^{\mathsf{c}} s'$	by Corollary A.5.3 on 4, 1, 2 (for some C, s')
7.	$C^{\bullet} = \Pi x{:}A.\, B$	
8.	$s'^{\bullet} = s$	

Step		Justification
9.	$C = \Pi^{\mathsf{r}} x{:}A'.\, B'$	}
10.	$A'^{\bullet} = A$	} by 7
11.	$B'^{\bullet} = B$	} (for some A', B')
12.	$\Gamma \vdash A' :^{\mathsf{c}} s_1'$	} by Corollary A.1.8
13.	$\Gamma, x{:}^{\mathsf{r}} A' \vdash B' :^{\mathsf{c}} s_2'$	} on 6, 9
14.	$\vdash \Gamma, x{:}^{\mathsf{r}} A'$ ctx	by 1, 12
15.	$(\Gamma, x{:}^{\mathsf{r}} A')^{\bullet} = \Delta, x{:}A$	by 3, 10, def. of $\bullet$
16.	$FV(\lambda x.\, M) = FV(M) - \{x\}$	def. of FV
17.	$FV(M) - \{x\} \subseteq RV(\Gamma)$	by 2, 16
18.	$FV(M) \subseteq RV(\Gamma) \cup \{x\}$	by 17
19.	$RV(\Gamma, x{:}^{\mathsf{r}} A') = RV(\Gamma) \cup \{x\}$	def. of RV
20.	$FV(M) \subseteq RV(\Gamma, x{:}^{\mathsf{r}} A')$	by 18, 19
21.	$\Gamma, x{:}^{\mathsf{r}} A' \vdash M' :^{\mathsf{r}} B''$	} ind. hyp. on
22.	$M'^{\bullet} = M$	} 5, 15, 14, 20
23.	$B''^{\bullet} = B$	} (for some M', B'')
24.	$B''^{\bullet} = B'^{\bullet}$	by 11, 23
25.	$B''^{\bullet} =_\beta B'^{\bullet}$	by 24, reflexivity of $=_\beta$
26.	$\Gamma, x{:}^{\mathsf{r}} A' \vdash M' :^{\mathsf{r}} B'$	by Conv on 21, 13, 25
27.	$\Gamma \vdash \lambda^{\mathsf{r}} x{:}A'.\, M' :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}A'.\, B'$	by Π-Intro on 6, 9, 26
28.	$(\lambda^{\mathsf{r}} x{:}A'.\, M')^{\bullet} = \lambda x.\, M$	by def. of $\bullet$, 22
29.	$(\Pi^{\mathsf{r}} x{:}A'.\, B')^{\bullet} = \Pi x{:}A.\, B$	by def. of $\bullet$, 10, 11
30.	$\Gamma \vdash \lambda^{\mathsf{r}} x{:}A'.\, M' :^{\mathsf{r}} \Pi^{\mathsf{r}} x{:}A'.\, B'$	
	$\wedge\ (\lambda^{\mathsf{r}} x{:}A'.\, M')^{\bullet} = \lambda x.\, M$	
	$\wedge\ (\Pi^{\mathsf{r}} x{:}A'.\, B')^{\bullet} = \Pi x{:}A.\, B$	by 27, 28, 29

Case $\forall$-Intro: $\left(\dfrac{\Delta \vdash \forall x{:}A.\, B : s \qquad \Delta, x{:}A \vdash M : B \qquad x \notin FV(M)}{\Delta \vdash M : \forall x{:}A.\, B} \right)$

