2022

A dependently typed programming language with dynamic equality

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

A DEPENDENTLY TYPED PROGRAMMING LANGUAGE WITH DYNAMIC EQUALITY

by

MARK LEMAY

B.S., Rochester Institute of Technology, 2010

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2022

Approved by

First Reader

Hongwei Xi, Ph.D.
Associate Professor of Computer Science

Second Reader

Assaf Kfoury, Ph.D.
Professor of Computer Science

Third Reader

Marco Gaboardi, Ph.D.
Associate Professor of Computer Science

Fourth Reader

Chris Casinghino, Ph.D.
Research Scientist
The Charles Stark Draper Laboratory, Inc.

What I mean is that if you really want to understand something, the best way is to try and explain it to someone else. That forces you to sort it out in your own mind. And the more slow and dim-witted your pupil, the more you have to break things down into more and more simple ideas. And that's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've certainly learned something about it yourself. The teacher usually learns more than the pupil. Isn't that true?

Douglas Adams, Dirk Gently's Holistic Detective Agency

Acknowledgments

First and foremost my fiancé Stephanie Savir, I could not have finished this without her support.

Boston University is an excellent school to learn Computer Science. It is so full of intelligent, passionate, and kind people that any list of people will feel incomplete. Given that I especially want to thank Dr.(!) Tomislav Petrovic his wife Ana and their adorably strong willed children for the longest friendship at BU; Qiancheng Fu, Cheng Zhang, and William Blair for their discussions, collaborations and friendship; and my advisor Dr. Hongwei Xi who has been more supportive than I would have thought possible. Also he is a genius. Thanks to Malavika Vishwanath of the Writing Assistance center for reviewing drafts of this thesis. Finally to the administrators (especially Kori MacDonald) who have always managed to get paperwork where it is needed to go in spite of me.

My family has also supported, encouraged and tolerated this thesis process. Matt Lemay and his wife Alex and their dog Lilly were always ready to provide a needed distraction and encouragement[1]. My parents Bob and Carol Lemay also need to be thanked for their support.

Our roommates and friends Eric Gibbs and Jess Noble (and pets Padfoot, Crookshanks, Mrs. Norris) for keeping us sane(?) during the pandemic lockdown. There are too many other friends to list here, but special thanks to Ramsay Hoguet and Elise Simons for proofreading a draft of this thesis.

[1] "Cs get degrees bro"

A DEPENDENTLY TYPED PROGRAMMING LANGUAGE WITH DYNAMIC EQUALITY

MARK LEMAY

Boston University, Graduate School of Arts and Sciences, 2022

Major Professor: Hongwei Xi, PhD
Associate Professor of Computer Science

ABSTRACT

Dependent types offer a uniform foundation for both proof systems and programming languages. While the proof systems built with dependent types have become relatively popular, dependently typed programming languages are far from mainstream.

One key issue with existing dependently typed languages is the overly conservative definitional equality that programmers are forced to use. When combined with a traditional typing workflow, these systems can be quite challenging and require a large amount of expertise to master.

This thesis explores an alternative workflow and a more liberal handling of equality. Programmers are given warnings that contain the same information as the type errors that would be given by an existing system. Programmers can run these programs optimistically, and they will behave appropriately unless a direct contradiction confirming the warning is found.

This is achieved by localizing equality constraints using a new form of elaboration based on bidirectional type inference. These local checks, or casts, are given a runtime behavior (similar to those of contracts and monitors). The elaborated terms have a weakened form of type soundness: they will not get stuck without an explicit counter

example.

The language explored in this thesis will be a Calculus of Constructions like language with recursion, type-in-type, data types with dependent indexing and pattern matching.

Several meta-theoretic results will be presented. The key result is that the core language, called the **cast system**, "will not get stuck without a counter example"; a result called **cast soundness**. A proof of cast soundness is fully worked out for the fragment of the system without user defined data, and a Coq proof is available. Several other properties based on the gradual guarantees of gradual typing are also presented. In the presence of user defined data and pattern matching these properties are conjectured to hold.

A prototype implementation of this work is available.

Preface

The source files for this dissertation are available on GitHub[2]. This document was compiled from those sources on 2022-07-11 11:48:34-04:00.

Please contact me if you find an error, unclear writing or something I should have cited.

Mark Lemay

[2] `https://github.com/marklemay/thesis`

Contents

1 Introduction **1**

1.1 Example: Head of a Non Empty List 2

1.2 A Different Workflow 4

1.3 Example: System F Interpreter 7

1.4 Design Decisions 10

1.5 Issues 11

1.6 The Work in This Thesis 12

2 A Dependent Type System **15**

2.1 Surface Language Syntax 16

2.2 Examples 18

2.2.1 Church Encodings 18

2.2.2 Proposition Encodings 20

2.2.3 Large Eliminations 21

2.2.4 Inequalities 22

2.2.5 Recursion 22

2.3 Surface Language Evaluation 23

2.4 Surface Language Type Assignment System 25

2.4.1 Definitional Equality 30

2.4.2 Preservation 35

2.4.3 Progress 39

2.4.4 Type Soundness 41

2.4.5 Type Checking Is Impractical 41
2.5 Bidirectional Surface Language 42
2.5.1 Bidirectional Type Checking 42
2.5.2 The Bidirectional System is Type Sound 45
2.5.3 The TAS System Is Weakly Annotatable by the Bidirectional System 45
2.6 Absent Logical Properties 46
2.6.1 Logical Inconsistency 46
2.7 Related Work 50
2.7.1 Bad Logics, ok Programming Languages? 50
2.7.2 Implementations 51
2.7.3 Other Dependent Type Systems 51

3 The Dependent Cast System **54**

3.1 Cast Language 55
3.1.1 How Should Casts Reduce? 57
3.2 Examples 60
3.2.1 Higher Order Functions 60
3.2.2 Type Universes 60
3.2.3 Pretending $true = false$ 61
3.3 Cast Language Evaluation and Blame 63
3.4 Cast System 65
3.4.1 Definitional Equality 68
3.4.2 Preservation 71
3.4.3 Cast Soundness 78
3.4.4 Discussion 79
3.5 Elaboration 80

3.5.1 Examples . . . 81
3.5.2 Elaboration Procedure . . . 82
3.6 Suitable Warnings . . . 85
3.7 Related Work . . . 86
3.7.1 Bidirectional Placement of Casts . . . 86
3.7.2 Contract Systems . . . 86
3.7.3 Refinement Style Approaches . . . 88

4 Data in the Surface Language **90**
4.1 Data . . . 91
4.2 Direct Elimination . . . 91
4.2.1 Type Assignment System . . . 98
4.2.2 Bidirectional Type Checking . . . 104
4.3 Pattern Matching . . . 104
4.3.1 First Order Unification . . . 109
4.4 Discussion . . . 113
4.5 Related work . . . 116
4.5.1 Dependent Systems with Data . . . 116
4.5.2 Dependent Pattern matching . . . 117

5 Data in the Cast Language **118**
5.1 Examples . . . 120
5.1.1 Head . . . 120
5.1.2 Sum . . . 122
5.1.3 Missing Branches . . . 123
5.1.4 Congruence (embedding equalities in terms) . . . 124
5.1.5 Peeking . . . 125
5.2 Syntax . . . 126

5.3 Endpoint Rules . . . 128
5.3.1 Function Fragment . . . 129
5.3.2 Data Endpoints . . . 129
5.4 Reductions . . . 133
5.4.1 Function Fragment . . . 133
5.4.2 Data Reductions . . . 133
5.4.3 Endpoint Preservation . . . 135
5.5 Cast soundness . . . 135
5.6 Elaboration . . . 138
5.6.1 Unification . . . 138
5.6.2 Elaboration . . . 140
5.7 Prior Work . . . 140
5.8 Future Work . . . 141

6 Notes and Future Work **143**

6.1 Automatic Testing . . . 143
6.1.1 Observable Blame . . . 144
6.1.2 Test Environment . . . 145
6.1.3 Related and Future Work . . . 149
6.2 Runtime Proof Search . . . 151
6.2.1 Prior Work . . . 154
6.3 Future work . . . 155
6.3.1 Effects . . . 155
6.4 User studies . . . 158
6.5 Semantics . . . 158

7 Conclusion **160**

Bibliography **162**

Curriculum Vitae **171**

List of Figures

1·1 Standard Typed Programming Workflow 5

1·2 Workflow Advocated in this Thesis 6

1·3 System F Interperter in the Proposed Language 9

2·1 Surface Language Syntax 17

2·2 Surface Language Abbreviations 17

2·3 Example Surface Language Expressions 19

2·4 Reflexivity, Symmetry, and Transitivity Proven in the Surface Language 21

2·5 Surface Language Value Syntax 23

2·6 Surface Language Call-by-Value Reductions 24

2·7 Surface Language Type Assignment System 26

2·8 Surface Language Parallel Reductions 29

2·9 Rewriting Diagrams 32

2·10 The **max** Function 33

2·11 Definitionally Equal Contexts 36

2·12 Surface Language Bidirectional Typing Rules 44

3·1 Cast Language Syntax 56

3·2 Cast Language Abbreviations 57

3·3 Approximate Cast Language Reductions and Blame 59

3·4 true=false 62

3·5 true=false cont. 63

3·6 Cast Language Values 63

3·7 Cast Language Call-by-Value Reductions 64
3·8 Cast Language Blame 66
3·9 Cast Language Type Assignment Rules 67
3·10 Cast Language Parallel Reductions 69
3·11 The **max** Function 70
3·12 Surface Language Syntax with Locations 81
3·13 Surface Language Abbreviations 81
3·14 Elaboration 84
3·15 Erasure 85

4·1 Definitions of Common Data Types 92
4·2 Direct Eliminator Scheme Examples 94
4·3 Surface Language (Direct Eliminator) Data 95
4·4 Meta rules 99
4·5 Surface Language Data **ok** 99
4·6 Surface Language Data Typing 101
4·7 Surface Language Data Reduction 102
4·8 Surface Language Data Call-by-Value 103
4·9 Surface Language Empty 103
4·10 Surface Language Bidirectional Type Checking 105
4·11 Eliminators vs. Pattern Matching 107
4·12 Surface Language Data 108
4·13 Surface Language Match 108
4·14 Surface Language Unification 110

5·1 Cast Pattern Matching 125
5·2 Cast Language Syntax 127
5·3 Endpoints (non-data) 130

5·4 Definitions 131
5·5 Cast Data Endpoint Rules 132
5·6 Typed Index Function 133
5·7 Cast Language Small Step (select rules) 134
5·8 Data Reductions 136
5·9 Pointwise Indexing 136
5·10 Consistent 137
5·11 Cast Language Unification Rules (select) 139

6·1 Ideal Workflow 144
6·2 Test Environment 145
6·3 Environment and Observation Abbreviations 146
6·4 Symbolic reduction 146

List of Abbreviations

CC Calculus of Constructions

CIC Calculus of Inductive Constructions

ECC Extended Calculus of Constructions

ETT Extensional Type Theory

ICC Implicit Calculus of Constructions

ITT Intensional Type Theory

MLTT Martin Löf Type Theory

PTS Pure Type Systems

TAS Type Assignment System

UTT Unified Type Theory

Chapter 1

Introduction

Writing correct programs is difficult. While formal methods can make some errors rare or impossible, they often require programmers to learn additional syntax and semantics. Dependent type systems can offer a simpler approach. In dependent type systems, proofs and properties use the same language and meaning already familiar to functional programmers.

While the type systems of mainstream programming languages allow tracking simple properties, like $\texttt{7} : \texttt{int}$ or $\texttt{not(x)} : \texttt{bool}$, dependent types allow complicated properties to be assumed and verified, such as a provably correct sorting function

$$sort \ : \ (input : \texttt{List Nat}) \rightarrow \Sigma ls : \texttt{List Nat.IsSorted} \ input \ ls$$

by providing an appropriate definition of *sort* at that type. From the programmer's perspective, the function arrow and the implication arrow are the same. The proof `IsSorted` is no different than any other term of a datatype like `List` or `Nat`.

The power of dependent types has been recognized for decades. Dependent types form the backbone of several poof systems, such as Coq[CDT12], Lean[MU21], and Agda[Nor07]. They have been proposed as a foundation for mathematics[ML72, Pro13]. Dependent types are directly used in several programming languages such as ATS[Xi07] and Idris[Bra13], while influencing many other programing languages such as Haskell and Scala.

Unfortunately, dependent types have not yet become mainstream in the software

industry. Many of the usability issues with dependent types can trace their root to the conservative nature of dependently typed equality. This thesis illustrates a new way to deal with equality constraints by delaying them until runtime.

A fragment of the system is proven correct according to a modified view of type soundness, and several of the proofs have been validated in Coq[1]. The system has been prototyped[2].

1.1 Example: Head of a Non Empty List

Dependent type systems can prevent an index-out-of-bounds error when trying to read the first element of a list. A version of the following type checks in virtually all dependent type systems:

$$
\begin{aligned}
&\texttt{Bool} : \star,\\
&\texttt{Nat} : \star,\\
&\texttt{Vec} : \star \to \texttt{Nat} \to \star,\\
&\texttt{add} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat},\\
&\texttt{rep} : (A : \star) \to A \to (x : \texttt{Nat}) \to \texttt{Vec}\, A\, x,\\
&\texttt{head} : (A : \star) \to (x : \texttt{Nat}) \to \texttt{Vec}\, A\, (\texttt{add}\, 1\, x) \to A
\end{aligned}
$$

$$
\vdash \lambda x \Rightarrow \texttt{head}\, \texttt{Bool}\, x\, (\texttt{rep}\, \texttt{Bool}\, \texttt{true}\, (\texttt{add}\, 1\, x)) : \texttt{Nat} \to \texttt{Bool}
$$

Where $\to$ is a function and $\star$ is the "type of types". `Vec` is a list indexed by the type of element it contains and its length[3]. `Vec` is a dependent type since it is a type of list that depends on its length. `rep` is a dependent function that produces

[1] Available at `https://github.com/marklemay/dtest-coq` with most of the Coq scripts due to Qiancheng Fu.

[2] Available at `https://github.com/marklemay/dDynamic`.

[3] Definitions of these types and functions can be found in Chapter 5.

a list with a given length by repeating its input that number of times. `head` is a dependent function that expects a list of length $\texttt{add}\,1\,x$, returning the first element of that non-empty list.

There is no risk that `head` inspects an empty list. Luckily, in the example, $\texttt{rep}\,\texttt{Bool}\,\texttt{true}\,(\texttt{add}\,1\,\texttt{x})$ will result in a list of length $\texttt{add}\,1\,x$, exactly the type that is required.

Unfortunately, programmers often find dependent type systems difficult to learn and use. This resistance has limited the ability of dependent types to reach their full potential to help eliminate the bugs that pervade software systems. One of the deepest underlying reasons for this frustration is the way dependent type systems handle equality.

For example, the following will not type check in any existing dependent type system[4],

$$\not\vdash \lambda x \Rightarrow \texttt{head}\,\texttt{Bool}\,x\;(\texttt{rep}\,\texttt{Bool}\,\texttt{true}\;(\texttt{add}\,x\,1)) : \texttt{Nat} \to \texttt{Bool}$$

While "obviously" $1 + x = x + 1$, in current dependently typed languages, $\texttt{add}\,1\,x$ and $\texttt{add}\,x\,1$ are not definitionally equal. **Definitional equality** is the name for the conservative approximation of equality used by dependent type systems for when two expressions are clearly the same. This prevents the use of a term of type $\texttt{Vec}\,\texttt{Bool}\;(\underline{\texttt{add}\,1\,x})$ where a term of type $\texttt{Vec}\,\texttt{Bool}\;(\underline{\texttt{add}\,x\,1})$ is expected. Usually when dependent type systems encounter situations like this, the type checker will give an error message and prevent the programmer from doing more work until the "mistake" is resolved.

In programming, types are used to avoid bad behavior. For instance, they are often used to avoid "stuck" terms. If it is the case that $\texttt{add}\,1\,x = \texttt{add}\,x\,1$ the program will

[4]At least without additional formal information about the `add` function that has been purposely withheld from this example.

never get stuck. However, if there is a mistake in the implementation of `add`, then it is possible $\texttt{add}\,1\,x \not= \texttt{add}\,x\,1$ and the program might get stuck. For instance, if the `add` function incorrectly computes $\texttt{add}\,8\,1 = 0$ the above function will "get stuck" on the input 8.

While the intent and properties of the `add` function are clear to programmers from its name and type, this information is unusable by the type system. If the programmer made a mistake in the definition of addition, such that for some x, $\texttt{add}\,1\,x \not= \texttt{add}\,x\,1$, the system will not provide hints on which x witnesses this inequality. Worse, the type system may even make it difficult to experiment with the `add` function by disallowing evaluation in an "untyped" file, which makes repairing an actual bug difficult.

Why *stop* programmers when there is a definitional equality "error"?

There appears to be no reason! Alternatively, we can track unclear equalities and if the program "gets stuck", we are able to stop the program execution and provide a concrete witness for the inequality at runtime. If the buggy `add` function above is encountered at runtime, we can give a runtime error stating $\texttt{add}\,1\,8 = 9 \neq 0 = \texttt{add}\,8\,1$. Which is exactly the kind of specific feedback programmers want when fixing bugs.

1.2 A Different Workflow

This thesis advocates an alternative usage of types. In most types systems a programmer can't run programs until the type system is convinced of their correctness[5], whereas this thesis argues "the programmer is always right (until proven wrong)". This philosophy might go over better with programmers.

More concretely, whenever possible, static errors should be replaced with:

- Static warnings containing the same information, indicating that an error might

[5] In dependent type systems this often requires advanced study and uncommon patience.

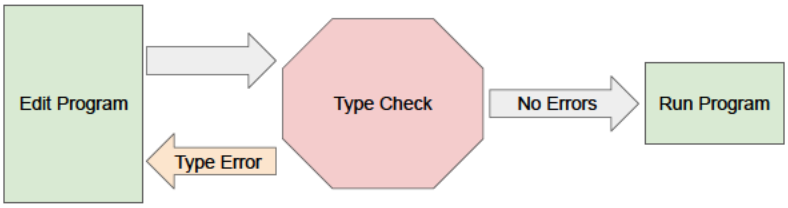


Figure 1·1: Standard Typed Programming Workflow

occur.

- More concrete and clear runtime errors that correspond to one of the warnings.

Figure 1·1 illustrates the standard workflow from the perspective of programmers in most typed languages. Figure 1·2 shows the workflow that is explored in this thesis.

These diagrams make it clear why there is so much pressure for type errors to be better in dependently typed programming. Type errors block programmers from running programs! However complaints about the type errors are probably better addressed by resolving the mismatch between the expectations of the programmer and the design of the underlying type theory. Better worded error messages are unlikely to bridge this gap when the type system doubts $x + 1 = 1 + x$.

The standard workflow seems sufficient for type systems in many mainstream typed programming languages, though there is experimental evidence that even OCaml can be easier to learn and use with the proposed workflow[SJW16]. In the presence of dependent types the standard workflow is challenging for both beginners and experts, making a new approach much more critical.

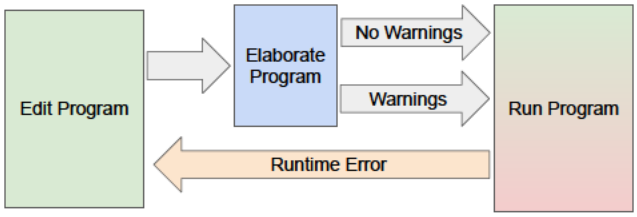


Figure 1·2: Workflow Advocated in this Thesis

By switching to the proposed workflow, type errors become type warnings, and the programmer is free to run their program and experiment, while still presented with all the information they would have received from a type error (in the form of warnings). If there are no warnings, the programmer could call their program a proof along the lines of the Curry-Howard correspondence[6]. If there is value in a type error it comes from the message itself and not the inconvenience it causes programmers.

The proposed workflow is necessary, since often the type system is too conservative and the programmer is correct in implicitly asserting an equality that is not provable within the system. That the programmer may need to go outside the conservative bounds of definitional equality has been recognized since the earliest dependent type theories [ML72], and difficulties in dependently typed equality have motivated many research projects [Pro13, SW15, CTW21]. However, these impressive efforts are still only usable by experts, since they frequently require the programmer prove their equalities explicitly[Pro13, SW15], or add custom rules into the type system

[6]In the system presented here, the programmer should informally argue all relevant constructions are well founded to claim they have a proof.

[CTW21]. Further, since program equivalence is undecidable in general, no system will be able to statically verify every "obvious" equality for arbitrary user defined data types and functions. In practice, every dependently typed language has a way to assume equalities, even though these assumptions will result in computationally bad behavior[7].

The proposed workflow presented in this thesis is justified by:

- The strict relation between warnings and runtime errors. A runtime error will always correspond exactly to a reported warning, always validating the warning with a specific example.
- A form of type soundness holds, programs will never "get stuck" unless a concrete witness that corresponds to a warning is found.
- Programs that type check against a conventional type system should not have warnings, and therefore cannot have errors.
- Other than warnings and errors the runtime behavior is similar to other conventional dependently typed languages.

1.3 Example: System F Interpreter

While the primary benefit of this system is the ability to experiment more freely with dependent types while still getting the full feedback of a dependent type system, it is also possible to encode examples that would be unfeasible in existing systems. This comes from accepting warnings that are justified with external mathematical or programmatic intuition, even while being theoretically thorny in dependent type theory.

[7] The program may "get stuck".

For instance, here is part of an interpreter for System F[8] that encodes the type of the term at the type level. The step function asserts type preservation of the interpreter in its function signature. It will generate warnings like the following:

- `tbod` in `Term` (`tSubCtx targ ctx`) (`tSubt targ tbod`) may have the wrong type

First, note that the program has assumed several of the standard properties of substitution. Informally, substitution and binding is usually considered obvious and uninteresting, and little explanation is usually given[9]. However, formalizing substitution in a dependent type theory is usually a substantial task[SSK19].

Second, the type contexts have been encoded as functions. This would be a reasonable encoding in a mainstream functional language since it hides the uninteresting lookup information. This encoding would be inadvisable in other dependently typed languages since functions that act the same may not be definitionally equal. Here we can rest on the intuition that functions that act the same are the same.

Finally, it is perfectly possible that there is a bug in the code invalidating one of the assumptions. There are two options for the programmer:

- Reformulate the above code so that there are no warnings, formally proving all the required properties as one might in a conventional dependently typed language (this is possible but would take a serious effort).

- Exercise the `step` function using standard software testing techniques. If the interpreter does not preserve types, then a concrete counter example can be found.

[8] System F is one of the foundational systems used to study programming languages. It is possible to fully encode evaluation and proofs into Agda, but it is difficult if computation like substitution happens in a type. In the system described in this thesis, it is possible to start with a type indexed encoding and build an interpreter, without proving any properties of substitution.

[9] A convention that will be followed in this thesis.

```
Ctx : * ;
Ctx = Var → Ty;

data Ty : * {
| tv : tVar → Ty
| arr : Ty → Ty → Ty
| forall : Ty → Ty
};

data Term : Ctx → Ty → * {
| V :    (ctx : Ctx) →
         (x : Var) →
         Term ctx (ctx x)
| lam :  (ctx : Ctx) → (targ : Ty) → (tbod : Ty) →
         (bod : Term (ext ctx targ) tbod) →
         Term ctx (arr targ tbod)
| app :  (ctx : Ctx) → (targ : Ty) → (tbod : Ty) →
         (func : Term ctx (arr targ tbod)) →
         (arg : Term ctx targ) →
         Term ctx tbod
| tlam : (ctx : Ctx) → (tbod : Ty) →
         (bod : Term ctx tbod) →
         Term ctx (forall tbod)
| tapp : (ctx : Ctx) → (targ : Ty) → (tbod : Ty) →
         (bod : Term ctx (forall tbod)) →
         Term (tSubCtx targ ctx) (tSubt targ tbod)
};

step : (ctx : Ctx) → (ty : Ty) →
       (term : Term ctx ty) →
       Term ctx ty ;
step ctx ty trm = case trm {
  | (app _ targ tbod (lam _ _ _ bod) a)
    ⇒ sub ctx targ a tbod bod
  | (tapp _ targ tbod (tlam _ _ bod))
    ⇒ subt ctx targ tbod bod
  | x ⇒ x
};
```

Figure 1·3: System F Interperter in the Proposed Language

The programmer is free to choose how much effort should go into removing warnings. But even if the programmer wanted a fully formally proven correct interpreter, it would still be wise to test the functions first before attempting such a proof.