Step		Justification
1.	$\vdash \Gamma$ ctx	assumption
2.	$FV(M) \subseteq RV(\Gamma)$	assumption
3.	$\Gamma^\bullet = \Delta$	hypothesis
4.	$\Delta \vdash \forall x{:}A.\, B : s$	hypothesis
5.	$\Delta, x{:}A \vdash M : B$	hypothesis
6.	$x \notin FV(M)$	hypothesis
7.	$\Gamma \vdash C :^{\mathsf{c}} s'$	by Corollary A.5.3
8.	$C^\bullet = \forall x{:}A.\, B$	on 4, 1, 2
9.	$s'^\bullet = s$	(for some C, s')
10.	$C = \Pi^{\mathsf{c}} x{:}A'.\, B'$	
11.	$A'^\bullet = A$	by 8
12.	$B'^\bullet = B$	(for some A', B')
13.	$\Gamma \vdash A' :^{\mathsf{c}} s'_1$	by Corollary A.1.8
14.	$\Gamma, x{:}^{\mathsf{c}}A' \vdash B' :^{\mathsf{c}} s'_2$	on 7, 10
15.	$\vdash \Gamma, x{:}^{\mathsf{c}}A'$ ctx	by 1, 13
16.	$(\Gamma, x{:}^{\mathsf{c}}A')^\bullet = \Delta, x{:}A$	by 3, 11, def. of $\bullet$
17.	$RV(\Gamma, x{:}^{\mathsf{c}}A') = RV(\Gamma)$	def. of RV
18.	$FV(M) \subseteq RV(\Gamma, x{:}^{\mathsf{c}}A')$	by 2, 17
19.	$\Gamma, x{:}^{\mathsf{c}}A' \vdash M' :^{\mathsf{r}} B''$	ind. hyp. on
20.	$M'^\bullet = M$	5, 16, 15, 18
21.	$B''^\bullet = B$	(for some M', B'')
22.	$B''^\bullet = B'^\bullet$	by 12, 21
23.	$B''^\bullet =_\beta B'^\bullet$	by 22, reflexivity of $=_\beta$
24.	$\Gamma, x{:}^{\mathsf{c}}A' \vdash M' :^{\mathsf{r}} B'$	by Conv on 19, 14, 23
25.	$\Gamma \vdash \lambda^{\mathsf{c}} x{:}A'.\, M' :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}A'.\, B'$	by Π-Intro on 7, 10, 24
26.	$(\lambda^{\mathsf{c}} x{:}A'.\, M')^\bullet = M$	by def. of $\bullet$, 20
27.	$(\Pi^{\mathsf{c}} x{:}A'.\, B')^\bullet = \forall x{:}A.\, B$	by def. of $\bullet$, 11, 12

Step		Justification
28.	$\Gamma \vdash \lambda^{\mathsf{c}} x{:}A'.\, M' :^{\mathsf{r}} \Pi^{\mathsf{c}} x{:}A'.\, B'$	
	$\wedge\ (\lambda^{\mathsf{c}} x{:}A'.\, M')^{\bullet} = M$	
	$\wedge\ (\Pi^{\mathsf{c}} x{:}A'.\, B')^{\bullet} = \forall x{:}A.\, B$	by 25, 26, 27

Case Π-Elim: $\left(\dfrac{\Delta \vdash M : \Pi x{:}A.\, B \qquad \Delta \vdash N : A}{\Delta \vdash M\ N : B[N/x]}\right)$

1.	$\vdash \Gamma\ \mathsf{ctx}$	assumption
2.	$FV(M\ N) \subseteq RV(\Gamma)$	assumption
3.	$\Gamma^{\bullet} = \Delta$	hypothesis
4.	$\Delta \vdash M : \Pi x{:}A.\, B$	hypothesis
5.	$\Delta \vdash N : A$	hypothesis
6.	$FV(M\ N) = FV(M) \cup FV(N)$	by def. of FV
7.	$FV(M) \subseteq RV(\Gamma)$	by 2, 6
8.	$FV(N) \subseteq RV(\Gamma)$	by 2, 6
9.	$\Gamma \vdash M' :^{\mathsf{r}} C$	by ind. hyp.
10.	$M'^{\bullet} = M$	on 4, 3, 1, 7
11.	$C^{\bullet} = \Pi x{:}A.\, B$	(for some M', C)
12.	$C = \Pi^{\mathsf{r}} x{:}A'.\, B'$	
13.	$A'^{\bullet} = A$	by 11
14.	$B'^{\bullet} = B$	(for some A', B')
15.	$\Gamma \vdash N' :^{\mathsf{r}} A''$	by ind. hyp.
16.	$N'^{\bullet} = N$	on 5, 3, 1, 8
17.	$A''^{\bullet} = A$	(for some N', A'')
18.	$A''^{\bullet} = A'^{\bullet}$	by 17, 13
19.	$A''^{\bullet} =_{\beta} A'^{\bullet}$	by 18, reflexivity of $=_{\beta}$
20.	$\Gamma \vdash \Pi^{\mathsf{r}} x{:}A'.\, B' :^{\mathsf{c}} s_3$	Coherence on 9, 12
21.	$\Gamma \vdash A' :^{\mathsf{c}} s_1$	Corollary A.1.8 on 20