1.4 Design Decisions

There are many flavors of dependent types that can be explored. This thesis attempts to always use the simplest and most programmer friendly formulations. Specifically:

- The type system in this thesis is a **full spectrum** dependent type system. The full spectrum approach is the most uniform approach to dependent type theory: computation behaves the same at the term and type level. This is contrasted with a leveled theory where terms embedded in types may have different or limited behavior[10]. The full spectrum approach is popular with theorem provers and has been advocated by many authors [Aug98, Nor07, Bra13, SCA+12]. While the full spectrum approach offers tradeoffs (it is harder to deal with effects), it seems to be the most predictable from a newcomer's perspective.

- Data types and pattern matching are essential to practical programming, so they are included in the implementation. While it is theoretically possible to simulate data types via church encodings, they are too awkward for programmers to work with, and would complicate the runtime errors this system hopes to deliver. To provide a better programming experience data types are built into the system and pattern matching is supported.

- The theories presented in this thesis will allow unrestricted general recursion and thus allow non-termination. While there is some dispute about how essential general recursion is, there is no mainstream general purpose

[10] This is the approach taken in ATS, and refinement type systems.

programming language that restricts termination. Allowing nontermination weakens the system when considered as a logic (any proposition can be given a nonterminating inhabitant). This removes any justification for a type universe hierarchy, so the system will support type-in-type. Similarly, non-positive data type definitions are allowed.

- Aside from the non-termination and runtime errors mentioned above, effects will not be allowed. Even though effects seem essential to mainstream programing, they are a very complicated area of active research that will not be considered here. The language studied is a "pure" (in the sense of Haskell) functional language. As in Haskell, effects can be simulated with other language constructs.

It is possible to imagine a system where a wide range of properties are held optimistically and tested at runtime. However the bulk of this thesis will only deal with equality, since that relation is uniquely fundamental to dependent type systems. Since computation can appear at the type level, and types must be checked for equality, dependent type theories must define what computations they intend to equate for type checking. It would be premature to deal with any other properties until definitional equality is dealt with.

1.5 Issues

Weakening the definitional equality relation in a dependent types system is easier said than done.

- Since the system is a full spectrum dependent type system, ill typed terms will appear in types. What does it mean when a term is a list of length `True`?

- If we insert new syntax to perform the checks, terms might "get stuck" in new ways. What happens when an equality check is used as a function by being

applied to an argument? What happens when a check blocks a pattern match?

- Equality is not decidable at many types, even in the empty context. For instance, functions of type `Nat` $\to$ `Nat` do not have decidable equality. Therefore the types that embed functions of type `Nat` $\to$ `Nat` do not have decidable equality.

These problems are solved by extending a dependent type theory with a cast operator along with appropriate typing rules and operational semantics. This cast operator will "get stuck" if there is a discrepancy, and we can show that a program will always resolve to a value or get stuck in such a way that a counterexample can be reported. Further,

- A system is needed to insert these casts. Casts can be generated by extending a bidirectional typing procedure to localize casts.

- Once the casts are inserted, evaluations are possible. Checking only needs to happen up to the outermost type constructor, avoiding issues of undecidable equality.

- Pattern matching can be extended to support equality evidence. The branches of the pattern match can modify and use this evidence. Unsatisfiable branches can redirect blame as needed.

1.6 The Work in This Thesis

While apparently a simple idea, the technical details required to manage checks that delay until runtime in a dependently typed language are fairly involved. To make the presentation easier, features will be added to the language in stages.

- Chapter 2 describes a dependently typed language (without data) intended to model standard dependent type theories, called the **surface language**. **Type soundness** is proven and a bidirectional type checking procedure system is presented. Though the system is not original, the proof presented is the simplest complete progress and preservation style proof I am aware of for a dependent type system[11].

- Chapter 3 describes a dependently typed language (without data) with embedded equality checks, called the **cast language**. The cast language has its own version of type soundness, called **cast soundness**. Cast soundness is proven for the cast language, using a similar strategy to the type soundness proof of Chapter 2. Then an elaboration procedure that takes most (untyped) terms of the surface syntax into terms in the cast language is presented. Several desirable properties for elaboration are presented.

- Chapter 4 reviews how dependent data types and pattern matching can be added to the surface language and explores some of the issues of data types in a dependent type system.

- Chapter 5 shows how to extend the cast language with dependent data types and pattern matching. Surprisingly the inclusion of dependent data types requires several changes to the system in Chapter 3, since finer observations are possible.

- Chapter 6 discusses other ideas related to usability and future work, such as automated testing and runtime proof search. These systems are made feasible given the more flexible approach to equality addressed in the rest of the thesis.

[11] [Sjö15] made a similar claim, though since it included more features (such as data), our proof would technically be simpler. Martin-Löf had a similar proof for a similar system in his unpublished notes [ML71], though that predates some of the proof techniques used in Chapter 2.

Versions of the proof of type soundness in Chapter 2 and the cast soundness in Chapter 3 have been formally proven in Coq.

Those interested in exploring the type soundness and type checking of a "standard" dependent type theory can read Chapters 2 and 4 which can serve as a self contained tutorial.

Chapter 2

A Dependent Type System

Despite the usability issues this thesis hopes to correct, dependent type systems are still one of the most promising technologies for correct programming. Since proofs are programs, there is no additional syntax for programmers to learn. The proof system is predictable from the perspective of a functional programmer.

The **surface type system** presented in this chapter provides a minimal dependent type system. The rules of the type system are intended to be as simple as possible and compatible with other well studied intensional dependent type theories. It has several (but not all) of the standard properties of dependent type theory. As much as possible, the syntax uses standard modern notation[1].

The surface type system will serve both as a foundation for later chapters and a self contained technical introduction to dependent types. Even when using the full system described in later chapters, programmers will only need to think about the surface system. By design, the machinery that deals with equality addressed in later chapters will be invisible to programmers. Everything presented in later chapters is designed to reinforce an understanding of the surface type system and make it easier to use.

The surface language deviates from a standard dependent type theory to include features for programming at the expense of logical correctness. Specifically, the

[1]Several alternative syntaxes exist in the literature. In this document the typed polymorphic identity function is written, $\lambda - x \Rightarrow x \ : (X : \star) \to X \to X$. In [Pro13] it might be written $\lambda X.\lambda x.x \ : \prod_{(X:\mathcal{U})} X \to X$. In [CH88] it might be written $(\lambda X : \star)\,(\lambda x : X)\,x \ : [X : \star]\,[x : X]\,X$.

language allows general recursion, since general recursion is useful for programmers. Type-in-type is also supported since it simplifies the system and makes the meta-theory slightly easier. Despite this, type soundness is achievable, and a practical type checking system is given.

Though similar systems have been studied over the last few decades, this chapter aims to give a self contained presentation, along with examples. The surface language has been a good platform to conduct research into full spectrum dependent type theory, and hopefully this exposition will be a helpful introduction for other researchers.

2.1 Surface Language Syntax

The syntax for the surface language is in Figure 2·1. The syntax supports: variables, type annotations, a single type universe, dependent function types, recursive functions, and function applications. Type annotations are written with two colons to differentiate it from the formal typing judgments that will appear more frequently in this text. In the implemented language a user of the programming language would use a single colon for an annotation.

There is no distinction between types and terms in the syntax[2]. Both are referred to as expressions. However, capital meta variables are used in positions that are intended as types, and lowercase meta variables are used when an expression may be a term. For instance, in annotation syntax where $m :: M$ means m can be a term and M should be a type.

Several standard abbreviations are listed in Figure 2·2.

[2]Terms and types are usually syntactically separated, except in the syntax of full spectrum dependent type systems where separating them would require many redundant rules.

variable identifiers,
x, y, z, f
expressions,

m, n, M, N	$::=$	x	variable
	$\mid$	$m :: M$	annotation
	$\mid$	$\star$	type universe
	$\mid$	$(x : M) \to N$	function type
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	function
	$\mid$	$m\, n$	application

type contexts,

Γ	$::=$	$\Diamond \mid \Gamma, x : M$

Figure 2·1: Surface Language Syntax

$(x : M) \to N$	written	$M \to N$	when	$x \notin fv\,(N)$
$\mathsf{fun}\, f\, x \Rightarrow m$	written	$\lambda x \Rightarrow m$	when	$f \notin fv\,(m)$
$\ldots\, x \Rightarrow \lambda y \Rightarrow m$	written	$\ldots\, x\, y \Rightarrow m$		
x	written	$-$	when	$x \notin fv\,(m)$ when x binds m

where fv is a function that returns the set of free variables in an expression

Figure 2·2: Surface Language Abbreviations

2.2 Examples

The surface system is extremely expressive. Several example surface language constructions can be found in Figure 2·3. Turnstile notion is abused slightly so that examples can be indexed by other expressions that obey type rules. For instance, given the definition of $+_c$ in Figure 2·3, we can say $2_c +_c 2_c \ : \ \mathbb{N}_c$ since $2_c : \mathbb{N}_c$.

2.2.1 Church Encodings

Data types are expressible using Church encodings (in the style of System F). Church encodings embed the elimination principle of a data type into higher order functions. For instance, boolean data is eliminated against true and false, two constructors with no additional data. This can also be recognized as the if-then-else construct that is built into most programming languages. As defined in Figure 2·3, $\mathbb{B}_c$ encodes the possibility of choice between two elements, $true_c$ picks the *then* branch, and $false_c$ picks the *else* branch.

Natural numbers[3] are encodable with two constructors, zero and successor. In this encoding, the successor constructor also contains the result of processing the preceding number. So $\mathbb{N}_c$ encodes those two choices, $(X \to X)$ handles the recursive result of the prior number in the successor case, and the X argument specifies how to handle the base case of 0. This can be viewed as a looping construct with temporary storage that loops exactly as many times as the number it represents.

Parameterized data types such as pairs and the $Either_c$ type can also be encoded in this scheme. A pair type can be used in any way the two terms it contains can, so a pair is defined as the curried input to a function. The $Either_c$ type is handled if both possibilities are handled, so it is defined as a higher order function that will return an output if both possibilities are handled for input.

[3]Called **church numerals** in this scheme.

	$\vdash \bot_c$	$\coloneqq (X : \star) \to X$	$: \star$	Void, "empty" type, logical false
	$\vdash Unit_c$	$\coloneqq (X : \star) \to X \to X$	$: \star$	Unit, logical true
	$\vdash tt_c$	$\coloneqq \lambda - \; x \Rightarrow x$	$: Unit_c$	trivial proposition, polymorphic identity
	$\vdash \mathbb{B}_c$	$\coloneqq (X : \star) \to X \to X \to X$	$: \star$	booleans
	$\vdash true_c$	$\coloneqq \lambda - \; then - \Rightarrow then$	$: \mathbb{B}_c$	boolean true
	$\vdash false_c$	$\coloneqq \lambda - \; - else \Rightarrow else$	$: \mathbb{B}_c$	boolean false
$x : \mathbb{B}_c$	$\vdash !_c x$	$\coloneqq x \, \mathbb{B}_c \, false_c \, true_c$	$: \mathbb{B}_c$	boolean not
$x : \mathbb{B}_c, y : \mathbb{B}_c$	$\vdash x \,\&_c\, y$	$\coloneqq x \, \mathbb{B}_c \, y \, false_c$	$: \mathbb{B}_c$	boolean and
	$\vdash \mathbb{N}_c$	$\coloneqq (X : \star) \to (X \to X) \to X \to X$	$: \star$	natural numbers
	$\vdash 0_c$	$\coloneqq \lambda - \; - z \Rightarrow z$	$: \mathbb{N}_c$	
	$\vdash 1_c$	$\coloneqq \lambda - \; s \, z \Rightarrow s \, z$	$: \mathbb{N}_c$	
	$\vdash 2_c$	$\coloneqq \lambda - \; s \, z \Rightarrow s \, (s \, z)$	$: \mathbb{N}_c$	
	$\vdash n_c$	$\coloneqq \lambda - \; s \, z \Rightarrow s^n \, z$	$: \mathbb{N}_c$	
$x : \mathbb{N}_c, y : \mathbb{N}_c$	$\vdash x +_c y$	$\coloneqq \lambda X \, s \, z \Rightarrow x \, X \, s \; (y \, X \, s \, z)$	$: \mathbb{N}_c$	addition
$X : \star, Y : \star$	$\vdash X \times_c Y$	$\coloneqq (Z : \star) \to (X \to Y \to Z) \to Z$	$: \star$	pair, logical and
$X : \star, Y : \star$	$\vdash Either_c \, X \, Y$	$\coloneqq (Z : \star) \to (X \to Z) \to (Y \to Z) \to Z$	$: \star$	either, logical or
$X : \star$	$\vdash \neg_c X$	$\coloneqq X \to \bot_c$	$: \star$	logical negation
$x : \mathbb{N}_c$	$\vdash Even_c \, x$	$\coloneqq x \star (\lambda x \Rightarrow \neg_c x) \; Unit_c$	$: \star$	x is an even number
$X : \star, Y : X \to \star$	$\vdash \exists_c x : X \Rightarrow Y \, x$	$\coloneqq (C : \star) \to ((x : X) \to Y \, x \to C) \to C$	$: \star$	dependent pair, logical exists
$X : \star, x_1 : X, x_2 : X$	$\vdash x_1 \doteq_X x_2$	$\coloneqq (C : (X \to \star)) \to C \, x_1 \to C \, x_2$	$: \star$	Leibniz equality

Figure 2·3: Example Surface Language Expressions

Church encodings provide a theoretically lightweight way of working with data in a minimal lambda calculus. However, they are inconvenient. For instance, the predecessor function on natural numbers is not as simple as it would seem. To make the system easier for programmers, data types will be added directly in Chapter 4.

2.2.2 Proposition Encodings

In general we associate the truth value of a proposition with the inhabitation of a type by a meaningful value. This meaningful term corresponds to a proof. So, $\perp_c$, the "empty" type, can be interpreted as a false proposition, while $Unit_c$ can be interpreted as a trivially true proposition, since it has only one good inhabitant[4].

Several of the Church encoded data types we have seen can also be interpreted as logical predicates. For instance, the tuple type can be interpreted as logical *and* since $X \times_c Y$ can only be meaningfully inhabited when both X and Y are inhabited. The *Either* type can be interpreted as logical *or* since $Either_c\, X\, Y$ can be inhabited when either X or Y is inhabited.

With dependent types, more interesting logical predicates can be encoded. For instance, we can characterize when a number is even with $Even_c$. We can show that 2 is even by showing that $Even_c\, 2_c$ is inhabited with the term $\lambda s \Rightarrow s\, tt_c$. Since the definition of $Even_c\, 2_c$ expands to $(Unit_c \to \perp_c) \to \perp_c$, given a function $s : (Unit_c \to \perp_c)$ we only need to give it a member of $Unit_c$ to satisfy the type constraint.

Other predicates are encodable in this style (See [ML71, Car86, CH88] for more examples). For instance, we can encode the existential quantifier as $\exists_c$ as shown in

[4]Remember to keep the the different notions of "truth" separate:

$\mathbb{B}$ is a collection of two constructors *true* and *false*. The names are arbitrary, nothing but convention informs their meaning. Just as if-then-else constructs could be reordered into if-else-then without changing anything essential.

In type theory the $\perp$ proposition has no proofs. If you ever get one, something has gone wrong and you have an excuse to do anything. Meanwhile the $Unit$ proposition contains just a single trivial proof. Nothing interesting can be done with it, just as nothing interesting can be done with the identity function.

Finally, these notions are distinct from the meta theoretic properties that will be presented later.

$$
\begin{array}{l}
X : \star, x : X \\
\quad \vdash refl_{x:X} \qquad\qquad \coloneqq \lambda - \ cx \Rightarrow cx \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : x \doteq_X x \\
X : \star, x_1 : X, x_2 : X \\
\quad \vdash sym_{x_1,x_2:X} \qquad \coloneqq \lambda p\ C \Rightarrow p\,(\lambda x \Rightarrow C\,x \to C\,x_1)\ (\lambda x \Rightarrow x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad : x_1 \doteq_X x_2 \to x_2 \doteq_X x_1 \\
X : \star, x_1 : X, x_2 : X, x_3 : X \\
\quad \vdash trans_{x_1,x_2,x_3:X} \quad \coloneqq \lambda p_{12}\ p_{23}\ C\ cx\ \Rightarrow p_{23}\,C\ (p_{12}\,C\,cx) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad : x_1 \doteq_X x_2 \to x_2 \doteq_X x_3 \to x_1 \doteq_X x_3
\end{array}
$$

Figure 2·4: Reflexivity, Symmetry, and Transitivity Proven in the Surface Language

Figure 2·3. Then we can show $\exists_c x : \mathbb{N}_c \Rightarrow\ Even_c\,x$ with a suitable inhabitant of that type. 0 is clearly an even number, so our inhabitant could be $\lambda - f \Rightarrow f\,0_c\,tt_c$, since the $Even_c\,0_c$ expands to $Unit_c$ so $tt_c\ :\ Even_c\,0_c$.

One of the most interesting propositions is the proposition of equality. $\doteq$ is referred to as **Leibniz equality** since two terms are equal when they behave the same on all predicates[5]. We can prove $\doteq$ is an equivalence within the system by proving it is reflexive, symmetric, and transitive. These proof expressions are listed in Figure 2·4.

2.2.3 Large Eliminations

It is useful for a type to depend specifically on term level data, this is called **large elimination**. Large elimination can be simulated with type-in-type.

$$
\begin{array}{lll}
toLogic & \coloneqq \lambda b \Rightarrow b \star Unit_c\ \bot_c & :\ \ \mathbb{B}_c \to \star \\
isPos & \coloneqq \lambda n \Rightarrow n \star (\lambda - \Rightarrow Unit_c)\ \bot_c & :\ \ \mathbb{N}_c \to \star
\end{array}
$$

For instance, $toLogic$ can convert a $\mathbb{B}_c$ term into its corresponding logical type, $toLogic\ true_c \equiv Unit_c$ while $toLogic\ false_c \equiv \bot_c$. The expression $isPos$ has similar behavior, going to $\bot_c$ at 0_c and $Unit_c$ otherwise.

[5]Originally, Leibniz assumed a metaphysical identification of "substance", not a mathematical notion of equality[Lei86, Section 9]. Over time the principle evolved into the current notion of "the identification of indiscernibles", and is referred to as **Leibniz's law**.

Note that such functions are not possible in the Calculus of Constructions.

2.2.4 Inequalities

Large eliminations can be used to prove inequalities that can be hard or impossible to express in other minimal dependent type theories. For instance,

$\lambda pr \Rightarrow pr\ (\lambda x \Rightarrow x)\ \bot_c$	:	$\neg_c(\star \doteq_\star \bot_c)$	The type universe is distinct from Logical False
$\lambda pr \Rightarrow pr\ (\lambda x \Rightarrow x)\ tt_c$	:	$\neg_c(Unit_c \doteq_\star \bot_c)$	Logical True is distinct from Logical False
$\lambda pr \Rightarrow pr\, toLogic\, tt_c$	:	$\neg(true_c \doteq_{\mathbb{B}_c} false_c)$	Boolean true and false are distinct
$\lambda pr \Rightarrow pr\, isPos\, tt_c$	:	$\neg(1_c \doteq_{\mathbb{N}_c} 0_c)$	1 and 0 are distinct

Note that a proof of $\neg 1_c \doteq_{\mathbb{N}_c} 0_c$ is not possible in the Calculus of Constructions[Smi88][6].

2.2.5 Recursion

The syntax of functions contain a variable to perform unrestricted recursion. Though not always necessary[7], recursion can be very helpful for writing programs. For instance, here is an (inefficient) function that calculates Fibonacci numbers $\mathsf{fun}\, f\, x \Rightarrow case_c\, x\, 0_c\, (\lambda px \Rightarrow case_c\, px\, 1_c\, (\lambda - \Rightarrow f\, (x -_c 1) +_c f\, (x -_c 2)))$ assuming appropriate definitions for $case_c$, and subtraction.

Recursion can also be used to simulate induction. We will not see much of recursion until Chapter 4, when data types are introduced and larger examples are easier to express.

---

[6] Martin Hofmann excellently motivates the reasoning in the exercises of [Hof97b].

[7] For instance, Church numerals have a limited form of recursive behavior built in.

values,

$$
\begin{array}{lll}
\mathrm{v} & ::= & \star \\
& | & (x : M) \to N \\
& | & \mathsf{fun}\, f\, x \Rightarrow m
\end{array}
$$

Figure 2·5: Surface Language Value Syntax

2.3 Surface Language Evaluation

As a programming language we should have some way to evaluate terms in the surface language. We can define a conventional call-by-value system of reductions on top of the syntax already defined. Call-by-value is a popular execution strategy that reflects the prototype implementation. It works by selecting some expressions as **value**s, expressions that have been computed enough; and a **reduction** relation that will compute terms.

Values are characterized by the sub-grammar in Figure 2·5. As usual, functions with any body are values. Additionally, the type universe ($\star$) is a value, and function types are values.

For instance, $\mathbb{B}_c$ and $true_c$ are values. However, $!_c x$ is not a value since it is defined as an application to a variable.

A call-by-value relation is defined in Figure 2·6. The reductions are standard for a call-by-value lambda calculus, except that type annotations are only removed from values. We will write $\rightsquigarrow_*$ as the transitive reflexive closure of the $\rightsquigarrow$.

For example, the expression $!_c \vdash true_c$ reduces as follows

$$\frac{}{(\mathsf{fun}\, f\, x \Rightarrow m)\, v \rightsquigarrow m\,[f := \mathsf{fun}\, f\, x \Rightarrow m, x := v]}$$

$$\frac{m \rightsquigarrow m'}{m\, n \rightsquigarrow m'\, n}$$

$$\frac{n \rightsquigarrow n'}{v\, n \rightsquigarrow v\, n'}$$

$$\frac{m \rightsquigarrow m'}{m :: M \rightsquigarrow m' :: M}$$

$$\frac{}{v :: M \rightsquigarrow v}$$

$$\frac{}{m \rightsquigarrow_* m}$$

$$\frac{m \rightsquigarrow_* m' \quad m' \rightsquigarrow m''}{m \rightsquigarrow_* m''}$$

Figure 2·6: Surface Language Call-by-Value Reductions

$$\begin{array}{ll} & !_c \vdash true_c \\ = & true_c\, \mathbb{B}_c\, false_c\, true_c \\ = & (\lambda - \, then - \Rightarrow then)\ \mathbb{B}_c\, false_c\, true_c \\ \rightsquigarrow & (\lambda\, then - \Rightarrow then)\ false_c\, true_c \\ \rightsquigarrow & (\lambda - \Rightarrow false_c)\ true_c \\ \rightsquigarrow & false_c \end{array}$$

This system of reductions raises the question, what happens if an expression is not a value but also not cannot be reduced? These terms will be called **Stuck**. Formally, m **Stuck** if m is not a value and there does not exist m' such that $m \rightsquigarrow m'$. For instance, $\star\star\star\star$ **Stuck**.