Step		Justification
22.	$\Gamma \vdash N' :^{\mathsf{r}} A'$	Conv on 15, 21, 19
23.	$\Gamma \vdash M@^{\mathsf{r}}N' :^{\mathsf{r}} B'[N'/x]$	Π-Elim on 9, 12, 22
24.	$(M'@^{\mathsf{r}}N')^\bullet = M\ N$	by def. of $\bullet$, 10, 16
25.	$(B'[N'/x])^\bullet = B'^\bullet[N'^\bullet/x]$	by Lemma A.2.2
26.	$B'^\bullet[N'^\bullet/x] = B[N/x]$	by 14, 16
27.	$(B'[N'/x])^\bullet = B[N/x]$	by 25, 26
28.	$\Gamma \vdash M@^{\mathsf{r}}N' :^{\mathsf{r}} B'[N'/x]$	
	$\wedge\ (M'@^{\mathsf{r}}N')^\bullet = M\ N$	
	$\wedge\ (B'[N'/x])^\bullet = B[N/x]$	by 23, 24, 27

Case ∀-Elim: $\left(\dfrac{\Delta \vdash M : \forall x{:}A.\, B \qquad \Delta \vdash N : A}{\Delta \vdash M : B[N/x]}\right)$

1.	$\vdash \Gamma\ \mathsf{ctx}$	assumption
2.	$FV(M) \subseteq RV(\Gamma)$	assumption
3.	$\Gamma^\bullet = \Delta$	hypothesis
4.	$\Delta \vdash M : \forall x{:}A.\, B$	hypothesis
5.	$\Delta \vdash N : A$	hypothesis
6.	$\Gamma \vdash M' :^{\mathsf{r}} C$	by ind. hyp.
7.	$M'^\bullet = M$	on 4, 3, 1, 7
8.	$C^\bullet = \forall x{:}A.\, B$	(for some M', C)
9.	$C = \Pi^{\mathsf{c}} x{:}A'.\, B'$	
10.	$A'^\bullet = A$	by 8
11.	$B'^\bullet = B$	(for some A', B')
12.	$\Gamma \vdash N' :^{\mathsf{c}} A''$	by Corollary A.5.3
13.	$N'^\bullet = N$	on 5, 3, 1
14.	$A''^\bullet = A$	(for some N', A'')
15.	$A''^\bullet = A'^\bullet$	by 14, 10

Step		Justification
16.	$A''^{\bullet} =_\beta A'^{\bullet}$	by 15, reflexivity of $=_\beta$
17.	$\Gamma \vdash \Pi^{\mathsf{c}} x{:}A'.\, B' :^{\mathsf{c}} s_3$	Coherence on 6, 9
18.	$\Gamma \vdash A' :^{\mathsf{c}} s_1$	Corollary A.1.8 on 17
19.	$\Gamma^\circ \vdash N' :^{\mathsf{r}} A''$	by 12
20.	$\Gamma^\circ \vdash A' :^{\mathsf{c}} s_1$	Lemma A.1.2 on 18
21.	$\Gamma^\circ \vdash N' :^{\mathsf{r}} A'$	Conv$^\bullet$ on 19, 20, 16
22.	$\Gamma \vdash N' :^{\mathsf{c}} A'$	Reset on 21
23.	$\Gamma \vdash M@^{\mathsf{c}}N' :^{\mathsf{r}} B'[N'/x]$	$\forall$-Elim on 6, 9, 22
24.	$(M'@^{\mathsf{c}}N')^\bullet = M$	by def. of $\bullet$, 7, 13
25.	$(B'[N'/x])^\bullet = B'^{\bullet}[N'^{\bullet}/x]$	by Lemma A.2.2
26.	$B'^{\bullet}[N'^{\bullet}/x] = B[N/x]$	by 11, 13
27.	$(B'[N'/x])^\bullet = B[N/x]$	by 25, 26
28.	$\Gamma \vdash M@^{\mathsf{c}}N' :^{\mathsf{r}} B'[N'/x]$	
	$\wedge\ (M'@^{\mathsf{c}}N')^\bullet = M$	
	$\wedge\ (B'[N'/x])^\bullet = B[N/x]$	by 23, 24, 27

Case Conv: $\left(\dfrac{\Delta \vdash M : A \qquad \Delta \vdash B : s \qquad A =_\beta B}{\Delta \vdash M : B}\right)$

1.	$\vdash \Gamma$ ctx	assumption
2.	$FV(M) \subseteq RV(\Gamma)$	assumption
3.	$\Gamma^\bullet = \Delta$	hypothesis
4.	$\Delta \vdash M : A$	hypothesis
5.	$\Delta \vdash B : s$	hypothesis
6.	$A =_\beta B$	hypothesis
7.	$\Gamma \vdash M' :^{\mathsf{r}} A'$	by ind. hyp.
8.	$M'^{\bullet} = M$	on 4, 3, 1, 2
9.	$A'^{\bullet} = A$	(for some M', A')