Though the call-by-value system described here is standard, other systems of reduction are also possible. We will see a non-deterministic system of reductions in Figure 2·8. Occasionally examples will be easier to demonstrate using a weak-head-

normal-form style of reduction, or with custom reduction rules.

2.4 Surface Language Type Assignment System

When is an expression reasonable? The expression $\star \star \star\star$ is allowed by the syntax of the language, but seems dubious. Type systems can disallow **Stuck** terms like these from ever occurring, which in turn prevents bad runtime behavior.

We will present our type system as a **type assignment system** (TAS). Type assignment systems are convenient to study the theory of a dependently typed language because terms do not need to contain information to help type checking, allowing simpler syntax. Practically this means that the type assignment system should not be used as a type checking algorithm since it may need to "infer" an unrealistic amount of information. This also means that terms do not necessarily have unique typings. For instance, $\vdash_{\text{TAS}} \lambda x \Rightarrow x : \mathbb{N}_c \to \mathbb{N}_c$, and $\vdash_{\text{TAS}} \lambda x \Rightarrow x : \mathbb{B}_c \to \mathbb{B}_c$. These issues will be addressed when the more practical, bidirectional type system is introduced in the next section.

The rules of the type assignment system are listed in Figure 2·7. Variables get their type from the typing context by the ty-var rule. Type annotations reflect a correct typing derivation in the ty-: : rule. Type-in-type is recognized by the ty-$\star$ rule. The ty-fun-ty rule forms dependent function types. The ty-fun-app rule shows how to type function application, by substituting the argument term directly into the dependent function type. Functions are typed with a variable for recursive reference along with a variable for the argument in ty-fun. Finally, ty-conv allows type derivations to be **converted** to an equivalent type.

The most important property of a type system is **type soundness**[8]. Type soundness is often motivated with the slogan, "well typed programs don't get

[8] Also called **type safety**.

$$\frac{x : M \in \Gamma}{\Gamma \vdash_{\text{TAS}} x \ : \ M} \text{ ty-var}$$

$$\frac{\Gamma \vdash_{\text{TAS}} m \ : \ M \quad \Gamma \vdash_{\text{TAS}} M \ : \ \star}{\Gamma \vdash_{\text{TAS}} m :: M \ : \ M} \text{ ty-:}:$$

$$\frac{}{\Gamma \vdash_{\text{TAS}} \star \ : \ \star} \text{ ty-}\star$$

$$\frac{\Gamma \vdash_{\text{TAS}} M \ : \ \star \quad \Gamma, x : M \vdash_{\text{TAS}} N \ : \ \star}{\Gamma \vdash_{\text{TAS}} (x : M) \to N \ : \ \star} \text{ ty-fun-ty}$$

$$\frac{\Gamma \vdash_{\text{TAS}} m \ : \ (x : N) \to M \quad \Gamma \vdash_{\text{TAS}} n \ : \ N}{\Gamma \vdash_{\text{TAS}} m\, n \ : \ M\,[x \coloneqq n]} \text{ ty-fun-app}$$

$$\frac{\Gamma, f : (x : N) \to M, x : N \vdash_{\text{TAS}} m \ : \ M}{\Gamma \vdash_{\text{TAS}} \mathsf{fun}\, f\, x \Rightarrow m \ : \ (x : N) \to M} \text{ ty-fun}$$

$$\frac{\Gamma \vdash_{\text{TAS}} m \ : \ M \quad M \equiv M'}{\Gamma \vdash_{\text{TAS}} m \ : \ M'} \text{ ty-conv}$$

Figure 2·7: Surface Language Type Assignment System

stuck"[Mil78][9]. Given the syntax of the surface language, there is potential for a program to "get stuck" when an argument is applied to a non-function constructor. For example, $\star\ 1_c$ would be stuck since $\star$ is not a function, so it cannot compute when given the argument 1_c. A good type system will make such unreasonable programs impossible.

Type soundness can be shown with a **progress** and **preservation**[10] style proof[11]. The preservation theorem shows that typing information is invariant over evaluation. The progress theorem shows that a step of computation for a well typed term in an empty context will not "get stuck". By iterating these theorems together, it is possible to show that the type system prevents a well typed term from ever reaching a stuck state. For a progress and preservation style proof of a dependently typed language, everything hinges on a suitable definition of the $\equiv$ relation.

The $\equiv$ relation characterizes when terms are "obviously" or "automatically" equal. Because the $\equiv$ relation is usually based on the definition of computation, rather than on observable properties, it is called **definitional equality**[12]. Usually it is desirable to make the definitional equality relation as large as possible, since the programmer in the system will get more equalities "for free". This Chapter will opt for an easier (but less powerful) $\equiv$ relation, since Chapter 3 will propose a way to avoid definitional equality in general.

In a progress and preservation style proof, the $\equiv$ relation should:

- Be reflexive, $m \equiv m$.
- Be symmetric, if $m \equiv m'$ then $m' \equiv m$.

[9] In Milner's original paper, he used "go wrong" instead of "get stuck". In that paper he defined "wrong" as a semantic notion that behaves like a runtime type error.

[10] Also called **Subject Reduction**.

[11] The first proof published in this style is [WF94] though their progress lemma is a bit different from modern presentations. Most relevant textbooks outline forms of this proof for non-dependent type systems. For instance, [Pie02, Part 2], [KSW20], and [Chl17, Chapter 11].

[12] Also called **Judgmental Equality**, since it is defined via judgments.

- Be transitive, if $m \equiv m'$ and $m' \equiv m''$ then $m \equiv m''$.
- Be closed under substitutions and evaluation, for instance if $m \equiv m'$ and $n \equiv n'$ then $m\,[x \coloneqq n] \equiv m'\,[x \coloneqq n']$.
- Distinguish between type formers, for instance $\star \not\equiv (x : N) \to M$.

A particularly clean definition of $\equiv$ arises by equating any terms that share a reduct via a system of parallel reductions ($\Rrightarrow$),

$$\frac{m \Rrightarrow_* n \quad m' \Rrightarrow_* n}{m \equiv m'} \equiv\text{-Def}$$

This relation:

- Is reflexive, by the definition of $\Rrightarrow_*$.
- Is symmetric, automatically.
- Is transitive, if $\Rrightarrow_*$ is confluent (Theorem 2.10).
- Is closed under substitution if $\Rrightarrow_*$ is closed under substitution (Lemma 2.5), and closed under evaluation automatically.
- Distinguishes type constructors, if they are stable under reduction. For instance,
  - for any N, M, $(x : N) \to M \Rrightarrow P$ implies $P = (x : N') \to M'$ (Lemma 2.13)
  - and $\star \Rrightarrow P$ implies $P = \star$ (by the definition of $\Rrightarrow$)
  - then $(x : N) \to M \not\equiv \star$

The system of parallel reductions is defined in Figure 2·8. Parallel reductions are defined to make confluence easy to prove, by allowing the simultaneous evaluation of any available reduction. The only interesting rules are $\Rrightarrow$ -fun-app-red and $\Rrightarrow$ -::-red

$$\frac{m \Rrightarrow m' \quad n \Rrightarrow n'}{(\mathsf{fun}\, f\, x \Rightarrow m)\, n \Rrightarrow m'\,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m', x \coloneqq n']} \Rrightarrow \text{-}\mathsf{fun}\text{-app-red}$$

$$\frac{m \Rrightarrow m'}{m :: M \Rrightarrow m'} \Rrightarrow \text{-}::\text{-red}$$

$$\frac{}{x \Rrightarrow x} \Rrightarrow \text{-var}$$

$$\frac{m \Rrightarrow m' \quad M \Rrightarrow M'}{m :: M \Rrightarrow m' :: M'} \Rrightarrow \text{-}::$$

$$\frac{}{\star \Rrightarrow \star} \Rrightarrow \text{-}\star$$

$$\frac{M \Rrightarrow M' \quad N \Rrightarrow N'}{(x : M) \to N \Rrightarrow (x : M') \to N'} \Rrightarrow \text{-}\mathsf{fun}\text{-ty}$$

$$\frac{m \Rrightarrow m'}{\mathsf{fun}\, f\, x \Rightarrow m \;\Rrightarrow\; \mathsf{fun}\, f\, x \Rightarrow m'} \Rrightarrow \text{-}\mathsf{fun}$$

$$\frac{m \Rrightarrow m' \quad n \Rrightarrow n'}{m\, n \Rrightarrow m'\, n'} \Rrightarrow \text{-}\mathsf{fun}\text{-app}$$

$$\frac{}{m \Rrightarrow_* m} \Rrightarrow_* \text{-refl}$$

$$\frac{m \Rrightarrow_* m' \quad m' \Rrightarrow m''}{m \Rrightarrow_* m''} \Rrightarrow_* \text{-trans}$$

Figure 2·8: Surface Language Parallel Reductions

since they directly perform reductions. The $\Rrightarrow$ -fun-app-red rule recursively reduces a function given an argument. The $\Rrightarrow$ -:: -red rule removes a type annotation, making type annotations definitionally irrelevant. The other rules are structural and allow parallel reductions in any subterms. Repeating parallel reductions zero or more times is written $\Rrightarrow_*$.

While this is a sufficient presentation of definitional equality, other variants of the relation are possible. For instance, it is possible to extend the relation with contextual information, type information, explicit proofs of equality (as in Extensional Type Theory), and uncomputable relations (as in [JZSW10]). It is also common to assume the properties of $\equiv$ hold without proof.

Some lemmas need to quantify over simultaneous substitutions. These simultaneous substitutions will be quantified with the variables σ, τ. For instance, if $\sigma(x) = \star$ and $\sigma(y) = 1_c$, then instead of writing $(x\ y)[x \coloneqq \star, y \coloneqq 1_c] \ = \ (\star\ 1_c)$ we would write $(x\ y)[\sigma] \ = \ (\star\ 1_c)$.

2.4.1 Definitional Equality

We now have enough information to prove the critical properties of definitional equality.

Reflexivity Lemmas

Lemma 2.1. $\Rrightarrow$ *is reflexive.*

The following rule is admissible:

$$\frac{}{m \Rrightarrow m} \Rrightarrow\text{-refl}$$

Proof. By induction on the syntax of m. □

Fact 2.2. $\Rrightarrow_*$ *is reflexive.*

Lemma 2.3. $\equiv$ *is reflexive.*

The following rule is admissible:

$$\frac{}{m \equiv m} \equiv\text{-refl}$$

Proof. Since $\twoheadrightarrow_*$ is reflexive. □

Closure Lemmas

Lemma 2.4. *$\twoheadrightarrow$ is closed under substitutions that parallel reduce.*

Where σ, τ are substitutions. Where $\sigma \twoheadrightarrow \tau$ means for every x, $\sigma(x) \twoheadrightarrow \tau(x)$. The following rule is admissible:

$$\frac{m \twoheadrightarrow m' \quad \sigma \twoheadrightarrow \tau}{m[\sigma] \twoheadrightarrow m'[\tau]} \twoheadrightarrow\text{-sub}$$

Proof. By induction on the $\twoheadrightarrow$ relation, since the substituted term will reduce in the $\twoheadrightarrow$-var case. □

Lemma 2.5. *$\twoheadrightarrow_*$ is closed under substitutions that parallel reduce.*

$$\frac{m \twoheadrightarrow_* m' \quad \sigma \twoheadrightarrow \tau}{m[\sigma] \twoheadrightarrow_* m'[\tau]} \twoheadrightarrow_*\text{-sub}$$

is admissible.

Proof. By induction on the $\twoheadrightarrow_*$ relation. □

Lemma 2.6. *$\equiv$ is closed under substitutions that parallel reduce.*

$$\frac{m \equiv m' \quad \sigma \twoheadrightarrow \tau}{m[\sigma] \equiv m'[\tau]} \equiv\text{-sub}$$

is admissible.

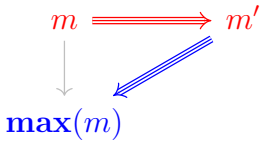
Corollary 2.7. *$\equiv$ is closed under substituted reduction.*

$$\frac{n \twoheadrightarrow_* n'}{m[x := n] \equiv m[x := n']}$$

Proof. By repeated $\twoheadrightarrow_*$-sub and $\equiv$-Def. □

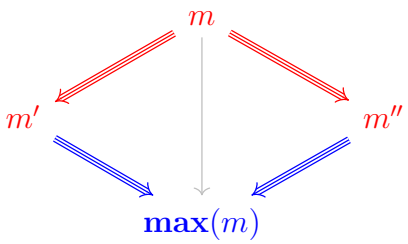
Triangle Property

$\forall m, m'.\ m \Rrightarrow m'$ implies $m' \Rrightarrow \mathbf{max}\,(m)$



Diamond Property

$\forall m, m', m''.\ m \Rrightarrow m' \ \wedge\ m \Rrightarrow m''$ implies $m' \Rrightarrow \mathbf{max}\,(m) \ \wedge\ m'' \Rrightarrow \mathbf{max}\,(m)$



Confluence

$\forall m, n, n'.\ m \Rrightarrow_* n \ \wedge\ m \Rrightarrow_* n'$ implies $\exists n'''.\ n \Rrightarrow_* n''' \ \wedge\ n' \Rrightarrow_* n'''$

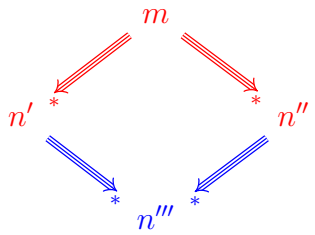


Figure 2·9: Rewriting Diagrams

Transitivity

To prove the transitivity of the $\equiv$ relation, we will first need to prove that $\Rrightarrow_*$ is **confluent**. A relation R is confluent[13] when, for all m, n, n', if mRn and mRn' then there exists n'' such that nRn'' and $n'Rn''$. If a relation is confluent, in a sense, specific reduction choices don't matter since you can alway rejoin at a future destination.

Since type equivalence is defined by parallel reductions we can show confluence

[13] Also called **Church-Rosser**.

$$
\begin{array}{lcll}
\mathbf{max}(& x &) = & x \\
\mathbf{max}(& m :: M &) = & \mathbf{max}\,(m) \\
\mathbf{max}(& \star &) = & \star \\
\mathbf{max}(& (x : M) \to N &) = & (x : \mathbf{max}\,(M)) \to \mathbf{max}\,(N) \\
\mathbf{max}(& \mathsf{fun}\, f\, x \Rightarrow m &) = & \mathsf{fun}\, f\, x \Rightarrow \mathbf{max}\,(m) \\
\mathbf{max}(& (\mathsf{fun}\, f\, x \Rightarrow m)\ n &) = & \mathbf{max}\,(m)\,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow \mathbf{max}\,(m)\,, x \coloneqq \mathbf{max}\,(n)] \\
 & & & \text{otherwise} \\
\mathbf{max}(& m\, n &) = & \mathbf{max}\,(m)\ \mathbf{max}\,(n)
\end{array}
$$

Figure 2·10: The **max** Function

following the proof in [Tak95][14]. The approach is motivated by the diagrams in Figure 2·9.

First, we define a function **max** in Figure 2·10. **max** takes the maximum possible parallel step, such that if $m \Rrightarrow m'$ then $m' \Rrightarrow \mathbf{max}\,(m)$.

Lemma 2.8. *Triangle Property of* $\Rrightarrow$.
If $m \Rrightarrow m'$ *then* $m' \Rrightarrow \boldsymbol{max}(m)$.

Proof. By induction on the derivation $m \Rrightarrow m'$, with the only interesting cases are where a reduction is not taken:

Case 1. In the case of $\Rrightarrow$ -::, $m' \Rrightarrow \mathbf{max}\,(m)$, by $\Rrightarrow$ -::-red.

Case 2. In the case of $\Rrightarrow$ -fun-app, $m' \Rrightarrow \mathbf{max}\,(m)$ by $\Rrightarrow$ -fun-app-red.

□

Lemma 2.9. *Diamond Property of* $\Rrightarrow$.
If $m \Rrightarrow m'$, $m \Rrightarrow m''$, *implies* $m' \Rrightarrow \boldsymbol{max}(m)$, $m'' \Rrightarrow \boldsymbol{max}(m)$.

Proof. By the triangle property. □

Theorem 2.10. *Confluence of* $\Rrightarrow_*$.
If $m \Rrightarrow_* n'$, $m \Rrightarrow_* n''$, *then there exists* n''' *such that* $n' \Rrightarrow n'''$,$n'' \Rrightarrow n'''$.

Proof. By induction. Intuitively by repeated application of the diamond property, "tiling" the interior region. □

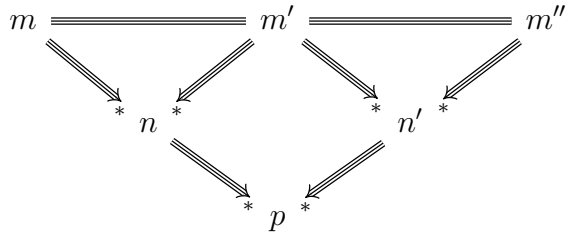
[14] Also well presented in [KSW20].

It follows that

Theorem 2.11. $\equiv$ *is transitive.*

If $m \equiv m'$ *and* $m' \equiv m''$ *then* $m \equiv m''$

Proof. Since if $m \equiv m'$ and $m' \equiv m''$ then by definition for some n, n', $m \Rrightarrow_* n$, $m' \Rrightarrow_* n$ and $m' \Rrightarrow_* n'$, $m'' \Rrightarrow_* n'$. If $m' \Rrightarrow_* n$ and $m' \Rrightarrow_* n'$. Then by confluence there exists some p such that $n \Rrightarrow_* p$ and $n' \Rrightarrow_* p$. By transitivity $m \Rrightarrow_* p$ and $m'' \Rrightarrow_* p$. So by definition $m \equiv m''$.



□

Fact 2.12. $\equiv$ *is an equivalence relation.*

Stability

Next we confirm that type formers are never equated by definitional equality. Specifically, $(x : N) \to M \not\equiv \star$. If type formers are associated, the entire $\equiv$ relation may degenerate. Since definitional equality is defined in terms of reduction, it is sufficient to show that $(x : N) \to M \Rrightarrow (x : N') \to M'$. We will prove slightly stronger lemmas about reduction that confirms this fact since it will be useful later.

Lemma 2.13. *Stability of* $\to$ *over* $\Rrightarrow_*$.

$\forall N, M, P. (x : N) \to M \Rrightarrow_* P$ *implies* $\exists N', M'. P = (x : N') \to M' \wedge N \Rrightarrow_* N' \wedge M \Rrightarrow_* M'$.

Proof. By induction on $\Rrightarrow_*$:

Case 1. $\Rrightarrow_*$ -refl follows directly.

Case 2. $\Rrightarrow_*$ -trans follows via the induction hypothesis and noting only the $\Rrightarrow$ -fun-ty rule is possible as a step.

□

Therefore, the we can derive an important fact about $\equiv$.

Corollary 2.14. *Stability of $\to$ over $\equiv$.*
The following rule is admissible:

$$\frac{(x : N) \to M \equiv (x : N') \to M'}{N \equiv N' \quad M \equiv M'}$$

Proof. By the definition of $\equiv$ and the lemma above. □

2.4.2 Preservation

A useful property of a type system is that reduction preserves type[15].

We need several more technical lemmas before we can prove that $\Rightarrow_*$ is type preserving. These lemmas will almost always be justified by induction on typing derivations.

Structural Properties

Theorem 2.15. *Context Weakening.*
The following rule is admissible:

$$\frac{\Gamma \vdash_{\text{TAS}} n : N}{\Gamma, \Gamma' \vdash_{\text{TAS}} n : N}$$

Proof. By induction on typing derivations. □

Lemma 2.16. *Substitution preserves types.*
The following rule is admissible:

$$\frac{\Gamma \vdash_{\text{TAS}} n : N \quad \Gamma, x : N, \Gamma' \vdash_{\text{TAS}} m : M}{\Gamma, \Gamma' [x := n] \vdash_{\text{TAS}} m [x := n] : M [x := n]}$$

Proof. By induction on typing derivations:

Case 1. ty-var follows by weakening the substituted term.

Case 2. ty-conv follows from $\equiv$-Def and that $\Rightarrow_*$ is closed under substitution.

[15]Similar proofs for dependent type systems can be found in [Luo94, Chapter 3], [Miq01, Section 3.1](including eta expansion in an implicit system), [SCA+12, appendix], and formalized in the the examples of Autosubst[STS15].

$$\frac{}{\Diamond \equiv \Diamond} \equiv\text{-ctx-empty}$$

$$\frac{\Gamma \equiv \Gamma' \quad M \equiv M'}{\Gamma, x : M \equiv \Gamma', x : M'} \equiv\text{-ctx-ext}$$

Figure 2·11: Definitionally Equal Contexts

Case 3. All other cases follow directly or by induction.

□

We extend the notion of definitional equality to contexts in Figure 2·11 so that we can ignore reductions in the context. When contexts are convertible, typing judgments still hold.

Lemma 2.17. *Contexts that are equivalent preserve types.*
The following rule is admissible:

$$\frac{\Gamma \vdash_{\text{TAS}} n : N \quad \Gamma \equiv \Gamma'}{\Gamma' \vdash_{\text{TAS}} n : N}$$

Proof. By induction over typing derivations:

Case 1. ty-var follows since $\equiv$ is symmetric.

Case 2. All other cases follow directly or by induction.

□

Inversion Lemmas

In the preservation proof we will need to reason backwards about the typing judgments implied by a typing derivation of a term with specific syntax. These are conventionally called inversion lemmas since they mirror the typing judgments. But unlike typing derivations which take in typing derivations for subterms, inversion lemmas conclude

typing derivations for subterms. For this Chapter we will only need to prove an inversion lemma for functions.

In a dependently typed setting, inversion lemmas cannot be proven directly by induction. The induction hypothesis must be extended over definitional equality.

Lemma 2.18. fun-*Inversion (generalized).*

$$\frac{\Gamma \vdash_{\text{TAS}} \mathsf{fun}\, f\, x \Rightarrow m \,:\, P \quad P \equiv (x : N) \to M}{\Gamma, f : (x : N) \to M, x : N \vdash_{\text{TAS}} m : M}$$

is admissible.

Proof. By induction on typing derivations:

Case 1. ty-fun follows by the stability of ty-fun and preservation of contexts.

Case 2. ty-conv follows by transitivity of $\equiv$ and induction.

Case 3. All other cases are impossible!

□

This allows us to conclude the more straightforward corollary:

Corollary 2.19. fun-*Inversion.*

$$\frac{\Gamma \vdash_{\text{TAS}} \mathsf{fun}\, f\, x \Rightarrow m \,:\, (x : N) \to M}{\Gamma, f : (x : N) \to M, x : N \vdash_{\text{TAS}} m : M}$$

Proof. By noting that $(x : N) \to M \equiv (x : N) \to M$, by reflexivity. □

Theorem 2.20. $\Rrightarrow$ *Preserves types.*
The following rule is admissible:

$$\frac{\Gamma \vdash_{\text{TAS}} m : M \quad m \Rrightarrow m'}{\Gamma \vdash_{\text{TAS}} m' : M}$$

Proof. By induction on the typing derivation $\Gamma \vdash_{\text{TAS}} m : M$, specializing on $m \Rrightarrow m'$:

Case 1. ty-: : when $\Rrightarrow$ -: :, ($m = n : N$, $m' = n' : N'$, $M = N$ for some n, N, n', and N') we must show $\Gamma \vdash_{\text{TAS}} n' :: N' : N$ from $\Gamma \vdash_{\text{TAS}} n : N$, $\Gamma \vdash_{\text{TAS}} N : \star$,

$n \Rrightarrow n'$, and $N \Rrightarrow N'$.

$\Gamma \vdash_{\text{TAS}} N' : \star$	by induction
$\Gamma \vdash_{\text{TAS}} n' \,:\, N$	by induction
$N \equiv N'$	by $N \Rrightarrow N'$
$\Gamma \vdash_{\text{TAS}} n' \,:\, N'$	by ty-conv
$\Gamma \vdash_{\text{TAS}} n' :: N' :\, N'$	by ty-::
$N' \equiv N$	by symmetry of $\equiv$
$\Gamma \vdash_{\text{TAS}} n' :: N' :\, N$	by ty-conv

Case 2. ty-fun-ty when $\Rrightarrow$ -fun-ty can be shown with preservation of contexts

Case 3. ty-fun-app when $\Rrightarrow$ -fun-app-red, we must show $\Gamma \vdash_{\text{TAS}} m' \,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m', x \coloneqq n'] : M \,[x \coloneqq n]$ from $\Gamma \vdash_{\text{TAS}} n \,:\, N$, $\Gamma \vdash_{\text{TAS}} \mathsf{fun}\, f\, x \Rightarrow m \,:\, (x : N) \to M$, $m \Rrightarrow m'$, and $n \Rrightarrow n'$.

$\mathsf{fun}\, f\, x \Rightarrow m \Rrightarrow \mathsf{fun}\, f\, x \Rightarrow m'$	by $\Rrightarrow$ -fun
$\Gamma \vdash_{\text{TAS}} \mathsf{fun}\, f\, x \Rightarrow m' \,:\, (x : N) \to M$	by induction
$\Gamma, f : (x : N) \to M, x : N \vdash_{\text{TAS}} m'$	by fun-inversion
$\Gamma \vdash_{\text{TAS}} n' \,:\, N$	by induction
$\Gamma \vdash_{\text{TAS}} m' \,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m', x \coloneqq n'] : M \,[x \coloneqq n']$	by substitution preservation
$M \,[x \coloneqq n] \equiv M \,[x \coloneqq n']$	by substitution by $\Rightarrow$
$\Gamma \vdash_{\text{TAS}} m' \,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow m', x \coloneqq n'] : M \,[x \coloneqq n]$	by ty-conv

Case 4. ty-fun-app when $\Rrightarrow$ -fun-app, we must show $\Gamma \vdash_{\text{TAS}} m'\, n' \,:\, M \,[x \coloneqq n]$ from $\Gamma \vdash_{\text{TAS}} n \,:\, N$, $\Gamma \vdash_{\text{TAS}} m \,:\, (x : N) \to M$, $m \Rrightarrow m'$, $n \Rrightarrow n'$.

$n \Rrightarrow n'$

$\Gamma \vdash_{\text{TAS}} m' \,:\, (x : N) \to M$	by induction
$\Gamma \vdash_{\text{TAS}} n' \,:\, N$	by induction
$\Gamma \vdash_{\text{TAS}} m'\, n' \,:\, M \,[x \coloneqq n']$	ty-fun-app
$M \,[x \coloneqq n] \equiv M \,[x \coloneqq n']$	by substitution by $\Rightarrow$
$\Gamma \vdash_{\text{TAS}} m'\, n' \,:\, M \,[x \coloneqq n]$	ty-conv

Case 5. All other cases follow directly or by induction

□

2.4.3 Progress

The second key theorem to show in this style of proof is called progress. The progress theorem states: for a well typed term in an empty context, then a further call-by-value[16] step can be taken or computation is finished.

Fact 2.21. $\rightsquigarrow$ *implies* $\Rrightarrow$.

The following rule is admissible:

$$\frac{m \rightsquigarrow m'}{m \Rrightarrow m'}$$

Thus $\rightsquigarrow$ also preserves types.

We will need a technical lemma that determines the syntax of a value in an empty context. These lemmas are usually called canonical form lemmas. Since the language of this chapter is so minimal, we only need to characterize the canonical form of functions.

Lemma 2.22. fun-*Canonical form (generalized).*

If $\vdash_{\mathrm{TAS}} v : P$ *and* $P \equiv (x : N) \to M$ *then* $v = \mathsf{fun}\, f\, x \Rightarrow m$, *for some* m.

Proof. By induction on the typing derivation,

Case 1. ty-fun follows immediately

Case 2. ty-conv by the equivalence of $\equiv$ and induction

Case 3. ty-$\star$, ty-fun-ty are impossible, by the stability of $\equiv$

Case 4. other rules are impossible, since they do not type values

□

[16] It is tempting to use $\Rrightarrow$ as the main notion of reduction, since it corresponds to $\equiv$. However, since $\Rrightarrow$ is reflexive, no expression could get stuck.

As a corollary,

Corollary 2.23. fun-*Canonical form.*
If $\vdash_{\text{TAS}} v : (x : N) \to M$ *then* $v = \mathsf{fun}\, f\, x \Rightarrow m$.

Finally we can prove the progress theorem.

Theorem 2.24. *Progress.*
If $\vdash_{\text{TAS}} m : M$ *then* m *is a value or there exists* m' *such that* $m \rightsquigarrow m'$

Proof. As usual this follows form induction on the typing derivation

Case 1. ty-$\star$, $\star$ is a value.

Case 2. ty-var, impossible in an empty context!

Case 3. ty-conv, by induction.

Case 4. ty-::, we have a typing derivation concluding $\vdash_{\text{TAS}} m :: M : M$. By induction, m is a value or there exists m' such that $m \rightsquigarrow m'$:

- *Case* i. If m is a value, then $m :: M \rightsquigarrow m$.
- *Case* ii. If $m \rightsquigarrow m'$,then $m :: M \rightsquigarrow m' :: M$.

Case 5. ty-fun-ty, $(x : M) \to N$ is a value.

Case 6. ty-fun, $\mathsf{fun}\, f\, x \Rightarrow m$ is a value.

Case 7. ty-fun-app, we have a typing derivation concluding $\vdash_{\text{TAS}} m\, n : M\,[x \coloneqq n]$ with the premises $\vdash_{\text{TAS}} m : (x : N) \to M$, $\Gamma \vdash_{\text{TAS}} n : N$. By induction, m is a value or there exists m' such that $m \rightsquigarrow m'$. By induction, n is a value or there exists n' such that $n \rightsquigarrow n'$.

- *Case* i. if $m \rightsquigarrow m'$, then $m\, n \rightsquigarrow m'\, n$
- *Case* ii. if m is a value, and $n \rightsquigarrow n'$, then $m\, n \rightsquigarrow m\, n'$
- *Case* iii. if m is a value, and n is a value, then $m = \mathsf{fun}\, f\, x \Rightarrow p$ by canonical forms of functions. The term steps $(\mathsf{fun}\, f\, x \Rightarrow p)\, n \rightsquigarrow p\,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow p, x \coloneqq n]$.

□

Progress via call-by-value can be seen as a specific sub-strategy of $\Rrightarrow_*$. An interpreter is always free to take any $\Rrightarrow_*$, but if it is unclear which $\Rrightarrow_*$ to take, either it is a value and no further steps are required, or can fall back on $\rightsquigarrow$ until the computation is a value.

2.4.4 Type Soundness

The surface language has type soundness:

Theorem 2.25. *Type Soundness.*
If $\vdash_{\text{TAS}} m : M$ *and* $m \rightsquigarrow_* m'$ *then* m' *cannot be* ***Stuck****.*

Proof. This follows by iterating the progress and preservation lemmas. □

2.4.5 Type Checking Is Impractical

This type system is inherently non-local. No type annotations are ever required to form a typing derivation. That means that a type checking algorithm that attempted to type check every well typed TAS term would need to guess the types of intermediate terms. For instance, a large function might use its argument different ways in different locations, as in

$$\begin{aligned}&\lambda f \Rightarrow \\ &\quad ...f\, 1_c\, true_c \\ &\quad ...f\, 0_c\, 1_c \\ &\quad ...\end{aligned}$$

What is the type of f? One possibility is $f : (n : \mathbb{N}) \to n \star (\lambda - \Rightarrow \mathbb{N}_c)\ \mathbb{B}_c \to ...$. But there are many other possibilities. Worse, if there is an error, it may be impossible

to localize to a specific region of that expression. To make a practical type checker the user will need to include some type annotations.

2.5 Bidirectional Surface Language

There are many possible ways to localize the type checking process. We could ask that all variables be annotated at binders. This is enticing from a theoretical perspective, since it matches how type contexts are built up.

However note that, our proof of $\neg 1_c \doteq_{\mathbb{N}_c} 0_c$ will look like:

$$\lambda pr\underline{: 1_c \doteq_{\mathbb{N}_c} 0_c} \Rightarrow$$
$$\Big(\lambda n\underline{: (C : (\mathbb{N}_c \to \star)) \to C\, 1_c \to C\, 0_c} \Rightarrow n \star (\lambda - :\underline{\star \Rightarrow Unit_c})\ \bot_c\Big)\ tt_c$$

More than half of the term is type annotations! Annotating every binding site requires a lot of redundant information. Luckily there's a better way.

2.5.1 Bidirectional Type Checking

Bidirectional type checking is a popular form of lightweight type inference, which strikes a good compromise between the required type annotations and the simplicity of the procedure, while allowing for localized errors[17]. In the usual bidirectional typing schemes, annotations are only needed at the top-level, or around a function that is directly applied to an argument[18]. For example $(\lambda x \Rightarrow x + x)7$ would need to be written $((\lambda x \Rightarrow x + x) :: \mathbb{N} \to \mathbb{N})\, 7$ to type check bidirectionally. Since programmers rarely write functions that are immediately evaluated, this style of type checking

[17] [Chr13] is a good tutorial, [DK21] is a survey of the technique.
[18] More generally when an elimination reduction is possible.

usually only needs top-level functions to be annotated[19]. In fact, almost every example in Figure 2·3 has enough annotations to type check bidirectionally without further information.

Bidirectional type checking is accomplished by separating the TAS typing judgments into two mutual judgments:

- **Type Inference** where type information propagates out of a term, $\overset{\rightarrow}{:}$ in our notation.
- **Type Checking** judgments where a term is checked against a type, $\overset{\leftarrow}{:}$ in our notation.

This allows typing information to flow from the "outside in" for type checking judgments and "inside out" for the type inference judgments. Check mode can be induced by the programmer with a type annotation. When an inference meets a check, a conversion verifies that the types are definitionally equal. This has the advantage of precisely limiting where the ty-conv rule can be used, since conversion checking is usually an inefficient part of dependent type checking.

This enforced flow of information results in a system that localizes type errors. If a type was inferred, it was unique, so it can be used freely. Checking judgments force terms that could have multiple typings in the TAS to have at most one type.

The surface language supports bidirectional type-checking over the syntax with the rules in Figure 2·12. The rules are almost the same as before, except that typing direction is now explicit in the judgment.

As mentioned, bidirectional type checking handles higher order functions very well. For instance, the expression $\vdash (\lambda x \Rightarrow x\,(\lambda y \Rightarrow y)\,2) \overset{\leftarrow}{:} ((\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ checks because $\vdash (\lambda y \Rightarrow y) \overset{\leftarrow}{:} (\mathbb{N} \to \mathbb{N})$ and $\vdash 2 \overset{\leftarrow}{:} \mathbb{N}$.

[19]Even in Haskell, with full Hindley-Milner type inference, top level type annotations are encouraged.

$$\frac{x : M \in \Gamma}{\Gamma \vdash x \overrightarrow{:} M}\ \overrightarrow{ty}\text{-var}$$

$$\frac{}{\Gamma \vdash \star \overrightarrow{:} \star}\ \overrightarrow{ty}\text{-}\star$$

$$\frac{\Gamma \vdash m \overleftarrow{:} M \quad \Gamma \vdash M \overleftarrow{:} \star}{\Gamma \vdash m :: M \overrightarrow{:} M}\ \overrightarrow{ty}\text{-}::$$

$$\frac{\Gamma \vdash M \overleftarrow{:} \star \quad \Gamma, x : M \vdash N \overleftarrow{:} \star}{\Gamma \vdash (x : M) \to N \overrightarrow{:} \star}\ \overrightarrow{ty}\text{-}\mathsf{fun}\text{-ty}$$

$$\frac{\Gamma \vdash m \overrightarrow{:} (x : N) \to M \quad \Gamma \vdash n \overleftarrow{:} N}{\Gamma \vdash m\, n \overrightarrow{:} M\,[x \coloneqq n]}\ \overrightarrow{ty}\text{-}\mathsf{fun}\text{-app}$$

$$\frac{\Gamma, f : (x : N) \to M, x : N \vdash m \overleftarrow{:} M}{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow m \overleftarrow{:} (x : N) \to M}\ \overleftarrow{ty}\text{-}\mathsf{fun}$$

$$\frac{\Gamma \vdash m \overrightarrow{:} M \quad M \equiv M'}{\Gamma \vdash m \overleftarrow{:} M'}\ \overleftarrow{ty}\text{-conv}$$

Figure 2·12: Surface Language Bidirectional Typing Rules

Unlike the undirected judgments of the type assignment system, the inference rule of the bidirectional system does not associate definitionally equivalent types. The inference judgment is unique up to syntax! For example $x : Vec\,3 \vdash x \overrightarrow{:} Vec\,3$, but $x : Vec\,3 \not\vdash x \overrightarrow{:} Vec\ (1 + 2)$. This could cause unexpected behavior around function applications. For instance, if $\Gamma \vdash m \overrightarrow{:} \mathbb{N} \to \mathbb{N}$ then $\Gamma \vdash m\,7 \overrightarrow{:} \mathbb{N}$ will infer, but only because the $\to$ is in the head position of the type $\mathbb{N} \underline{\to} \mathbb{N}$. If $\Gamma \vdash m \overrightarrow{:} ((\mathbb{N} \to \mathbb{N}) :: \star)$ then $::$ is in the head position of $(\mathbb{N} \to \mathbb{N}) \underline{::} \star$ and $\Gamma \not\vdash m\,7 \overrightarrow{:} \mathbb{N}$ will will not infer.

A similar issue exists with check rules around function definitions. For instance, $\vdash ((\lambda x \Rightarrow x) :: \mathbb{N} \to \mathbb{N}) \ \overrightarrow{:} \mathbb{N} \to \mathbb{N}$ will infer, but if computation blocks the $\to$ from being in the head position, inference will be impossible. As in the expression, $((\lambda x \Rightarrow x) :: ((\mathbb{N} \to \mathbb{N}) \underline{::} \star))$ which will not infer.

For these reasons, many presentations of bidirectional dependent typing will evaluate the types needed for $\overleftarrow{ty} - \mathsf{fun}$, and $\overrightarrow{ty}$-fun-app into weak-head-normal-form[20].

[20] As in [Coq96].

With that caveat, this document opts for an unevaluated version of the rules to make some properties easier to present and prove. Specifically, this will allow us to present a terminating elaboration procedure in Chapter 3[21].

Though this chapter opts for a simple presentation of bidirectional type checking, it is possible to take the ideas of bidirectional typing very far. More advanced bidirectional implementations such as Agda[Nor07] even perform unification as part of their bidirectional type checking.

2.5.2 The Bidirectional System is Type Sound

It is possible to prove bidirectional type systems are type sound directly[NM05]. But it would be difficult for the system described here since type annotations evaluate away, contradicting a potential preservation lemma. Alternatively we can show that a bidirectional typing judgment implies a type assignment system typing judgment.

Theorem 2.26. *Bidirectional implies TAS.*
If $\Gamma \vdash m \overrightarrow{:} M$ *then* $\Gamma \vdash m : M$.
If $\Gamma \vdash m \overleftarrow{:} M$ *then* $\Gamma \vdash m : M$.

Proof. by mutual induction on the bidirectional typing derivations. □

Therefore the bidirectional system is also type sound.

2.5.3 The TAS System Is Weakly Annotatable by the Bidirectional System

In bidirectional systems, **annotatability**[22] is the property that any expression that types in a TAS will type in the bidirectional system with only additional annotations. To save space we can instead show that for every well typed TAS expression there is

[21] The prototype implementation uses the more conventional weak-head-normal-form check up to some "time bound", which avoids the issues above from the programmer's perspective but is more messy in theory.

[22] Also called **completeness**.

an equivalent bidirectional expression, though annotations may need to be added (or removed). We will call this property **weak annotatability**.

Theorem 2.27. *Weak annotatability.*
If $\Gamma \vdash m : M$ *then* $\Gamma \vdash m' \overleftarrow{:} M'$, $m \equiv m'$ *and* $M \equiv M'$ *for some* m' *and* M'.
If $\Gamma \vdash m : M$ *then* $\Gamma \vdash m' \overrightarrow{:} M'$, $m \equiv m'$ *and* $M \equiv M'$ *for some* m' *and* M'.

Proof. By induction on the typing derivation, adding and removing annotations at each step that are convertible with the original term. □

2.6 Absent Logical Properties

When type systems are used as logics, it is desirable that:

- There exists a type that is uninhabited in the empty context, so the system is **logically consistent**[23].
- Type checking is decidable.

Neither the TAS system nor the bidirectional systems have these properties[24].

2.6.1 Logical Inconsistency

The surface language is logically inconsistent, since every type is inhabited.

Example 2.28. Every Type is Inhabited (by recursion).
$\mathsf{fun}\, f\, x \Rightarrow f\, x : \bot_c$

It is also possible to encode Girard's paradox, producing another source of logical unsoundness.

Example 2.29. Every Type is Inhabited (by type-in-type).

[23] Also called **logically sound**.

[24] These properties are usually shown by showing that the computation that generates definitional equality is normalizing. A proof for a logically consistent system can be found in [Luo94, Chapter 4]. Another excellent tutorial can be found in [Cas14, Chapter 2]

A subtle form of recursive behavior can be built out of Gerard's paradox[Rei89], but this behavior is no worse than the unrestricted recursion already allowed. While it is possible to "prove" logically incorrect theorems this way by accident, doing so seems rare in practice.

Operationally, logical inconsistency will be recognized by programmers as non-termination. Non-termination seems not to matter for programming languages in practice. For instance, in ML the type `f : Int → Int` does not imply the termination of `f 2`. While unproductive non-termination is always a bug, it seems an easy bug to detect and fix when it occurs in programs. In mainstream languages, types help to communicate the intent of termination, even though termination is not guaranteed by the type system. Importantly, no computation is prevented in the surface language in order to preserve logical consistency. Due to the halting problem, there is no way to allow all the terminating computations and exclude all the nonterminating computations. A tradeoff must be made, and programmers likely care more about having all possible computations than preventing non-termination. Therefore, logical unsoundness seems suitable for a dependently typed programming language.

While the surface language supports proofs, not every term typed in the surface language is a proof. Terms can still be called proofs as long as the safety of recursion and type-in-type are checked externally. In this sense, the listed example inequalities are proofs, as they make no use of general recursion (so all recursions are well founded) and universes are used in a safe way (for instance predicative universe levels could be assigned). In an advanced implementation, an automated process could supply warnings when constructs are used in potentially unsafe ways. Traditional software testing can be used to discover if there are actual proof bugs. Even though the type system is not logically consistent, type checking still eliminates a large class of possible mistakes. While it is possible to make an error, this is true of much more popular

proof mediums like blackboards, or typeset LaTeX.

Finally by separating non-termination concerns from the core of the theory, this architecture is resilient to change. If the termination checker is updated in Coq, there is some chance older proof scripts will no longer type check. With the architecture proposed here, code will always have the same static and dynamic behavior, though our understanding of termination might change.

Type Checking is Undecidable

Theorem 2.30. *Type checking is undecidable.*

Proof. Given an expression of type $q : Unit$ defined in PCF[25], that expression can be encoded into the surface system as $m_q : Unit_c$, such that if q reduces to the canonical $Unit$ then $m_q \Rrightarrow_* \lambda A.\lambda a.a$

$\vdash \star : m_q \star \star$ type-checks by conversion exactly when q halts.

If there is a procedure to decide type checking then we can decide exactly when a PCF expression of type $Unit$ halts. Since checking if a PCF expression halts is undecidable, type checking is undecidable.

□

Again the root of the problem is the non-termination that results by allowing as many computations as possible, which seem necessary in a realistic programming language.

Luckily, undecidability of type checking is not as bad as it sounds for several reasons. First, the pathological terms that cause non-terminating conversion are rarely created on purpose or even by accident. In the bidirectional system, conversion checks will only happen at limited positions, and it is possible to use a counter to warn or give errors at code positions that do not convert because normalization takes too long. Heuristic methods of conversion checking have worked so surprisingly well

[25]The system PCF is a simply typed lambda calculus with recursion and a few built in data types. Formal definitions can be found in many textbooks, such as [Str06].

in our prototypes that implementing the counter limited equality was never a pressing concern.

Many dependent type systems, such as Agda, Coq, and Lean, aspire to decidable type checking. However, these systems allow extremely fast growing functions to be encoded (such as Ackerman's function). A fast growing function can generate a very large index that can be used to check some concrete but unpredictable property, (how many Turing machines whose code is smaller then n halt in n steps?). When this kind of computation is lifted to the type level, type checking is computationally infeasible, to say the least.

Decidability of type checking is often used as a proxy for efficiency of type checking. However, it may be a poor measure of efficiency for the kinds of programs and proofs that are likely to occur.

Many mainstream programming languages have undecidable type checking. If a language admits a sufficiently powerful macro or preprocessor system that can modify typing, this would make type checking undecidable (this makes the type system of C, C++, Scala, and Rust undecidable). Unless type features are considered very carefully, they can cause undecidable type checking (Java generics[Gri17], C++ templates[Vel03], and OCaml modules[Ros99], make type checking undecidable in those languages). Haskell may be the most popular statically typed language with decidable type checking (and even then, popular GHC compiler flags, such as the aptly named `UndecidableInstances`, make type checking undecidable). Even the Hindley-Milner type checking algorithm that underlies Haskell and ML, has a worst case complexity that is double exponential, which under normal circumstances would be considered intractable.

In practice these theoretical concerns are irrelevant since programmers are not giving the compiler "worst case" code. Even if they did, the worst that can

happen is the type checker will hang in the type checking process. When this happens in a mainstream language, programmers can fix their code, modify or remove macros, or add typing annotations. Programmers in conventional languages are already entrusted with almost unlimited power over their programming environments. Programs regularly delete files, read and modify sensitive information, and send emails (some of these are even possible from within the language's macro systems). Relatively speaking, undecidable type checking is not a programmer's biggest concern.

Most importantly for the system described in this thesis, users are expected to use the elaboration procedure defined in the next Chapter instead of the bidirectional type checking described here. Unlike the bidirectional described in this section, the elaboration of Chapter 3 is technically decidable.

2.7 Related Work

2.7.1 Bad Logics, ok Programming Languages?

Unsound logical systems that work as programming languages go back to at least Church's lambda calculus which, was originally intended to be part of a foundation for mathematics[26]. In the 1970s, Martin-Löf proposed a system with type-in-type[ML71] that was shown logically unsound by Girard (as described in the introduction of [ML72]). In the 1980s, Cardelli explored the domain semantics of a system with general recursive dependent functions and type-in-type[Car86]. Independently, Viggo and Stoltenberg-Hansen[PSH90] explored the domain semantics of Martin-Löf's type theory with a fixed point operator.

The first progress and preservation style proof of type soundness for a language with general recursive dependent functions and type-in-type seem to come from the Trellys Project[SCA+12]. At the time, their language had several additional

[26] "There may, indeed, be other applications of the system than its use as a logic."[Chu32, p.349]

features not included in the surface language presented here. Additionally, the surface language uses a simpler notion of definitional equality resulting in a simpler proof of type soundness. Later work in the Trellys Project[CSW14, Cas14] used modalities to separate terminating and non-terminating fragments of the language, to allow both general recursion and logically sound reasoning. In general, the surface language has been deeply informed by the Trellys project[SCA+12, CSW14, Cas14, SW15, Sjö15] and the Zombie language[27] it produced.