Step		Justification
10.	$\Gamma \vdash B' :^{\mathsf{c}} s'$	by Corollary A.5.3
11.	$B'^{\bullet} = B$	on 4, 3, 1
12.	$s'^{\bullet} = s$	(for some M', A')
13.	$A'^{\bullet} =_{\beta} B'^{\bullet}$	by 6, 9, 11
14.	$\Gamma \vdash M' :^{\mathsf{r}} B'$	by Conv$^{\bullet}$ on 7, 10, 13
15.	$\Gamma \vdash M' :^{\mathsf{r}} B'$	
	$\wedge\ M'^{\bullet} = M\ \wedge\ B'^{\bullet} = B$	by 14, 8, 11

□

Lemma A.5.4 (Context Elaboration)

$$\frac{\Gamma \vdash M : A}{(\exists\, \Gamma') \qquad \vdash \Gamma'\ \mathsf{ctx} \qquad \Gamma'^{\bullet} = \Gamma}$$

Proof. By induction on the derivation of $\Gamma \vdash M : A$.

Step		Justification
Case Axiom: $\left(\dfrac{(s_1, s_2) \in \mathcal{A}}{\varepsilon \vdash s_1 : s_2}\right)$		
1.	$\vdash \varepsilon\ \mathsf{ctx}$	def. of ctx
2.	$\varepsilon^{\bullet} = \varepsilon$	def. of $\bullet$
3.	$\vdash \varepsilon\ \mathsf{ctx}\ \wedge\ \varepsilon^{\bullet} = \varepsilon$	by 1, 2
Case Var: $\left(\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}\right)$		
1.	$\Gamma \vdash A : s$	hypothesis
2.	$\vdash \Gamma'\ \mathsf{ctx}$	ind. hyp. on 1
3.	$\Gamma'^{\bullet} = \Gamma$	(for some Γ')
4.	$\Gamma' \vdash A' :^{\mathsf{c}} s'$	Lemma A.5.3 ...

Step	Justification
5. $A'^{\bullet} = A$	…on 1, 2, 3
6. $s'^{\bullet} = s$	(for some A', s')
7. $\vdash \Gamma', x{:}^{\tau} A'$ ctx	by 2, 4 (for an arbitrary τ)
8. $(\Gamma', x{:}^{\tau} A')^{\bullet} = \Gamma, x{:}A$	by 3, 5
9. $\vdash \Gamma', x{:}^{\tau} A'$ ctx $\wedge$ $(\Gamma', x{:}^{\tau} A')^{\bullet} = \Gamma, x{:}A$	by 7, 8

Case WEAK

similar to the VAR case

Case Π-FORM: $\left(\dfrac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.\, B : s_3}\right)$

1. $\Gamma \vdash A : s_1$	hypothesis
2. $\vdash \Gamma'$ ctx $\wedge$ $\Gamma'^{\bullet} = \Gamma$	ind. hyp. on 1 (for some Γ')

Cases $\forall$-FORM, Π-INTRO, $\forall$-INTRO, Π-ELIM, $\forall$-ELIM, and CONV

similar to the Π-FORM case

□

Corollary A.5.5 (Elaboration)

$$\frac{\Gamma \vdash M : A}{(\exists\, \Gamma'\, M'\, A') \qquad \Gamma' \vdash M' :^{\tau} A' \qquad \Gamma'^{\bullet} = \Gamma \qquad M'^{\bullet} = M \qquad A'^{\bullet} = A}$$

Proof. By Lemma A.5.4, we have $\vdash \Gamma'$ ctx for some Γ' such that $\Gamma'^{\bullet} = \Gamma$. Note from the proof of Lemma A.5.4 that the annotations on context entries in Γ' may be chosen in any way we choose (see the arbitrary τ in the VAR case). In particular, we may arrange things so that all the annotations on context entries for variables that appear free in M are set to r — In other words, that $FV(M) \subseteq RV(\Gamma')$. At this point Lemma A.5.2 applies, yielding $\Gamma' \vdash M' :^{\mathsf{r}} A'$. If $\tau = \mathsf{r}$ then we have

already proved $\Gamma' \vdash M' :^{\tau} A'$. If $\tau = \mathsf{c}$ then $\Gamma' \vdash M' :^{\mathsf{c}} A'$ (i.e., $\Gamma' \vdash M' :^{\tau} A'$) follows by phase weakening. In either case, we have $\Gamma' \vdash M' :^{\tau} A'$ for some Γ', M', and A' such that $\Gamma'^{\bullet} = \Gamma$, $M'^{\bullet} = M$, and $A'^{\bullet} = A$. $\square$