2.7.2 Implementations

Several programming language implementations support features of the surface language without a proof of type soundness. Pebble[BL84] was a very early language with dependent types, though conversion did not associate types that differ only in variable naming[28]. Coquand implemented an early bidirectional algorithm to type-check a language with type-in-type[Coq96]. Cayenne[Aug98] is a Haskell-like language that combines dependent types with type-in-type and non-termination. ΠΣ[ADLO10] is a language with type-in-type and several features for a dependently typed core calculus outlined here. Like here, ΠΣ advocates separating termination concerns from type soundness concerns, though type soundness was never established.

Agda supports general recursion and type-in-type with compiler flags. Idris supports similar "unsafe" features.

2.7.3 Other Dependent Type Systems

There are many flavors of dependent type systems that are similar in spirit to the language presented here but maintain logical soundness at the expense of computation.

[27] `https://github.com/sweirich/trellys`
[28] According to [Rei89].

The Calculus of Constructions (CC, CoC)[CH88] is one of the first minimal dependent type systems. It contains shockingly few rules but can express a wide variety of constructions via parametric encodings. The system does not allow type-in-type, instead $\star$[29] lives in a larger universe $\Box$ ($\star : \Box$), where $\Box$ is not itself a type. Even though the Calculus of Constructions does not allow type-in-type, it is still **impredicative** in the sense that function types can quantify over $\star$ while still being in $\star$. For instance, the polymorphic identity $id : (X : \star) \to X \to X$ has type $\star$ so the polymorphic identity can be applied to itself, $id\ ((X : \star) \to X \to X)\ id$. From the perspective of the surface language this impredictivity is modest[30] but still causes issues in the presence of classical logical assumptions.

Several other systems were developed that directly extended or modified the Calculus of Constructions. The Extended Calculus of Constructions (ECC)[Luo90, Luo94], extends the Calculus of Constructions with a predicative hierarchy of universes and dependent pair types. The Implicit Calculus of Constructions (ICC)[Miq01, BB08] presents an extrinsic typing system[31]. Unlike the type assignment system presented in this Chapter, the Implicit Calculus of Constructions allows implicit qualification over terms in addition to explicit quantification over terms (also a hierarchy of universes, and a universe of "sets"). Other extensions to the Calculus of Constructions that are primarily concerned with data will be surveyed in Chapter 4.

The lambda cube is a system for relating 8 interesting typed lambda calculi to each other. Presuming terms should always depend on terms, there are 3 additional dimensions of dependency: term depending on types, types dependent on types, and types depending on terms. The simply typed lambda calculus has only term

[29] Called **prop**, for proposition.

[30] As in [ADLO10], we can say "the surface language is very impredictive".

[31] Sometimes called **Curry-style**, in contrast to intrinsic type systems which are sometimes called **Church-style**.

dependency. System F additionally allows types to depend on types. The Calculus of Constructions has all forms of dependency[32].

Pure Type Systems (PTS)[33] generalizes the lambda cube to allow any number of type universes with any forms of dependency. Notably this includes the system with one type universe where type-in-type holds. Universe hierarchies can also be embedded in a PTS. The system described in this Chapter is almost a PTS, except that it contains unrestricted recursion and the method of type annotation is different. All pure type systems such as System F and the Calculus of Constructions have corresponding terms in the surface language by collapsing their type universes into the surface language type universe.

As previously mentioned, Martin-Löf Type Theory (MLTT)[ML72] is one of the oldest frameworks for dependent type systems. MLTT is designed to be open so that new constructs can be added with the appropriate introduction, elimination, computation, and typing rules. The base system comes with a predicative hierarchy of universes and at least dependently typed functions and a propositional equality type. The system has two flavors characterized by its handling of definitional equality. If types are only identified by computation (as the system described in this Chapter) it is called Intensional Type Theory (ITT). If the system allows proofs of equality to associate types, it is called Extensional Type Theory (ETT). Since MLTT is open ended, the Calculus of Constructions can be added to it as a subsystem[AH04, Hof97a]. Many of the examples from this Chapter are adapted from examples that were collected with early versions of MLTT[ML71].

[32]Recommended reading [SU06, Chapter 14].
[33]Previously called **Generalized Type Systems**.

Chapter 3

The Dependent Cast System

Chapter 2 outlined a minimal dependent type system, called the surface language. Like all dependent type systems, the surface language has a fundamental problem: definitional equalities are pervasive and unintuitive.

For instance, the motivating example from Chapter 1 can be stated more precisely in terms of the surface language. Recall, dependent types can prevent an out-of-bounds error when extracting the first element of a length indexed list.

$$\begin{aligned}
\texttt{Vec} &: * \to \mathbb{N}_c \to *, \\
\texttt{rep} &: (X : *) \to X \to (y : \mathbb{N}_c) \to \texttt{Vec}\, X\, y, \\
\texttt{head} &: (X : *) \to (y : \mathbb{N}_c) \to \texttt{Vec}\, X\ (1_c +_c y) \to X
\end{aligned}$$

$$\vdash \lambda x \Rightarrow \texttt{head}\, \mathbb{B}_c\, x\ (\texttt{rep}\, \mathbb{B}_c\, true_c\ (1_c +_c x)) \ :\ \mathbb{N}_c \to \mathbb{B}_c$$

Where `head` is a function that expects a list of length $1_c +_c y$, making it impossible for `head` to inspect an empty list.

Unfortunately, the following will not type check in the surface language,

$$\nvdash \lambda x \Rightarrow \texttt{head}\, \mathbb{B}_c\, x\ (\texttt{rep}\, \mathbb{B}_c\, true_c\ (x +_c 1_c)) \ :\ \mathbb{N}_c \to \mathbb{B}_c$$

While "obviously" $1 + x = x + 1$, in the surface language, definitional equality does not associate these two terms, $1_c +_c x \not\equiv x +_c 1_c$.

This Chapter will handle the issue of definitional equalities by avoiding them. The system will optimistically assume equalities implied by the programmer and deal with incorrect equalities at runtime in a principled way. This will be done with the two systems described in this Chapter:

- The **cast language**, a dependently typed language with embedded checks that have evaluation behavior.
- The **elaboration procedure** that transforms appropriate untyped surface syntax into a well cast expressions.

The cast language's type system will be called the **cast system** to distinguish it from the two type systemd already introduced in Chapter 2. Similarly expressions that type in the cast system will be called **well cast**.

The presentation in this Chapter mirrors Chapter 2. The cast system plays the role of the type assignment system, while the elaboration procedure corresponds with the bidirectional system.

We show that a novel form of type soundness holds, that we call **cast soundness**. Instead of "well typed terms don't get stuck", we prove "well cast terms don't get stuck without blame".

Blame will carry the necessary information to construct a reasonable runtime error message. It is related to the similarly named notion from contract and monitor systems. Several desirable properties, modeled on the gradual guarantee of gradual types, relate the cast system elaboration and the bidirectional system of Chapter 2.

3.1 Cast Language

The syntax for the cast language can be found in Figure 3·1. By design the cast language is almost identical to the surface language except that the cast construct has been added and annotations have been removed.

source locations,
ℓ
variable contexts,

Γ,H	::=	$\Diamond \mid \Gamma, x : A$	
expressions,			
a,b,A,B	::=	x	
	$\mid$	$a ::_{A,\ell,o} B$	cast
	$\mid$	$\star$	
	$\mid$	$(x : A) \to B$	
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow b$	
	$\mid$	$b\, a$	
observations,			
o	::=	.	
	$\mid$	$o.Arg$	function type-arg
	$\mid$	$o.Bod_a$	function type-body

Figure 3·1: Cast Language Syntax

The cast language can assume type equalities on top of terms, $A = B$, with a cast, $a ::_{A,\ell,o} B$ given:

- An underlying term a.
- A source location ℓ where it was asserted.
- A concrete observation o that refines the source location ℓ.
- The type of the underlying a term A.
- The expected type of the term B.

Every time there is a mismatch between the type inferred from a term and the type expected from the usage, the elaboration procedure will produce a cast.

Observations allow indexing into terms to pinpoint errors. For instance, if we want to highlight the C sub expression in $(x : A) \to (y : (x : B) \to \underline{C}) \to D$ we can

$a ::_{A,\ell,o} B$	written	$a ::_{A,\ell} B$	when	the observation is not relevant
$a ::_{A,\ell} B$	written	$a ::_{A} B$	when	the location is not relevant
$a ::_{A} B$	written	$a :: B$	when	the type of a is clear
$o.Bod_a$	written	$o.Bod$	when	observing a non dependent function type
.	written		when	. could be inferred

Figure 3·2: Cast Language Abbreviations

use the observation $Bod_x.Arg.Bod_y$. In general, the C may specifically depend on x and y so they are tracked as part of the observation. For instance, given the type $(X : \star) \to X$ we might want to point out A when $X = A \to B$ resulting in the type $(X : \star) \to (\underline{A} \to B)$. The observation would then read $Bod_{A \to B}.Arg$, recording the specific type argument that produces a body that can be inspected.

Locations and observations will be used to form blame and produce the runtime error message users will see if their assumptions are wrong.

In addition to the abbreviations from Chapter 2, some new abbreviations for the cast language are listed in Figure 3·2.

3.1.1 How Should Casts Reduce?

Unlike the annotations in Chapter 2, casts cannot simply be erased. How does the cast construct interact with the existing constructs? Casts should not block reduction when there is no problem. Casts should also not prevent terms from checking in the cast system. There are three combinations of syntax that could cause a term to be stuck in reduction or block checking in the cast system:

$\star :: B$	universe under cast	will it "type check" as a type?
$((x : A) \to B) :: C$	function type under cast	will it "type check" as a type?
$(b :: C)\, a$	application to a cast	will it block reduction?

When possible, obvious casts should reduce away, freeing up the underlying term for further reduction and checking. Figure 3·3 shows approximately how these

reductions should be carried out. The most interesting case is when a cast confirms that the applied term is a function, but with potentially different input and output types. Then we use the function type syntax to determine a reasonable cast over the argument, and maintain the appropriate cast over the resulting computation. This operation is similar to the way higher order contracts invert the polarity of blame for the arguments of higher order functions [FF02] and also found in gradual type systems, such as [WF09].

Sometimes casts are correct in blocking reductions, such as when a cast asserts an impossible equality. When a term reaches this state a separate blame judgment will extract the runtime error from the term. Type universes live in the type universe, so any cast that contradicts this should be blamed. Similarly for function types. Terms that take input must be functions, so any cast that contradicts this should blame the source location.

Note that the system outlined here leaves open many possible strategies of reduction and blame. One of the subtle innovations of this system is to completely separate blame from reduction. This sidesteps many of the complexities of having a reduction relevant `abort` term in a dependent type theory [SCA+12, PT18]. As far as reduction is concerned, bad terms simply "get stuck" as it might on a variable from a nonempty typing context. Blame will extract errors from stuck terms in the typing context, but can also be much more aggressive.

This Chapter will outline the minimum requirements for cast reductions that support cast soundness. But one can imagine more sophisticated ways to extract blame from terms or more optimistic reductions. Some some particularly tempting reductions are

$$a ::_C C \quad \rightsquigarrow \quad a$$

$$a ::_{C'} C \quad \rightsquigarrow \quad a \quad \text{when} \quad C' \equiv C$$

However these ignore the possibility that a source of blame may be hiding within

$$
\begin{array}{lcll}
\star :: \star & \rightsquigarrow & \star & \\
\star :: B & \rightsquigarrow & \star :: B' & \text{when} \quad B \rightsquigarrow B' \\
\star ::_{\ell,o} B & \textbf{Blame} & \ell, o & \text{when} \quad B \text{ cannot be } \star \\
((x : A) \to B) :: \star & \rightsquigarrow & (x : A) \to B & \\
((x : A) \to B) :: C & \rightsquigarrow & ((x : A) \to B) :: C' & \text{when} \quad C \rightsquigarrow C' \\
((x : A) \to B) ::_{\ell,o} C & \textbf{Blame} & \ell, o & \text{when} \quad C \text{ cannot be } \star \\
\big(b ::_{(x:A') \to B'} (x : A) \to B\big)\, a & \rightsquigarrow & (b\ (a ::_A A')) ::_{B'[x:=a::_A A']} B\,[x := A] & \\
(b :: C)\, a & \rightsquigarrow & (b :: C')\, a & \text{when} \quad C \rightsquigarrow C' \\
(b ::_{\ell,o} C)\, a & \textbf{Blame} & \ell, o & \text{when} \quad C \text{ cannot be } (x : A) \to B
\end{array}
$$

Figure 3·3: Approximate Cast Language Reductions and Blame

the cast syntax (C, C'). In Chapter 6 we will separate the syntax casts from the equality assertions they can contain, making reductions like the above reasonable. But for this Chapter the theory will be easier with a single syntax form the both casts and asserts. Despite this we will use these reductions in some examples for a cleaner presentation with the notation $\rightsquigarrow_=$ when we erase an exact cast.

3.2 Examples

We can re-examine some of the example terms from Chapter 2, but this time using casts that contain non standard equality assumptions.

3.2.1 Higher Order Functions

Higher order functions are dealt with by distributing casts around applications. If a cast of function type is applied, the argument and body casts are separated and the arguments are swapped. For instance:

$$\begin{aligned}
&((\lambda x \Rightarrow x \,\&\, x) ::_{\mathbb{B}_c \to \mathbb{B}_c, \ell} \mathbb{N}_c \to \mathbb{N}_c)\, 7_c \\
\rightsquigarrow &((\lambda x \Rightarrow x \,\&\, x)\, (7 ::_{\mathbb{N}_c, \ell, Arg} \mathbb{B}_c)) ::_{\mathbb{B}_c, \ell, Bod_{7_c}} \mathbb{N}_c \\
\rightsquigarrow &((7_c ::_{\mathbb{N}_c, \ell, Arg} \mathbb{B}_c) \;\&\; (7_c ::_{\mathbb{N}_c, \ell, Arg} \mathbb{B}_c)) ::_{\mathbb{B}_c, \ell, Bod_{7_c}} \mathbb{N}_c
\end{aligned}$$

If evaluation gets stuck on & we can blame the argument of the cast for equating $\mathbb{N}_c$ and $\mathbb{B}_c$. The *Body* observation records the argument the function is called with. For instance, in the Bod_{7_c} observation. In a dependently typed function the exact argument may be important to give a good error.

3.2.2 Type Universes

Because casts can be embedded inside of casts, types themselves need to normalize and casts need to simplify. Since our system has one universe of types, type casts

only need to simplify themselves when a term of type $\star$ is cast to $\star$. For instance:

$$\begin{aligned} &\left((\lambda x \Rightarrow x) ::_{(\mathbb{B}_c \to \mathbb{B}_c) ::_{\star,\ell,Arg\star,\ell'} \mathbb{N}_c \to \mathbb{N}_c}\right) 7_c \\ \rightsquigarrow &\left((\lambda x \Rightarrow x) ::_{\mathbb{B}_c \to \mathbb{B}_c,\ell'} \mathbb{N}_c \to \mathbb{N}_c\right) 7_c \end{aligned}$$

3.2.3 Pretending $true = false$

Recall that we proved $\neg true_c \doteq_{\mathbb{B}_c} false_c$ in Chapter 2. What happens if it is assumed anyway? Every type equality assumption needs an underlying term, here we can choose $refl_{true_c:\mathbb{B}_c} : true_c \doteq_{\mathbb{B}_c} true_c$, and cast that term to $true_c \doteq_{\mathbb{B}_c} false_c$ resulting in $refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}_c} true_c} true_c \doteq_{\mathbb{B}_c} false_c$. Recall that $\neg true_c \doteq_{\mathbb{B}_c} false_c$ is a shorthand for $true_c \doteq_{\mathbb{B}_c} false_c \to \bot_c$. What if we try to use our term of type $true_c \doteq_{\mathbb{B}_c} false_c$ to get a term of type $\bot_c$?

There is enough static information to generate a warning like:

$(C : (\mathbb{B}_c \to \star)) \to C\, true_c \to \underline{C\, true_c} \stackrel{?}{=} (C : (\mathbb{B}_c \to \star)) \to C\, true_c \to \underline{C\, false_c}$

To let the programmer know they are not doing something safe. But the program can still be run, the reductions are presented in Figure 3·4.

The term reduces to $tt_c ::_{Unit_c} \bot_c$, but has not yet "gotten stuck". Applying the term to any input will uncover the error, so we can inspect the term with $\star$. These reductions are listed in Figure 3·5.

If we explicitly tracked the location and observation information an error message could be generated:

$(C : (\mathbb{B}_c \to \star)) \to C\, true_c \to \underline{C\, true_c} \neq (C : (\mathbb{B}_c \to \star)) \to C\, true_c \to \underline{C\, false_c}$

when

$C \coloneqq \lambda b \Rightarrow b \star \star \bot_c$

$C\, true_c = \bot_c \neq \star = C\, false_c$

Reminding the programmer not to confuse true with false.

	$(\lambda pr \Rightarrow pr\, toLogic\ tt_c)\left(refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}_c} true_c} true_c \doteq_{\mathbb{B}_c} false_c\right)$
$\rightsquigarrow$	$\left(refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}_c} true_c} true_c \doteq_{\mathbb{B}_c} false_c\right)\ toLogic\ tt_c$
$true_c \doteq_{\mathbb{B}_c} false_c$ expands to $(C : (\mathbb{B}_c \to \star)) \to C\, true_c \to C\, false_c$	$\left(refl_{true_c:\mathbb{B}_c} ::_{true_c \doteq_{\mathbb{B}_c} true_c} (C : (\mathbb{B}_c \to \star)) \to C\, true_c \to C\, false_c\right)\ toLogic\ tt_c$
$true_c \doteq_{\mathbb{B}_c} true_c$ expands to $(C : (\mathbb{B}_c \to \star)) \to C\, true_c \to C\, true_c$	$\left(refl_{true_c:\mathbb{B}_c} ::_{(C:(\mathbb{B}_c \to \star)) \to C\, true_c \to C\, true_c} (C : (\mathbb{B}_c \to \star)) \to C\, true_c \to C\, false_c\right)$ $toLogic\ tt_c$
$\rightsquigarrow\rightsquigarrow_=$	$((refl_{true_c:\mathbb{B}_c} toLogic)$ $::_{toLogic\ true_c \to toLogic\ true_c} toLogic\, true_c \to toLogic\ false_c)\ tt_c$
$\rightsquigarrow\rightsquigarrow_=$	$(refl_{true_c:\mathbb{B}_c}\ toLogic\ tt_c) ::_{toLogic\ true_c} toLogic\ false_c$
$refl_{true_c:\mathbb{B}_c}$ expands to $\lambda -\ cx \Rightarrow cx$	$((\lambda -\ cx \Rightarrow cx)\ \ toLogic\ tt_c) ::_{toLogic\ true_c} toLogic\ false_c$
$\rightsquigarrow\rightsquigarrow$	$tt_c ::_{toLogic\ true_c} toLogic\ false_c$
$toLogic \coloneqq \lambda b \Rightarrow b \star Unit_c\ \bot_c$	$tt_c ::_{(\lambda b \Rightarrow b \star Unit_c\ \bot_c)\ true_c} (\lambda b \Rightarrow b \star Unit_c\ \bot_c)\ \ false_c$
$\rightsquigarrow\rightsquigarrow$	$tt_c ::_{true_c \star Unit_c\ \bot_c} false_c \star Unit_c\ \bot_c$
$true_c \coloneqq \lambda -\ x - \Rightarrow x$	$tt_c ::_{(\lambda -\ x\ - \Rightarrow x) \star Unit_c\ \bot_c} false_c \star Unit_c\ \bot_c$
$\rightsquigarrow\rightsquigarrow\rightsquigarrow$	$tt_c ::_{Unit_c} false_c \star Unit_c\ \bot_c$
$\rightsquigarrow\rightsquigarrow\rightsquigarrow$	$tt_c ::_{Unit_c} false_c \star Unit_c\ \bot_c$
$false_c \coloneqq \lambda -\ -\ y \Rightarrow y$	$tt_c ::_{Unit_c} (\lambda -\ -\ y \Rightarrow y) \star Unit_c\ \bot_c$
$\rightsquigarrow\rightsquigarrow\rightsquigarrow$	$tt_c ::_{Unit_c} \bot_c$

Multiple reductions are sometimes grouped together with multiple $\rightsquigarrow\rightsquigarrow$ ..., vacuous casts are ignored with $\rightsquigarrow_=$.

Figure 3·4: true=false

	$(tt_c ::_{Unit_c} \bot_c)\star$
$\bot_c := (X:\star) \to X$	$(tt_c ::_{Unit_c} (X:\star) \to X)\star$
$Unit_c := (X:\star) \to X \to X$	$\big(tt_c ::_{(X:\star)\to X\to X} ((X:\star) \to X)\big)\star$
$\rightsquigarrow_=$	$(tt_c\star) ::_{\star\to\star} \star$
Blame!	$(tt_c\star) ::_{\star \underline{\to} \star} \underline{\star}$

Figure 3·5: true=false cont.

$$\frac{}{\star\ \mathbf{Val}}\ \text{Val-}\star$$

$$\frac{}{(x:A)\to B\ \mathbf{Val}}\ \text{Val-fun-ty}$$

$$\frac{}{\mathsf{fun}\ f\ x \Rightarrow b\ \mathbf{Val}}\ \text{Val-fun}$$

$$\frac{a\ \mathbf{Val} \quad A\ \mathbf{Val} \quad B\ \mathbf{Val} \quad a \not= \star \quad a \not= (x:C)\to C'}{a ::_{A,\ell,o} B\ \mathbf{Val}}\ \text{Val-::}$$

Figure 3·6: Cast Language Values

3.3 Cast Language Evaluation and Blame

As in Chapter 2 we can equip the surface language with a call-by-value reduction system.

Unlike the surface language, it is no longer practical to characterize values syntactically. Values are specified by judgments in Figure 3·6. They are standard except for the Val-::, which states that a type ($\star$ or function type) under a cast is not a value.

For example, $(\lambda x \Rightarrow a) :: \star$ is a value while $\star :: \star$ is not a value. Values are characterized this way to match reduction, since $\star :: \star \rightsquigarrow \star$. If the underling term of a cast is a type, such as $A ::_B B'$ then the cast will eventually reduce to $A ::_B B' \rightsquigarrow_* A ::_\star \star \rightsquigarrow A$ if there is no blame. If there is blame then the reduction

$$\frac{a\,\mathbf{Val}}{(\mathsf{fun}\,f\,x \Rightarrow b)\,a \rightsquigarrow b\,[f \coloneqq \mathsf{fun}\,f\,x \Rightarrow b, x \coloneqq a]} \rightsquigarrow \text{-}\mathsf{fun}\text{-app-red}$$

$$\frac{b\,\mathbf{Val} \quad a\,\mathbf{Val}}{\begin{array}{l}\big(b ::_{(x:A_1)\to B_1,\ell,o} (x : A_2) \to B_2\big)\,a \rightsquigarrow \\ (b\ (a ::_{A_2,\ell,o.Arg} A_1)) ::_{B_1\left[x:=a::_{A_2,\ell,o.Arg}A_1\right],\ell,o.Bod_a} B_2\,[x \coloneqq a]\end{array}} \rightsquigarrow \text{-::-app-red}$$

$$\frac{A\,\mathbf{Val}}{A ::_{\star,\ell,o} \star \rightsquigarrow A} \rightsquigarrow \text{-::-}\star\text{-red}$$

$$\frac{a \rightsquigarrow a'}{a ::_{A,\ell,o} B \rightsquigarrow a' ::_{A,\ell,o} B} \rightsquigarrow \text{-::-1}$$

$$\frac{a\,\mathbf{Val} \quad A \rightsquigarrow A'}{a ::_{A,\ell,o} B \rightsquigarrow a ::_{A',\ell,o} B} \rightsquigarrow \text{-::-2}$$

$$\frac{a\,\mathbf{Val} \quad A\,\mathbf{Val} \quad B \rightsquigarrow B'}{a ::_{A,\ell,o} B \rightsquigarrow a ::_{A,\ell,o} B'} \rightsquigarrow \text{-::-3}$$

$$\frac{b \rightsquigarrow b'}{b\,a \rightsquigarrow b'\,a} \rightsquigarrow \text{-app-1}$$

$$\frac{b\,\mathbf{Val} \quad a \rightsquigarrow a'}{b\,a \rightsquigarrow b\,a'} \rightsquigarrow \text{-app-2}$$

Figure 3·7: Cast Language Call-by-Value Reductions

will get stuck around that cast.

Call-by-value reductions are listed in Figure 3·7. They are standard for call-by-value except that casts can distribute over application in $\rightsquigarrow$ -::-app-red, and casts can reduce when both types are $\star$ in $\rightsquigarrow$ -::-$\star$-red.

As hinted at in the examples, the $\rightsquigarrow$ -::-app-red rule allows an argument to be pushed under a cast between two function types. For instance, if $\big(b ::_{(x:A_1)\to B_1} (x : A_2) \to B_2\big)\,a$ is well cast then $b : (x : A_1) \to B_1$ and $a : A_2$. We cannot move a directly under the cast since it may not have the correct type. However, the cast asserts $(x : A_1) \to B_1 = (x : A_2) \to B_2$, so we can assume $A_1 = A_2$ and use it to construct a new cast for $a ::_{A_2} A_1$. Similarly we can perform the substitution $B_1[x \coloneqq a ::_{A_2} A_1]$, since B_1 is expecting a type of A_1, while the substitution $A_2[x \coloneqq a]$

can be performed directly. Finally, the location and observation data is accounted for.

The $\rightsquigarrow$ -::-$\star$-red is the only way to remove a cast in this reduction system. This rule is sufficient to keep reductions from getting stuck when there is no blame. Since types without blame will have their casts reduced, leaving $\star$ or $(x : A) \to B$. Which is exactly what is needed so that $\rightsquigarrow$ -::-app-red and $\rightsquigarrow$ -::-$\star$-red are not blocked.

The definition of **Stuck** from Chapter 2 applies equally well to the cast language: m **Stuck** if m is not a value and there does not exist m' such that $m \rightsquigarrow m'$.

In addition to reductions and values we also specify blame judgments in Figure 3·8. Blame tracks the information needed to create a good error message and is inspired by the many systems that use blame tracking [FF02, WF09, Wad15]. Specifically the judgment $a\,\mathbf{Blame}_{\ell,\,o}$ means that a witnesses a contradiction in the source code at location ℓ under the observations o. With only dependent functions and universes, only inequalities of the form $* \not\equiv A \to B$ can be witnessed. The first two rules of the blame judgment witness these concrete type inequalities. The rest of the blame rules recursively extract concrete witnesses from larger terms. Limiting the observations to the form $* \not\equiv A \to B$ which makes the system in this Chapter simpler than the system in Chapter 5 where more observations are possible because of the addition of data.

3.4 Cast System

In a programming language, type soundness proves some undesirable behaviors are unreachable from a well typed term. How should this apply to the cast language, where bad behaviors are intended to be reachable? The cast language allows the program to be stuck in a bad state, but requires that when that state is reached we have a good explanation to give the programmer that can blame the original faulty

$$\frac{}{\left(a ::_{(x:A)\to B,\ell,o} \star\right)\ \mathbf{Blame}_{\ell,\,o}}$$

$$\frac{}{(a ::_{\star,\ell,o} (x : A) \to B)\ \mathbf{Blame}_{\ell,\,o}}$$

$$\frac{a\,\mathbf{Blame}_{\ell,\,o}}{(a ::_{A,\ell',o'} B)\ \mathbf{Blame}_{\ell,\,o}}$$

$$\frac{a\,\mathbf{Blame}_{\ell,\,o}}{(a ::_{A,\ell',o'} B)\ \mathbf{Blame}_{\ell,\,o}}$$

$$\frac{B\,\mathbf{Blame}_{\ell,\,o}}{(a ::_{A,\ell',o'} B)\ \mathbf{Blame}_{\ell,\,o}}$$

$$\frac{b\,\mathbf{Blame}_{\ell,\,o}}{(b\,a)\ \mathbf{Blame}_{\ell,\,o}}$$

$$\frac{a\,\mathbf{Blame}_{\ell,\,o}}{(b\,a)\ \mathbf{Blame}_{\ell,\,o}}$$

Figure 3·8: Cast Language Blame

type assumption in their source code. Where the slogan for type soundness is "well typed terms don't get stuck", the slogan for cast soundness is "well cast terms don't get stuck without blame". Formally, if $\vdash a : A$ and $a \rightsquigarrow_* a'$ and $a'\,\mathbf{Stuck}$ then $a'\,\mathbf{Blame}_{\ell,\,o}$ for some ℓ and o. This will be called **cast soundness**.

In Chapter 2 we proved type soundness for a minimal dependently typed language with a progress and preservation style proof given a suitable definition of term equivalence. We can extend that proof to support cast soundness with only a few modifications.

The cast language supports its own type assignment system, defined in Figure 3·9. This system ensures that computations will not get stuck without enough information for good runtime error messages. Specifically computations will not get stuck without a source location and a witness of inequality. The only rule that works differently than the surface language is the cast-: : rule that allows runtime type assertions.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \ : \ A}\ \text{cast-var}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A : \star \quad \Gamma \vdash B : \star}{\Gamma \vdash a ::_{A,\ell,o} B \ : \ B}\ \text{cast-:::}$$

$$\frac{}{\Gamma \vdash \star : \star}\ \text{cast-}\star$$

$$\frac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star}{\Gamma \vdash (x : A) \to B \ : \ \star}\ \text{cast-}\mathsf{fun}\text{-ty}$$

$$\frac{\Gamma, f : (x : A) \to B, x : A \vdash b : B}{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow b \ : \ (x : A) \to B}\ \text{cast-}\mathsf{fun}$$

$$\frac{\Gamma \vdash b : (x : A) \to B \quad \Gamma \vdash a : A}{\Gamma \vdash b\, a \ : \ B\,[x \coloneqq a]}\ \text{cast-fun-app}$$

$$\frac{\Gamma \vdash a : A \quad A \equiv A'}{\Gamma \vdash a : A'}\ \text{cast-conv}$$

Figure 3·9: Cast Language Type Assignment Rules

As before we need a suitable reduction relation to generate our equivalence relation. Figure 3·10 shows that system of reductions. The full rule for function reduction is given in $\Rrightarrow$ -fun-::-red which makes the behavior from the examples explicit: argument types are swapped as a term is applied under a cast. Casts from a type universe to a type universe are allowed by the $\Rrightarrow$ -::-red rule. Since observations embed expressions, they must also be given parallel reductions.

3.4.1 Definitional Equality

As in Chapter 2, we will define a suitable notion of definitional equality to derive the other properties of the system. While it may seem counterintuitive to define a definitional equality in a system that is intended to avoid definitional equality, this is fine since programmers will never interact directly with the cast system. Programmers will only interact with elaboration, and elaboration will only result in well cast terms. The cast system only exists to give theoretical assurances.

As before $\Rrightarrow_*$ can be shown to be confluent, and used to generate the equality relation. The proofs follow the same structure as Chapter 2, but since observations can contain terms, $\Rrightarrow$ and **max** must be extended to observations. Proofs must be extended to mutually induct on observations, since they can contain expressions that could also reduce.

The explicit new rules for **max** are given in Figure 3·11 with the structural rules omitted since they are the same as Chapter 2.

The expected lemas hold.

Lemma 3.1. *Triangle Properties of* $\Rrightarrow$.
If $a \Rrightarrow a'$ *then* $a' \Rrightarrow \boldsymbol{max}(a)$.
If $o \Rrightarrow o'$ *then* $o' \Rrightarrow \boldsymbol{max}(o)$.

Proof. By mutual induction on the derivations of $m \Rrightarrow m'$ and $o \Rrightarrow o'$. □

Lemma 3.2. *Diamond Property of* $\Rrightarrow$.
If $a \Rrightarrow a'$ *and* $a \Rrightarrow a''$ *implies* $a' \Rrightarrow \boldsymbol{max}(a)$ *and* $a'' \Rrightarrow \boldsymbol{max}(a)$.

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a'}{(\mathsf{fun}\, f\, x \Rightarrow b)\, a \Rrightarrow b'\,[f \coloneqq \mathsf{fun}\, f\, x \Rightarrow b', x \coloneqq a']} \Rrightarrow \text{-}\mathsf{fun}\text{-app-red}$$

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a' \quad A_1 \Rrightarrow A_1' \quad A_2 \Rrightarrow A_2' \quad B_1 \Rrightarrow B_1' \quad B_2 \Rrightarrow B_2' \quad o \Rrightarrow o'}{\begin{array}{c}\left(b :: _{(x:A_1)\to B_1,\ell,o}\ (x : A_2) \to B_2\right) a \Rrightarrow \\ \left(b'\ \left(a' :: _{A_2',\ell,o.Arg}\ A_1'\right)\right) :: _{B_1'\left[x:=a' :: _{A_2',\ell,o'.Arg} A_1'\right],\ell,o'.Bod_{a'}}\ B_2'\,[x \coloneqq a']\end{array}} \Rrightarrow \text{-fun-}:\text{:-red}$$

$$\frac{a \Rrightarrow a'}{a :: _{\star,\ell,o} \star \Rrightarrow a'} \Rrightarrow \text{-}:\text{:-red}$$

$$\frac{a \Rrightarrow a' \quad A_1 \Rrightarrow A_1' \quad A_2 \Rrightarrow A_2' \quad o \Rrightarrow o'}{a :: _{A_1,\ell,o} A_2 \Rrightarrow a' :: _{A_1',\ell,o'} A_2'} \Rrightarrow \text{-}:\text{:}$$

$$\frac{}{x \Rrightarrow x} \Rrightarrow \text{-var} \qquad \frac{}{\star \Rrightarrow \star} \Rrightarrow \text{-}\star$$

$$\frac{A \Rrightarrow A' \quad B \Rrightarrow B'}{(x : A) \to B \Rrightarrow (x : A') \to B'} \Rrightarrow \text{-}\mathsf{fun}\text{-ty} \qquad \frac{b \Rrightarrow b'}{\mathsf{fun}\, f\, x \Rightarrow b \ \Rrightarrow\ \mathsf{fun}\, f\, x \Rightarrow b'} \Rrightarrow \text{-}\mathsf{fun}$$

$$\frac{b \Rrightarrow b' \quad a \Rrightarrow a'}{b\, a \Rrightarrow b'\, a'} \Rrightarrow \text{-}\mathsf{fun}\text{-app}$$

$$\frac{}{\cdot \Rrightarrow \cdot} \Rrightarrow \text{-obs-emp} \quad \frac{o \Rrightarrow o'}{o.Arg \Rrightarrow o'.Arg} \Rrightarrow \text{-obs-Arg} \qquad \frac{o \Rrightarrow o' \quad a \Rrightarrow a'}{o.Bod_a \Rrightarrow o'.Bod_{a'}} \Rrightarrow \text{-obs-Bod}$$

$$\frac{a \Rrightarrow_* a'' \quad a' \Rrightarrow_* a''}{a \equiv a'} \equiv \text{-def}$$

Figure 3·10: Cast Language Parallel Reductions

$$
\begin{array}{lcll}
\mathbf{max}(& (\mathsf{fun}\, f\, x \Rightarrow b)\; a &) = & \mathbf{max}\,(b)\,[f := \mathsf{fun}\, f\, x \Rightarrow \mathbf{max}\,(b)\,, x := \mathbf{max}\,(a)] \\
 & & & \left(\mathbf{max}\,(b)\;\left(\mathbf{max}\,(a) ::_{\mathbf{max}(A_2),\ell,\mathbf{max}(o).Arg} \mathbf{max}\,(A_1)\right)\right) \\
\mathbf{max}(& \left(b ::_{(x:A_1)\to B_1,\ell,o} (x : A_2) \to B_2\right) a &) = & ::_{\mathbf{max}(B_2)\left[x:=\mathbf{max}(a)::_{\mathbf{max}(A_2),\ell,\mathbf{max}(o).Arg}\mathbf{max}(A_1)\right],\ell,\mathbf{max}(o).Bod_{\mathbf{max}(a)}} \\
 & & & \mathbf{max}\,(B_2)\,[x := \mathbf{max}\,(a)] \\
\mathbf{max}(& b ::_{\star,\ell,o} \star &) = & \mathbf{max}\,(b) \\
 & & & \text{otherwise} \\
\mathbf{max}(& b ::_{B_1,\ell,o} B_2 &) = & \mathbf{max}\,(b) ::_{\mathbf{max}(B_1),\ell,\mathbf{max}(o)} \mathbf{max}\,(B_2) \\
\mathbf{max}(& \ldots &) = & \ldots \\
\mathbf{max}(& . &) = & . \\
\mathbf{max}(& o.Arg &) = & \mathbf{max}\,(o)\,.Arg \\
\mathbf{max}(& o.Bod_a &) = & \mathbf{max}\,(o)\,.Bod_{\mathbf{max}(a)}
\end{array}
$$

Figure 3·11: The **max** Function

Proof. This follows directly from the triangle property. □

Lemma 3.3. *Confluence of* $\Rrightarrow_*$.

Proof. By repeated application of the diamond property. □

$\equiv$ is an equivalence

As before, this allows us to prove $\equiv$ is transitive, and therefore $\equiv$ is an equivalence.

Theorem 3.4. $\equiv$ *is transitive.*
If $a \equiv a'$ *and* $a' \equiv a''$ *implies* $a \equiv a''$.

Proof. Follows from the confluence of $\Rrightarrow_*$. □

Stability

Similar to Chapter 2 we need to prove that equality is stable over type constructors.

Lemma 3.5. *Stability of* $\rightarrow$ *over* $\Rrightarrow_*$.
$\forall A, B, C.\ (x : A) \rightarrow B \Rrightarrow_* C$ *implies* $\exists A', B'.C = (x : A') \rightarrow B' \wedge A \Rrightarrow_* A' \wedge B \Rrightarrow_* B'$.

Proof. By induction on $\Rrightarrow$, which implies the result for $\Rrightarrow_*$. □

Corollary 3.6. *Stability of* $\rightarrow$ *over* $\equiv$.
The following rule is admissible:

$$\frac{(x : A) \rightarrow B \equiv (x : A') \rightarrow B'}{A \equiv A' \quad B \equiv B'}$$

With these properties proving $\equiv$ is suitable as a definitional equivalence, we can now tackle the progress and preservation lemmas.

3.4.2 Preservation

As in Chapter 2, $\Rrightarrow$ preserves types. The argument is similar to that of Chapter 2 though more inversion lemmas are needed.

Structural Properties

We begin by proving the structural properties:

Lemma 3.7. *Context Weakening.*
The following rule is admissible:

$$\frac{\Gamma \vdash a : A}{\Gamma, \Gamma' \vdash a : A}$$

Proof. By induction on the derivations of the cast system. □

Lemma 3.8. *Substitution preserves types.*
The following rule is admissible:

$$\frac{\Gamma \vdash c : C \quad \Gamma, x : C, \Gamma' \vdash a : A}{\Gamma, \Gamma' \left[x \coloneqq c\right] \vdash a \left[x \coloneqq c\right] : A \left[x \coloneqq c\right]}$$

Proof. By induction on the derivations of the cast system. □

As before the notion of definitional equality can be extended to cast contexts.

Lemma 3.9. *Contexts that are equivalent preserve types.*
The following rule is admissible:

$$\frac{\Gamma \vdash n : N \quad \Gamma \equiv \Gamma'}{\Gamma' \vdash n : N}$$

Proof. By induction over the derivations of the cast system. □

As before we show inversions on the term syntaxes, generalizing the induction hypothesis up to equality when needed.

Lemma 3.10. fun*-Inversion (generalized).*

$$\frac{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow a \,:\, C \quad C \equiv (x : A) \to B}{\Gamma, f : (x : A) \to B, x : A \vdash b : B}$$

Proof. By induction on the derivations of the cast system. □

This allows us to conclude the corollary

Corollary 3.11. fun-*Inversion.*

$$\frac{\Gamma \vdash \mathsf{fun}\, f\, x \Rightarrow a \;:\; (x : A) \to B}{\Gamma, f : (x : A) \to B, x : A \vdash b : B}$$

Unlike Chapter 2, we also need an inversion for function types.

Lemma 3.12. $\to$*-Inversion (generalized).*

The following rule is admissible

$$\frac{\Gamma \vdash (x : A) \to B \;:\; C \quad C \equiv \star}{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star}$$

Proof. By induction on the typing derivations □

Which allows the expected corollary:

Corollary 3.13. $\to$*-Inversion.*

$$\frac{\Gamma \vdash (x : A) \to B \;:\; \star}{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star}$$

We also need a lemma that will invert the typing information out of the cast operator. This can be proven directly without generalizing over definitional equality.

Lemma 3.14. ::*-Inversion.*

The following rule is admissible:

$$\frac{\Gamma \vdash a ::_{A,\ell,o} B \;:\; C}{\Gamma \vdash a : A \quad \Gamma \vdash A : \star \quad \Gamma \vdash B : \star}$$

Proof. By induction on the typing derivations:

Case 1. cast-: : follows directly.

Case 2. cast-conv by induction.

Case 3. All other cases impossible!

□

Note that the derivations of the conclusion of this theorem can always be made smaller than the derivation from the premise. This allows other proofs to use induction on the output of this lemma while still being well founded.

Theorem 3.15. $\Rrightarrow$ *Preserves types.*

The following rule is admissible:

$$\frac{a \Rrightarrow a' \quad \Gamma \vdash a : A}{\Gamma \vdash a' : A}$$

Proof. By induction on the cast derivation $\Gamma \vdash m : M$, specializing on $m \Rrightarrow m'$:

Case 1. If the term typed with cast-$:::$

Case i. If the term reduced with $\Rrightarrow$ -$:::$-red then preservation follows by induction.

Case ii. If the term reduced with $\Rrightarrow$ -$:::$ then preservation follows by induction and conversion.

Case 2. If the term typed with cast-fun-app:

Case i. If the term reduced with $\Rrightarrow$ -fun-$:::$-red then we have $\Gamma \vdash \left(b ::_{(x:A_1)\to B_1,\ell,o} (x : A_2) \to B_2\right) : (x : A_2) \to B_2$, $\Gamma \vdash a : A_2$, $b \Rrightarrow b'$, $a \Rrightarrow a'$, $A_1 \Rrightarrow A_1'$, $A_2 \Rrightarrow A_2'$, $B_1 \Rrightarrow B_1'$, $B_2 \Rrightarrow B_2'$, and $o \Rrightarrow o'$. We must show $\Gamma \vdash (b'\ ac) ::_{B_1'[x:=ac],\ell,o'.Bod_{a'}} B_2'\,[x := a']$, where $ac = a' ::_{A_2',\ell,o'.Arg} A_1'$.

With cast-inversion we can show $\Gamma \vdash b : (x : A_1) \to B_1$, $\Gamma \vdash (x : A_1) \to B_1 : \star$, $\Gamma \vdash (x : A_2) \to B_2 : \star$. Since these derivations are structurally smaller, we can use induction on them.

$\Gamma \vdash a' : A_2'$	cast-conv
$(x : A_2) \to B_2 \Rrightarrow (x : A_2') \to B_2'$	by $\Rrightarrow$ -fun-ty
$\Gamma \vdash (x : A_2') \to B_2' : \star$	by induction with $\Gamma \vdash (x : A_2) \to B_2 : \star$
$\Gamma \vdash A_2' : \star, \Gamma, x : A_2' \vdash B_2' : \star$	fun-ty-inversion
$(x : A_1) \to B_1 \Rrightarrow (x : A_1') \to B_1'$	$\Rrightarrow$ -fun-ty
$\Gamma \vdash (x : A_1') \to B_1' : \star$	by induction with $\Gamma \vdash (x : A_1) \to B_1 : \star$
$\Gamma \vdash A_1' : \star, \Gamma, x : A_1' \vdash B_1' : \star$	fun-ty-inversion
$\Gamma \vdash ac : A_1'$	by cast-: :
$\Gamma \vdash b' : (x : A_1) \to B_1$	by induction with $\Gamma \vdash b : (x : A_1) \to B_1$
$\Gamma \vdash b' : (x : A_1') \to B_1'$	by cast-conv
$\Gamma \vdash b'\ ac : B_1'\,[x \coloneqq ac]$	by cast-fun-app
$\Gamma \vdash B_1'\,[x \coloneqq ac] : \star$	by subst. preservation
$\Gamma \vdash B_2'\,[x \coloneqq a'] : \star$	by subst. preservation

Which allows us to conclude $\Gamma \vdash (b'\ ac) :: _{B_1'[x:=ac],\ell,o'.Bod_{a'}} B_2'\,[x \coloneqq a']$ by cast-: :.

Case ii. All other reductions are similar to Chapter 2.

Case 3. All other cases follow along the lines of Chapter 2.

□

Progress

Preservation alone isn't sufficient for a cast sound language. We also need to show that there is an evaluation that behaves appropriately in an empty typing context. Again this will broadly follow the outline of the surface language proof in Chapter 2, with a few substantial changes.

As before we have that $\rightsquigarrow$ preserves types.

Fact 3.16. *$\rightsquigarrow$ preserves types.*

Since the following rule is admissible:

$$\frac{m \rightsquigarrow m'}{m \Rrightarrow m'}$$

As in Chapter 2 we will need technical lemmas that determine the syntax of a value of a given type in the empty context. However, canonical function values look different because they must account for the possibility of blame arising from a stuck term.

Lemma 3.17. *$\star$-Canonical forms (generalized).*
*If $\vdash a : A$, a **Val**, and $A \equiv \star$ then either.*
$a = \star$,
or there exists C, B, such that $a = (x : C) \to B$.

Proof. By induction on the cast derivation:

Case 1. cast-$\star$ and cast-fun-ty follow directly.

Case 2. ty-conv follows by induction and that $\equiv$ is an equivalence.

Case 3. cast-fun is impossible since $(x : A) \to B \not\equiv \star$!

Case 4. cast-: : is impossible! Inductively the underlying term must be $\star$, or $(x : C) \to B$. Which contradicts the side conditions of Val-: :.

Case 5. Other rules are impossible, since they do not type values in an empty context!

□

Leading to the corollary:

Corollary 3.18. *$\star$-Canonical forms.*
*If $\vdash A : \star$, and A **Val** then either*
$A = \star$,
or there exists C, B, such that $A = (x : C) \to B$.

Likewise:

Lemma 3.19. *$\to$-Canonical forms (generalized).*
If $\vdash a : A$, a ***Val***, *and* $A \equiv (x : C) \to B$ *then either*
$a = \mathsf{fun}\, f\, x \Rightarrow b$
or $a = d ::_{D,\ell,o} (x : C') \to B'$, d ***Val***, D ***Val***, $C' \equiv C$, $B' \equiv B$

Proof. By induction on the cast derivation:

Case 1. cast-fun follows directly.

Case 2. cast-:: then it must be a value from Val-:: satisfying the 2nd conclusion.

Case 3. ty-conv follows by induction and the transitivity of $\equiv$.

Case 4. cast-$\star$ and cast-fun-ty are impossible by the stability of $\equiv$!

Case 5. Other rules are impossible, since they do not type values in an empty context!

□

As a corollary:

Corollary 3.20. *$\to$-Canonical forms.*
If $\vdash a : (x : C) \to B$, *and* a ***Val***
$a = \mathsf{fun}\, f\, x \Rightarrow b$
or $a = d ::_{D,\ell,o} (x : C') \to B'$, d ***Val***, D ***Val***, $C' \equiv C$, $B' \equiv B$.

This further means if $\vdash a : (x : C) \to B$, and a **Val** then a is not a type, $a \not= \star$, $a \not= (x : C) \to C'$.

We can now prove the progress lemma.

Theorem 3.21. *Progress.*
If $\vdash a : A$ *then either*
a ***Val***,
there exists a' *such that* $a \rightsquigarrow a'$,
or there exists ℓ, o *such that* a $\boldsymbol{Blame}_{\ell,o}$.

Proof. As usual this follows form induction on the typing derivation:

Case 1. cast-$\star$, cast-fun-ty, and cast-fun follow directly.

Case 2. cast-var, impossible in an empty context!

Case 3. cast-conv, by induction.

Case 4. cast-::, then the term is $a ::_{A,\ell,o} B : B$. Each of a, A, and B can be blamed, can step, or is a value (By induction):

- *Case* i. If any of a, A, or B, step then the entire term can step.
- *Case* ii. If any of a, A, or B, can be blamed then the entire term can be blamed.
- *Case* iii. If all of a, A, and B, are values then A and B are types of canonical form.
  - *Case a.* If both A, and B, is $\star$ then step.
  - *Case b.* If one of A, and B, is $\star$ and the other is $(x : C_A) \to D_A$ then blame.
  - *Case c.* Otherwise the term is a value by the canonical forms.

Case 5. cast-fun-app, then the term is $(b\, a)$. Each of a, and b can be blamed, can step, or is a value (By induction):

- *Case* i. If any of b, or a, step then the entire term can step
- *Case* ii. If any of b, or a, can be blamed then the entire term can be blamed
- *Case* iii. If b **Val**, a **Val** by canonical forms:
  - *Case a.* $b = \mathsf{fun}\, f\, x \Rightarrow c$ the term steps.
  - *Case b.* $b = d_b ::_{D_b,\ell_b\, o_b} (x : A') \to B'$ we have by canonical forms and induction that:
    - *Case* 1. $D_b = \star$ and blame can be generated.
    - *Case* 2. Otherwise $D_b = (x : A_{D_b}) \to B_{D_b}$ and a step is possible.

□

3.4.3 Cast Soundness

Cast soundness follows from progress and preservation as expected.

Theorem 3.22. *Cast soundness. If* $\vdash a : A$ *and* $a \rightsquigarrow_* a'$ *and* a' ***Stuck*** *then* a' $\boldsymbol{Blame}_{\ell,o}$ *for some* ℓ *and* o.

Proof. This follows by iterating the progress and preservation lemmas. □

3.4.4 Discussion

Because of the conversion rule and non-termination, checking in the cast system is undecidable. This is fine since the cast system only exists to ensure theoretical properties. Programmers will only use the system through the elaboration procedure described in the next section. Every term produced by elaboration will cast check, and the elaboration is decidable.

As in the surface language TAS, the cast language is logically unsound by design.

Just as there are many different flavors of definitional equality that could have been used in Chapter 2, there are also many possible degrees to which runtime equality can be enforced. The **Blame** relation in Figure 3·8 outlines a minimal checking strategy that supports cast soundness. For instance[1], `head Bool 1 (rep Bool True 0)` will result in blame since 1 and 0 have different head constructors. But `head Bool 1 (rep Bool True 9)` will not result in blame since 1 and 9 have the same head constructor and the computation can reduce to `True`.

It is likely that more aggressive checking is preferable in practice, especially in the presence of data types. That is why our implementation checks equalities up to binders. This corresponds better to the call-by-value behavior of the implemented interpreter. For this reason we call this strategy **check-by-value**.

This behavior is consistent with the conjectured partial correctness of logically unsound call-by-value execution for dependent types in [JZSW10].

Unlike static type-checking, these runtime checks have runtime costs. Since the language allows nontermination, checks can take forever to resolve at runtime.

[1] Assuming the data types of Chapter 5.

We don't expect this to be a large issue in practice, at least any more than is usual in mainstream languages that allow many other sources of non termination. The implementation optimizes away casts when it knows that blame is impossible. Additionally, we could limit the number of steps allowed in cast normalization and blame slow code.

3.5 Elaboration

Even though the cast language allows us to optimistically assert equalities, manually noting every cast would be cumbersome. This bureaucracy is solved with an elaboration procedure that translates (untyped) terms from the surface language into the cast language. If the term is well typed in the surface language, elaboration will produce a term without blamable errors. Terms with unproven equality in types are mapped to a cast with enough information to point out the original source when an inequality is witnessed.

Elaboration serves a similar role as the bidirectional type system did in Chapter 2, and uses a similar methodology. Instead of performing a static equality check when the inference mode and the check mode meet, a runtime cast is inserted asserting the types are equal.

In order to perform elaboration, the surface language needs to be enriched with location information, ℓ, at every position that could result in a type mismatch. This is done in Figure 3·12. Note that the location tags correspond with the check annotations of the bidirectional system. For technical reasons the set of locations is nonempty, and a specific null location (.) is designated. That null location can be used when we need to generate fresh terms, but have no sensible location information available. All the meta theory from Chapter 2 goes through assuming that all

source labels,			
ℓ	$::=$	...	
	$\mid$	.	no source label
expressions,			
m, n, M, N	$::=$	x	variable
	$\mid$	$m ::_\ell M^{\ell'}$	annotation
	$\mid$	$\star$	type universe
	$\mid$	$(x : M_\ell) \to N_{\ell'}$	function type
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow m$	function
	$\mid$	$m_\ell\, n$	application

Figure 3·12: Surface Language Syntax with Locations

$m ::_\ell M^{\ell'}$	written	$m ::_\ell M$	when	ℓ' is irrelevant
$m ::_\ell M$	written	$m :: M$	when	ℓ is irrelevant
$(x : M_\ell) \to N_{\ell'}$	written	$(x : M) \to N$	when	ℓ, ℓ' are irrelevant
$m_\ell\, n$	written	$m\, n$	when	ℓ is irrelevant

Figure 3·13: Surface Language Abbreviations

locations are indistinguishable and by generating null locations when needed[2]. We will avoid writing these annotations when they are unneeded (explicitly in Figure 3·13).

3.5.1 Examples

Functions will elaborate the expected types to their arguments when they are applied.

Example 3.23. Assuming $f : \mathbb{B}_c \to \mathbb{B}_c$ then $f_\ell 7_c : \mathbb{B}_c$ elaborates to $f\,(7_c ::_{\mathbb{N}_c,\ell,.Arg} \mathbb{B}_c) : \mathbb{B}_c$.

As with bidirectional type checking, variable types will be inferred from the typing environment.

Example 3.24. $(\lambda x \Rightarrow 7_c) ::_\ell \mathbb{B}_c \to \mathbb{B}_c$ elaborates to $(\lambda x \Rightarrow 7 ::_{\mathbb{N}_c,\ell,Bod_x} \mathbb{B}_c)$.

[2]For instance, the parallel reduction relation will associate all locations, $\frac{M \Rightarrow M' \quad N \Rightarrow N'}{(x:M_l)\to N_{l'} \Rightarrow (x:M'_{l''})\to N'_{l'''}} \Rightarrow$-fun-ty, so that the relation does not discriminate over syntaxes that come from different locations. While the **max** function will map terms into the null location, $\mathbf{max}\,((x : M_\ell) \to N_{\ell'}) = \mathbf{max}\,((x : \mathbf{max}\,(M)_\cdot) \to \mathbf{max}\,(N)_\cdot)$.

To keep the theory simple, we allow vacuous casts to be created,

Example 3.25. Assuming $f : \mathbb{N}_c \to \mathbb{B}_c \to \mathbb{B}_c$ then $f_\ell 7_{c\ell'} 3_c \ : \mathbb{B}_c$ elaborates to $f\,(7_c ::_{\mathbb{N}_c,\ell,Arg} \mathbb{N}_c)\,(3_c ::_{\mathbb{N}_c,\ell',Arg} \mathbb{B}_c) \ : \mathbb{B}_c$.

Unlike in gradual typing, we cannot elaborate arbitrary untyped syntax. The underlying type of a cast needs to be known so that a function type can swap its argument type at application. For instance, $\lambda x \Rightarrow x$ will not elaborate since the intended type is not known. Fortunately, our experimental testing suggests that a majority of randomly generated terms can be elaborated, compared to the surface language where only a small minority of terms would type check. The programmer can make any term elaborate if they annotate the intended type. For instance, $(\lambda x \Rightarrow x) :: * \to *$ will elaborate.

3.5.2 Elaboration Procedure

Like the bidirectional rules, the rules for elaboration are broken into two judgments:

- $H \vdash m \overleftarrow{:_{\ell,o}} A \,\mathbf{Elab}\, a$, that generates a cast term a from a surface term m given its expected type A along with a location ℓ and observation o that made that assertion.
- $H \vdash m \,\mathbf{Elab}\, a \overrightarrow{:} A$, that generates a cast term a and its type A from a surface term m.

The rules for elaboration are presented in Figure 3·14. Elaboration rules are written in a style of bidirectional type checking, with arrows pointing in the direction information flows. However, unlike bidirectional type checking, when checking a type that was inferred in the $\overleftarrow{\mathbf{Elab}}$-cast rule, elaboration adds a cast assertion that the two types are equal. Thus any conversion checking can be suspended until runtime. Additionally we will allow the mode to change at the type universe with the $\overleftarrow{\mathbf{Elab}}$-conv-$\star$ rule,

to avoid unneeded checks on the type universe. As formulated here, the elaboration procedure is terminating.

There are several desirable properties of elaboration that can be shown with the help of an erasure function (defined in 3·15). Erasure is defined over all syntactic forms, removing annotations, locations, and casts.

Theorem 3.26. *Elaborated terms preserve erasure.*
If $H \vdash m \; \textbf{\textit{Elab}} \; a \overset{\rightarrow}{:} A$ *then* $|m| = |a|$.
If $H \vdash m \, a \overset{\leftarrow}{:}_{\ell,o} A \; \textbf{\textit{Elab}} \, a$ *then* $|m| = |a|$.

Proof. By mutual induction on the **Elab** derivations. □

It follows that whenever an elaborated cast term evaluates, the corresponding surface term evaluates consistently. Explicitly,

Theorem 3.27. *Surface language and cast language have consistent evaluation.*
If $H \vdash m \; \textbf{\textit{Elab}} \; a \overset{\rightarrow}{:} A$, *and* $a \rightsquigarrow_* \star$ *then* $m \rightsquigarrow_* \star$.
If $H \vdash m \overset{\leftarrow}{:}_{\ell,o} A \; \textbf{\textit{Elab}} \; a$, *and* $a \rightsquigarrow_* (x : A) \to B$ *then there exists* N *and* M *such that* $m \rightsquigarrow_* (x : N) \to M$.

Proof. Since $a \rightsquigarrow_* a'$ implies $|a| \rightsquigarrow_* |a'|$ and $m \rightsquigarrow_* m'$ implies $|m| \rightsquigarrow_* |m'|$. □

Elaborated terms are well-cast in a well formed context. We will use H **ok** to mean for all x, $x : A \in H$ then $H \vdash A : \star$.

Theorem 3.28. *Elaborated terms are well-cast.*
For any H ***ok***, $H \vdash a \; \textbf{\textit{Elab}} \, m \overset{\rightarrow}{:} A$ *then* $H \vdash a : A$, $H \vdash A : \star$.
For any H ***ok***, $H \vdash A : \star$, $H \vdash m \overset{\leftarrow}{:}_{\ell,o} A \; \textbf{\textit{Elab}} \, a$ *then* $H \vdash a : A$.
For any H ***ok***, $H \vdash M \overset{\leftarrow}{:}_{\ell,o} \star \; \textbf{\textit{Elab}} \, A$ *then* $H \vdash A : \star$.

Proof. By mutual induction on **Elab** derivations. □

Some additional properties likely hold, though they have not yet been proven.

Claim 3.29. Every term well typed in the bidirectional surface language elaborates.
If $\vdash m \overset{\rightarrow}{:} M$ then there exists a and A such that $\vdash m \; \textbf{Elab} \; a \overset{\rightarrow}{:} A$.
If $\vdash m \overset{\leftarrow}{:} M$ and given ℓ, o then there exists a and A such that $\vdash m \overset{\leftarrow}{:}_{\ell,o} A \; \textbf{Elab} \, a$.

$$\frac{x : A \in H}{H \vdash x \,\mathbf{Elab}\, x \overrightarrow{:} A}\ \overrightarrow{\mathbf{Elab}}\text{-var}$$

$$\frac{}{H \vdash \star \,\mathbf{Elab}\, \star \overrightarrow{:} \star}\ \overrightarrow{\mathbf{Elab}}\text{-}\star$$

$$\frac{H \vdash M \overleftarrow{:_{\ell,\cdot}} \star \,\mathbf{Elab}\, A \quad H, x : A \vdash N \overleftarrow{:_{\ell',\cdot}} \star \,\mathbf{Elab}\, B}{H \vdash ((x : M_\ell) \to N_{\ell'}) \,\mathbf{Elab}\, ((x : A) \to B) \overrightarrow{:} \star}\ \overrightarrow{\mathbf{Elab}}\text{-fun-ty}$$

$$\frac{H \vdash m \,\mathbf{Elab}\, b \overrightarrow{:} (x : A) \to B \quad H \vdash n \overleftarrow{:_{\ell,Arg}} A \,\mathbf{Elab}\, a}{H \vdash (m_\ell \, n) \,\mathbf{Elab}\, (b \, a) \overrightarrow{:} B\,[x := a]}\ \overrightarrow{\mathbf{Elab}}\text{-fun-app}$$

$$\frac{H \vdash M \overleftarrow{:_{\ell',\cdot}} \star \,\mathbf{Elab}\, A \quad H \vdash m \overleftarrow{:_{\ell,\cdot}} A \,\mathbf{Elab}\, a}{H \vdash (m ::_\ell M^{\ell'}) \,\mathbf{Elab}\, a \overrightarrow{:} A}\ \overrightarrow{\mathbf{Elab}}\text{-::}$$

$$\frac{H, f : (x : A) \to B, x : A \vdash m \overleftarrow{:_{\ell,o.Bod_x}} B \,\mathbf{Elab}\, b}{H \vdash (\mathsf{fun}\, f\, x \Rightarrow m) \overleftarrow{:_{\ell,o}} (x : A) \to B \,\mathbf{Elab}\, (\mathsf{fun}\, f\, x \Rightarrow b)}\ \overleftarrow{\mathbf{Elab}}\text{-fun}$$

$$\frac{H \vdash m \,\mathbf{Elab}\, a \overrightarrow{:} A}{H \vdash m \overleftarrow{:_{\ell,o}} B \;\mathbf{Elab}\, (a ::_{A,\ell,o} B)}\ \overleftarrow{\mathbf{Elab}}\text{-cast}$$

$$\frac{H \vdash m \,\mathbf{Elab}\, a \overrightarrow{:} \star}{H \vdash m \overleftarrow{:_{\ell,o}} \star \;\mathbf{Elab}\, a}\ \overleftarrow{\mathbf{Elab}}\text{-conv-}\star$$

Figure 3·14: Elaboration

$$
\begin{array}{rcl}
|x| & = & x \\
|\star| & = & \star \\
|m ::_\ell M| & = & |m| \\
|(x : M_\ell) \to N_{\ell'}| & = & (x : |M|) \to |N| \\
|m_\ell\, n| & = & |m|\,|n| \\
|\mathsf{fun}\, f\, x \Rightarrow m| & = & \mathsf{fun}\, f\, x \Rightarrow |m| \\
|\Diamond| & = & \Diamond \\
|\Gamma, x : A| & = & |\Gamma|, x : |A| \\
|a ::_{A,\ell,o} B| & = & |a| \\
|(x : A) \to B| & = & (x : |A|) \to |B| \\
|\mathsf{fun}\, f\, x \Rightarrow b| & = & \mathsf{fun}\, f\, x \Rightarrow |b| \\
|b\, a| & = & |b|\,|a| \\
|H, x : M| & = & |H|, x : |M|
\end{array}
$$

Figure 3·15: Erasure

Which would lead to the corollary:

Claim 3.30. Blame never points to something that checked in the bidirectional system.
If $\vdash m \overset{\rightarrow}{:} M$, and $\vdash m\, \mathbf{Elab}\ a \overset{\rightarrow}{:} A$, then for no $a \rightsquigarrow_* a'$ will $a'\, \mathbf{Blame}_{\ell,o}$ occur.
If $\vdash m \overset{\leftarrow}{:} M$, and $\vdash m \overset{\leftarrow}{:} A\, \mathbf{Elab}\, a$, then for no $a \rightsquigarrow_* a'$ will $a'\, \mathbf{Blame}_{\ell,o}$ occur.

These properties are inspired by the gradual guarantee[SVCB15] for gradual typing.

3.6 Suitable Warnings

As presented here, not every cast corresponds to a reasonable warning. For instance, $(\lambda x \Rightarrow x) ::_{\star\to\star} \star \to \star$ is a possible output from elaboration. By the rules given the cast will not reduce without input, it will never cause blame. In fact since the user only interacts with the surface language, any cast $a ::_A B$ where $|A| \equiv |B|$ will not produce an understandable warning.

In Chapter 5 casts will be separated from the assertions that they contain, and it will be more clear how to extract warnings.

3.7 Related Work

3.7.1 Bidirectional Placement of Casts

This is not the first work to use bidirectional type checking to place errors. The Haskell compiler, GHC, supplements Hindley-Minler style type checking with bidirectionality to localize error messages. This approach was extended in [VPJMa12] which weakens the regular type checking to allow runtime casts[3]. The casts themselves are different from the ones described here since they do not optimistically compute, they will only give errors when reached. Though more restrictive than our casts, that system enforces parametricity, which makes sense in the context of Haskell.

3.7.2 Contract Systems

Several of the tricks and notations in this Chapter find their basis in the large amount of work on higher order contracts and gradual types. Higher order contracts were introduced in [FF02] as a way to dynamically enforce invariants of software interfaces, specifically higher order functions. The notion of blame dates at least that far back. Swapping the type cast of the input argument of a function type is reminiscent of that paper's use of blame contravariance, though it is presented in a much different way.

Contract semantics were revisited in [DFFF11, DTHF12] where a more specific correctness criteria based on blame is presented.

Contract systems still generally rely on users annotating their intentions explicitly. Similar to how programmers might include `asserts` in an imperative language. In this thesis annotations are added automatically though elaboration based on type annotations.

While there are similarities between contract systems and the cast system outlined

[3]Available with the $-$`fdefer` $-$ `type` $-$ `errors` compiler flag.

here, the cast system is designed to address only issues with definitional equality in a dependent type theory. Since contract systems are generally used in untyped languages with contracts written in the host language, definitional equality simply isn't applicable in the vast majority of contract systems.

Gradual Types

Types can be viewed as a very specific form of contracts that are usually enforced statically. **Gradual type systems** allow for a mixing of the static type checking and dynamic type assertions. Often type information can be inferred using standard techniques, allowing programmers to write fewer annotations.

Gradual type systems usually achieve this by adding a ? meta character into the type language to denote imprecise typing information. The first popular account of gradual type semantics appeared in [SVCB15] with the alliterative "gradual guarantee" which has inspired some of the properties targeted in this Chapter.

Additionally some of the formalism from this Chapter were inspired by the "Abstracting gradual typing" methodology [GCT16], where static evidence annotations become runtime checks.

This thesis borrows some notational conventions from gradual typing such as the $a :: A$ construct for type assertions.

A system for gradual dependent types has been proposed in [ETG19]. That paper is largely concerned with establishing a decidable type checking procedure via an approximate term normalization. However, that system retains the conventional style of definitional equality, so that it is possible, in principle, to get $\texttt{Vec}\,(1+x) \neq \texttt{Vec}\,(x+1)$ as a runtime error. Additionally it is unclear if adding the ? meta-symbol into an already very complicated type theory is easier or harder from the programmer's perspective.

The common motivation for gradual type systems is to gradually convert a code

base from untyped to (usually simply) typed code. However, anyone choosing to use a dependent type system has already bought into the usefulness of types in general and will probably not want fragments of completely untyped code. Gradually converting untyped code to include dependent types is far less plausible than gradually converting untyped code to use simple types. Especially considering that most real-life codebases will use effects, while adding effects into a simply typed programming language is straightforward, mixing dependent types and effects is a complicated area of ongoing research.

While the gradual typing goals of mixing static certainty with runtime checks are similar to our work here, the approach and details are different. Instead of trying to strengthen untyped languages by adding types, we take a dependent type system and weaken it with a cast operator. This leads to different trade-offs in the design space. For instance, we cannot support completely unannotated code, but we do not need to complicate the type language with a ? meta-symbol for uncertainty.

One might characterize this work in this Chapter as gradualizing only the definitional equality relation with a degenerate notion of imprecision.

Blame

Blame is one of the key ideas explored in the contract type and gradual types literature[WF09, Wad15, AJSW17]. Often the reasonableness of a system can be judged by the way blame is handled[Wad15]. This Chapter goes beyond blaming a source location and also tracks a witnessing observation that can also be made.

3.7.3 Refinement Style Approaches

This thesis describes a full spectrum dependently typed language. This means computation can appear uniformly in both term and type position. An alternative approach to dependent types is found in **refinement type systems**.

Refinement type systems restrict type dependency, possibly to specific base types such as `int` or `bool`. Under this restriction, it is straightforward to check type level equalities and additional properties hold at runtime.

One approach which explores this is **hybrid type checking** [Fla06, KF09, KF10] which performs "static analysis where possible, ... dynamic checks where necessary". However, there are several differences in that work: they have a simply typed system, static warnings for programmers are not considered, and type checking can reject "clearly ill-typed programs". For the system defined in this thesis there is no clear boundary between clearly ill-typed programs and subtly ill-typed programs, so we treat all potential inequalities uniformly with a static warning and a runtime check.

Another notable example is [OTMW04] which describes a refinement system that limits predicates to base types. Another example is [LT17], a refinement system treated in a specifically gradual way. A refinement type system with higher order features is gradualized in [ZMMW20].

Chapter 4

Data in the Surface Language

User defined data is an important part of a realistic programming language. Programmers need to be able to define concrete types that are meaningful for the problems they are trying to solve.

Dependent data types allow these user defined types, while also unifying many types that are handled as special cases in most mainstream languages. For instance, "primitive" data types like `Nat` and `Bool` are degenerate forms of dependent data. Dependent data can represent mathematical predicates like equality or the evenness of a number. Dependent data can also be used to preserve invariants, like the length of a list in `Vec`, or the "color" of a node in a red-black-tree.

The encoding scheme for data presented in Chapter 2 could handle all of these cases, so data types will not add any theoretical power to the system. However, those encodings are very inconvenient. Since our language is intended to be easy to use, user defined data will need to be supported.

In this Chapter we will show two different ways to add data to the surface language and bidirectional system. The first, a direct eliminator scheme, is meta-theoretically well behaved but cumbersome to use. It will peel off exactly one constructor at a time. The second is dependent pattern matching (similar to [Coq92]), and is extremely convenient, though its meta-theory is too difficult for a rigorous exposition here. It will allow any number of constructors to be matched simultaneously. The direct eliminator scheme is designed to have additional annotations that makes it a special

case of dependent pattern matching.

4.1 Data

A dependent data type is defined by a type constructor indexed[1] by arguments, and a set of data constructors that tag data and refine those arguments. Several familiar data types are defined in Figure 4·1. For example, the data type of natural numbers is defined with the type constructor `Nat` (which has no type arguments), the data constructors `Z` which takes no further information and the data constructor `S` which is formed with the prior number. The data type `Vec` has two type arguments corresponding to the type contained in the vector and its length; it has two data constructors that allow building an empty vector, or to add an element to the front of an existing vector.

Data defined in this style is simple to build and reason about, since data can only be created from its constructors. Unfortunately the details of data elimination are a little more involved.

4.2 Direct Elimination

How should a program observe data? Since a value of a given data type can only be created with one of the constructors from its definition, we can completely handle a data expression if each possible constructor is accounted for. For instance, `Nat` has the two constructors `Z` and `S` (which holds the preceding number), so the expression $\mathsf{case}\, n\ \{|\, \mathtt{Z} \Rightarrow \mathtt{Z} \,|\, \mathtt{S}\, x \Rightarrow x\}$ will extract the proceeding number from n (or 0 if $n = 0$).

[1] In more developed systems such as Coq and Agda data types may also have parameters. These are indices that apply uniformly over the type and every term construtor. For instance, the first argument of `Vec` is often given by a parameter. Parameters help remove clutter, help inference and erasure, and presumably have parametric properties in the sense of [Wad89]. However parameters are easy to simulate with indexes, so for simplicity we will only deal with *indexed* dependent data. Parameters can simulate indexes with a suitable equality type[SCA+12], though this is not always possible[LBMTT22, Section 8].

```
data Void : * {};

data Unit : * {
| tt : Unit
};

data Bool : * {
| True : Bool
| False : Bool
};

data Nat : * {
| Z : Nat
| S : Nat → Nat
};
three : Nat;
three = S (S (S Z)));
-- Syntactic sugar allows 3 = S (S (S Z)))

data Vec : (A : *) → Nat → * {
| Nil  : (A : *) → Vec A Z
| Cons : (A : *) → A → (x : Nat)
        → Vec A x → Vec A (S x)
};

someBools : Vec Bool 2;
someBools = Cons Bool True 1 (Cons Bool False 0 (Nil Bool));

data Id : (A : *) → A → A → * {
| refl  : (A : *) → (a : A) → Id A a a
};

threeEqThree : Id Nat 3 3;
threeEqThree = refl Nat 3;
```

Figure 4·1: Definitions of Common Data Types

We will call the term being inspected the **scrutinee**[2]. Here n is the scrutinee. $|\,\mathtt{Z} \Rightarrow \mathtt{Z}$ and $|\,\mathtt{S}\,x \Rightarrow x$ are called branches. The $\Rightarrow$ indicates that variables (such as x) may be bound.

This `Nat` elimination is type checkable since we know the intended output type of each branch (`Nat` in the above example), and can check that they are compatible. But in the presence of dependent types the output may not be obvious.

We will need to extend the syntax of `case`s to support dependent type checking. Specifically, we will need to add a **motive** annotation that allows the type checker to compute the output type of the branches if they vary in terms of the input. These annotations occur between the tehr angle brackets following the scrutinee. For instance, the `case` in the the `rep` function in Figure 4·2, has the motive $\mathtt{n'} : \mathtt{Nat} \Rightarrow \mathtt{Vec\ A\ n'}$. Each branch is typed knowing what it has observed about the input. For instance, in the first branch $\mathtt{Nil\ A} : \mathtt{Vec\ A\ 0}$. This allows the input to generalize to any appropriately typed term, even those that do not begin with a constructor. For instance, $\mathtt{rep\ Bool\ True}\ (f\,x) : \mathtt{Vec\ Bool}\ (f\,x)$ if $(f\,x) : \mathtt{Nat}$.

We may also want to use some values of the type level argument to calculate the motive, and type the branches. This will be allowed with additional scrutinees, bindings in the motive, and bindings in each branch. For example the `mapVec'` function in Figure 4·2 has motive $\mathtt{A} : \star \Rightarrow \mathtt{n} : \mathtt{Nat} \Rightarrow - : \mathtt{Vec\,A\,n} \Rightarrow (\mathtt{B} : \star) \to (\mathtt{A} \to \mathtt{B}) \to \mathtt{Vec\,B\,n}$. In `mapVec'` the motive allows the output to vary along with the length argument n that appears in the type position. In general, the motive will be treated like the typing annotations in Chapter 2: we will only allow obviously correct motives in a well typed term, motives are not always required, and the motive will be definitionally irrelevant.

This version of data can be given by extending the surface language syntax in Chapter 2, as in Figure 4·3. Data type constructors and data term constructors are

[2]Also called a **discriminee**.

```
-- generate a vector of a given length
rep : (A : *) → A → (n : Nat) → Vec A n ;
rep A a n =
  case n < n' : Nat  ⇒  Vec A n' >{
    | (Z)    ⇒  Nil A
    | (S p)  ⇒  Cons A a p (rep A a p)
    } ;

trues : Vec Bool 3;
trues = rep Bool True 3;
-- = [True,True,True]

mapVec' : (A : *) → (n : Nat) → Vec A n
 → (B : *) → (A → B)
 → Vec B n ;
mapVec' A n v =
  case A, n, v
    < A : *  ⇒  n : Nat  ⇒  _ : Vec A n
       ⇒  (B : *) → (A → B) → Vec B n
    >{
   | _  ⇒  _  ⇒  (Nil A)          ⇒
     \ B  ⇒  \ _  ⇒  Nil B
   | _  ⇒  _  ⇒  (Cons A a pn pv)  ⇒
     \ B  ⇒  \ f  ⇒  Cons B (f a) pn (mapVec' A pn pv B f)
  };

falses : Vec Bool 3;
falses = mapVec' Bool 3 trues Bool not;
-- = [False,False,False]
```

Figure 4·2: Direct Eliminator Scheme Examples

telescopes,

$\Delta, \Theta \quad ::= \quad \overline{(x : M) \to}$

data type identifier,

D

data constructor identifier,

d

contexts,

$$
\begin{array}{lll}
\Gamma & ::= & ... \\
& | & \Gamma, \mathsf{data}\, D\ :\ \Delta \to \star\ \left\{\overline{|\, d\ :\ \Theta \to D\overline{m}}\right\} \qquad \text{data def.} \\
& | & \Gamma, \mathsf{data}\, D\ :\ \Delta \to \star \qquad \text{abstract data}
\end{array}
$$

expressions,

$$
\begin{array}{lll}
m, ... & ::= & ... \\
& | & D \qquad \text{type cons.} \\
& | & d \qquad \text{data cons.} \\
& | & \mathsf{case}\, \overline{N}, n\ \left\{\overline{|\overline{x} \Rightarrow (d\, \overline{y}) \Rightarrow m}\right\} \qquad \text{data elim.} \\
& | & \mathsf{case}\, \overline{N}, n\ \langle \overline{x} \Rightarrow y : D\, \overline{x} \Rightarrow M \rangle\ \left\{\overline{|\overline{x} \Rightarrow (d\, \overline{y}) \Rightarrow m}\right\} \qquad \text{data elim. (motive)}
\end{array}
$$

values,

$$
\begin{array}{lll}
v & ::= & ... \\
& | & D\, \overline{v} \\
& | & d\, \overline{v}
\end{array}
$$

Figure 4·3: Surface Language (Direct Eliminator) Data

presented as function like identifiers (in this syntax they reuse function application to collect arguments).

As in the examples the case eliminator first takes the explicit type arguments, followed by a scrutinee list. Then optionally a motive that characterizes the output type of each branch with all the type arguments and arguments for each element of the scrutinees.

For instance, this case expression checks if a vector x : `Vec Bool 1` is empty:

$$\mathsf{case}\,\mathtt{Bool}, 1, x\ \langle y \Rightarrow z \Rightarrow s : \mathtt{Vec}\,y\,z \Rightarrow \mathtt{Bool}\rangle$$
$$\left\{\begin{array}{lll} |y & \Rightarrow z & \Rightarrow \mathtt{Nil} - \Rightarrow \mathtt{True} \\ |y & \Rightarrow z & \Rightarrow \mathtt{Cons} - - - - \Rightarrow \mathtt{False} \end{array}\right\}$$

The grammar includes a little more syntax than is strictly necessary, since the $\mathtt{Bool}, 1$ part of the scrutinee list could be inferred from the type of x and the $y \Rightarrow z \Rightarrow$ binders are not needed in the branch. This slightly verbose case eliminator syntax is designed to be forward compatible with the pattern matching system used in the rest of this thesis.

Additionally we define telescopes, which generalize zero or more typed bindings. This allows a much cleaner definition of data than is otherwise possible. Expressions in a list can be type checked against a telescope. For instance, the list $\mathtt{Nat}, 2, 2, \mathtt{refl}\,\mathtt{Nat}\,2$ type checks against $(X : \star) \to (y : X) \to (z : X) \to (- : \mathtt{Id}\ X\,y\,z)$. This becomes helpful in several situations, but especially when we need work with the listed arguments of the data type constructor. We will allow several syntactic puns, such as treating telescopes as prefixes for function types. For instance, if $\Delta = (y : \mathtt{Nat}) \to (z : \mathtt{Nat}) \to (- : \mathtt{Id}\ \mathtt{Nat}\,y\,z)$ then writing $f : \Delta \to \mathtt{Nat}$ will be shorthand for $f : (y : \mathtt{Nat}) \to (z : \mathtt{Nat}) \to \mathtt{Id}\ \mathtt{Nat}\,y\,z \to \mathtt{Nat}$.

In the presence of general recursion case elimination is powerful. For instance, all the functions in Figure 4·2 use recursion. Additionally, well-founded recursion can be used to represent inductive proofs.

Adding data allows for two new potential sources of bad behavior: incomplete matches, and nontermination from non-strictly positive data.

Incomplete Eliminations

Consider the match

$$x : \texttt{Nat} \vdash \texttt{case}\, x\, \langle s : \texttt{Nat} \Rightarrow \texttt{Bool}\rangle\, \{|\texttt{S}- \Rightarrow \texttt{True}\}$$

This match will "get stuck" if 0 is substituted for x. Recall that the key theorem of the surface language is type soundness, "well typed terms don't get stuck". Since verifying every constructor has a branch is relatively easy, the surface language TAS will require every constructor to be handled in order to type check with direct elimination. This is in contrast to most programming languages, which do allow incomplete patterns, though usually a warning is given, and a runtime error is raised if the scrutinee cannot be handled.

This thesis already has a philosophy for handling warnings and runtime errors through the cast language. When we get to the cast language data in Chapter 5, we will allow non-exhaustive data to be reported as a warning and that will allow "unmatched" errors to be observed at runtime.

For similar reasons, in the direct eliminator scheme, we will insist that each constructor is handled at most once, so there is no ambiguity for how a case is eliminated.

(non-)Strict Positivity

A more subtle concern is posed by data definitions that are not strictly positive. Consider the following definition,

```
data Bad : * {
| C : (Bad → Bad) → Bad
};

selfApply : Bad → Bad;
selfApply b =
  case b {
    | C f ⇒ f b
  };
```

```
loop : Bad;
loop = selfApply (C selfApply)
```

The `C` constructor in the definitions of `Bad` has a self reference in a negative position, (`Bad` → `Bad`) → `Bad`. Because of this, the `loop` term above will never reduce to a value.

Non-strictly positive data definitions can cause non-termination, independent of the two other sources of non-termination already considered (general recursion and type-in-type). Dependent type systems usually require a strictness check on data definitions to avoid this possibility. However, this would disallow some useful constructions like higher order abstract syntax. Since non-termination is already allowed in the surface TAS, we will not restrict the surface language to strictly positive data.

4.2.1 Type Assignment System

Before the typing rules for data can be considered, first some rules must be presented that will allow the simultaneous type-checking of lists and telescopes. These rules are listed in 4·4, and are standard. Telescopes are **ok** when they extend the context in an **ok** way. Lists of expressions can be said to have the type of the telescope if every expression in the list type checks successively.

Data definitions can be added to contexts if all of their constituents are well typed and **ok**. The rules are listed in Figure 4·5. The **ok**-abs-data rule allows data to be considered abstractly if it is formed with a plausible telescope. **ok**-data checks a full data definition with an abstract reference to a data definition in context, which allows recursive data definitions such as `Nat` which needs `Nat` to be in scope to define the `S` constructor. This thesis does not formalize a module syntax that adds data to context, though a very simple module system has been implemented in the prototype.

$$\frac{}{\Gamma \vdash . \, \mathbf{ok}} \, \mathbf{ok}\text{-Tel-empty}$$

$$\frac{\Gamma \vdash M : \star \quad \Gamma, x : M \vdash \Delta \, \mathbf{ok}}{\Gamma \vdash (x : M) \to \Delta \, \mathbf{ok}} \, \mathbf{ok}\text{-Tel-ext}$$

$$\frac{}{\Gamma \vdash , : .} \, \text{ty-ls-empty}$$

$$\frac{\Gamma, x : M \vdash \Delta \quad \Gamma \vdash m : M \quad \Gamma \vdash \overline{n}, : \Delta \, [x \coloneqq m]}{\Gamma \vdash m, \overline{n} \; : \; (x : M) \to \Delta} \, \text{ty-ls-ext}$$

Figure 4·4: Meta rules

$$\frac{\Gamma \vdash \Delta \, \mathbf{ok}}{\Gamma \vdash \mathsf{data} \, D \, \Delta \, \mathbf{ok}} \, \mathbf{ok}\text{-abs-data}$$

$$\frac{\Gamma \vdash \mathsf{data} \, D \, \Delta \, \mathbf{ok} \quad \forall d. \Gamma, \mathsf{data} \, D \, \Delta \vdash \Theta_d \, \mathbf{ok} \quad \forall d. \, \Gamma, \mathsf{data} \, D \, \Delta, \Theta_d \vdash \overline{m}_d : \Delta}{\Gamma \vdash \mathsf{data} \, D \; : \; \Delta \left\{ \overline{\mid d \; : \; \Theta_d \to D \overline{m}_d} \right\} \, \mathbf{ok}} \, \mathbf{ok}\text{-data}$$

Figure 4·5: Surface Language Data **ok**

It is taken for granted that any scheme to add well formed data into a type context is fine.

The type assignment system with direct elimination must be extended with the rules in 4·6. The ty-TCon and ty-Con rules allow type and data constructors to be used as functions of appropriate type. The ty-case <> rule types a case expression by ensuring that the correct data definition for D is in context, the scrutinee n has the correct type, the motive M is well formed under the type arguments and the scrutinee, finally every data constructor is verified to have a corresponding branch. ty-case allows for the same typing logic, but does not require the motive be annotated in syntax. In

both rules we allow telescopes to rename their variables with the shorthand $\overline{x} : \Delta$.

These rules make use of several convenient shorthands: $\mathsf{data}\, D\, \Delta \in \Gamma$ and $d\ :\ \Theta \to D\overline{m} \in \Gamma$ extract the type constructor definitions and data constructor definitions from the context respectively; telescopes can be added to context, such as $\Gamma, \overline{x} : \Delta, z : D\, \overline{x} \vdash M\ :\ \star$; telescopes can be added to context, reparameterized by an existing list of variables, $\Gamma, \overline{y}_d : \Theta$; and telescopes can be used as variable lists to substitute against, as in $\Theta \coloneqq \overline{y}_d$.

Extensions to the parallel reduction rules are listed in Figure 4·7. They follow the scheme of parallel reductions laid out in Chapter 2. The $\Rrightarrow$ -case-red rule[3] reduces a case expression by choosing the appropriate branch. The $\Rrightarrow$ -case <> -red rule removes the motive annotation, much like the annotation rule in Chapter 2. The rules $\Rrightarrow$ -case <>, $\Rrightarrow$ -D, and $\Rrightarrow$ -d keep the $\Rrightarrow$ relation reflexive. The reduction relation is generalized to lists in the expected way.

We are now in a position to select a sub relation of $\Rrightarrow$ reductions that will be used to characterize call-by-value evaluation. This relation could be used to prove type safety, and is close to the reduction used in the implementation. The rules are listed in Figure 4·8.

Finally we need to redefine what it means for a type context to be empty in the presence of data definitions (in Figure 4·9). We will say a context is empty only if it contains concrete data definitions.

Since a system with a similar presentation has proven type soundness in [SCA+12], we will not prove the type soundness of the system here.

Claim 4.1. The surface language extended with data and elimination preserves types over reduction.

Claim 4.2. The surface language extended with data and elimination has progress.

If $\Gamma\, \mathbf{Empty}$, $\Gamma \vdash m : M$, then m is a value, or $m \rightsquigarrow m'$.

[3] Also called ι, or Iota reduction.

$$\frac{\mathsf{data}\, D\, \Delta \in \Gamma}{\Gamma \vdash D\ :\ \Delta \to \star}\ \text{ty-TCon}$$

$$\frac{d\ :\ \Theta \to D\overline{m} \in \Gamma}{\Gamma \vdash d\ :\ \Theta \to D\overline{m}}\ \text{ty-Con}$$

$$\frac{\begin{array}{c}\mathsf{data}\, D\, \Delta\, \{...\} \in \Gamma \\ \Gamma \vdash n : D\overline{N} \\ \Gamma, \overline{x} : \Delta, z : D\,\overline{x} \vdash M : \star \\ \forall\, d\ :\ \Theta \to D\overline{o} \in \Gamma. \\ \Gamma, \overline{y}_d : \Theta \vdash m_d\,[\overline{x} \coloneqq \overline{o}\,[\Theta \coloneqq \overline{y}_d]] : M\,[\overline{x} \coloneqq \overline{o}\,[\Theta \coloneqq \overline{y}_d]\,, z \coloneqq d\,\overline{y}_d] \\ \text{No duplicate branches}\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{case}\,\overline{N}{,}n\ \langle \overline{x} \Rrightarrow z : D\,\overline{x} \Rightarrow M\rangle \left\{\overline{|\ \overline{x} \Rrightarrow d\overline{y}_d\ \Rightarrow m_d}\right\} \\ : M\left[\overline{x} \coloneqq \overline{N}, z \coloneqq n\right]\end{array}}\ \text{ty-}\mathsf{case}\ <>$$

$$\frac{\begin{array}{c}\mathsf{data}\, D\, \Delta\, \{...\} \in \Gamma \\ \Gamma \vdash n : D\overline{N} \\ \Gamma, \overline{x} : \Delta, z : D\,\overline{x} \vdash M : \star \\ \forall\, d\ :\ \Theta \to D\overline{o} \in \Gamma. \\ \Gamma, \overline{y}_d : \Theta \vdash m_d\,[\overline{x} \coloneqq \overline{o}\,[\Theta \coloneqq \overline{y}_d]] : M\,[\overline{x} \coloneqq \overline{o}\,[\Theta \coloneqq \overline{y}_d]\,, z \coloneqq d\,\overline{y}_d]\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{case}\,\overline{N}{,}n\ \left\{\overline{|\ \overline{x} \Rrightarrow d\overline{y}_d\ \Rightarrow m_d}\right\} \\ : M\left[\overline{x} \coloneqq \overline{N}, z \coloneqq n\right]\end{array}}\ \text{ty-}\mathsf{case}$$

Figure 4·6: Surface Language Data Typing

$$\frac{\begin{array}{c}\overline{N} \Rrightarrow \overline{N'} \quad \overline{m} \Rrightarrow \overline{m'} \\ \exists \overline{x \Rightarrow}(d\,\overline{y}_d) \Rightarrow m_d \in \left\{\overline{\mid \overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\}. \\ m_d \Rrightarrow m'_d \end{array}}{\mathsf{case}\,\overline{N}, d\overline{m}\ \left\{\overline{\mid \overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \Rrightarrow m_d\left[\overline{x} := \overline{N'}, \overline{y}_d := \overline{m'}\right]} \Rrightarrow \text{-}\mathsf{case}\text{-red}$$

$$\frac{\begin{array}{c}\overline{N} \Rrightarrow \overline{N'} \quad m \Rrightarrow m' \\ \forall \overline{x \Rightarrow}(d\,\overline{y}_d) \Rightarrow m_d \in \left\{\overline{\mid \Rightarrow x \Rightarrow (d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\}.\, m_d \Rrightarrow m'_d \end{array}}{\begin{array}{r}\mathsf{case}\,\overline{N}, m\ \langle ... \rangle \left\{\overline{\mid \Rightarrow x \Rightarrow (d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \\ \Rrightarrow \quad \mathsf{case}\,\overline{N'}, m'\left\{\overline{\mid \Rightarrow x \Rightarrow (d'\,\overline{y}_{d'}) \Rightarrow m'_{d'}}\right\}\end{array}} \Rrightarrow \text{-}\mathsf{case} <> \text{-red}$$

$$\frac{\begin{array}{c}\overline{N} \Rrightarrow \overline{N'} \quad m \Rrightarrow m' \\ M \Rrightarrow M' \\ \forall \overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'} \in \left\{\overline{\mid \Rightarrow x \Rightarrow (d\,\overline{y}_d) \Rightarrow m_d}\right\}.\, m_{d'} \Rrightarrow m'_{d'} \end{array}}{\begin{array}{r}\mathsf{case}\,\overline{N}, m\ \langle \overline{x \Rightarrow} z : D\,\overline{x} \Rightarrow M \rangle \left\{\overline{\mid \Rightarrow x \Rightarrow (d\,\overline{y}_d) \Rightarrow m_d}\right\} \Rrightarrow \\ \mathsf{case}\,\overline{N}, m'\ \langle \overline{x \Rightarrow} z : D\,\overline{x} \Rightarrow M' \rangle \left\{\overline{\mid \Rightarrow x \Rightarrow (d\,\overline{y}_d) \Rightarrow m'_d}\right\}\end{array}} \Rrightarrow \text{-}\mathsf{case} <>$$

$$\frac{}{D \Rrightarrow D} \Rrightarrow \text{-D}$$

$$\frac{}{d \Rrightarrow d} \Rrightarrow \text{-d}$$

Figure 4·7: Surface Language Data Reduction

$$\frac{}{\begin{array}{l}\mathsf{case}\,\overline{N}, n\ \langle ... \rangle \left\{\overline{|\Rightarrow x \Rightarrow (d\,\overline{y}) \Rightarrow m}\right\} \\ \rightsquigarrow \quad \mathsf{case}\,\overline{N}, n \left\{\overline{|\Rightarrow x \Rightarrow (d\,\overline{y}) \Rightarrow m}\right\}\end{array}} \rightsquigarrow \text{-case <>}$$

$$\frac{\exists \overline{x \Rightarrow}(d\,\overline{y}_d) \Rightarrow m_d \in \left\{\overline{|\,\overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\}}{\mathsf{case}\,\overline{V}, d\overline{v}\ \left\{\overline{|\,\overline{x \Rightarrow}(d'\,\overline{y}_{d'}) \Rightarrow m_{d'}}\right\} \rightsquigarrow m_d\left[\overline{x} \coloneqq \overline{V}, \overline{y}_d \coloneqq \overline{v}\right]} \rightsquigarrow \text{-case-red}$$

$$\frac{\overline{N} \rightsquigarrow \overline{N'}}{\mathsf{case}\,\overline{N}, n\ \left\{\overline{|\,\overline{x \Rightarrow}(d\,\overline{y}) \Rightarrow m}\right\} \rightsquigarrow \mathsf{case}\,\overline{N'}, n\ \left\{\overline{|\,\overline{x \Rightarrow}(d\,\overline{y}) \Rightarrow m}\right\}}$$

$$\frac{n \rightsquigarrow n'}{\mathsf{case}\,\overline{V}, n\ \left\{\overline{|\,\overline{x \Rightarrow}(d\,\overline{y}) \Rightarrow m}\right\} \rightsquigarrow \mathsf{case}\,\overline{V}, n'\ \left\{\overline{|\,\overline{x \Rightarrow}(d\,\overline{y}) \Rightarrow m}\right\}}$$

Figure 4·8: Surface Language Data Call-by-Value

$$\frac{}{\Diamond\ \mathbf{Empty}}\ \text{Empty-ctx}$$

$$\frac{\Gamma\ \mathbf{Empty} \quad \Gamma \vdash \mathsf{data}\,D\ :\ \Delta \left\{\overline{|\,d\ :\ \Theta \to D\overline{m}}\right\}\ \mathbf{ok}}{\Gamma, \mathsf{data}\,D\ :\ \Delta \left\{\overline{|\,d\ :\ \Theta \to D\overline{m}}\right\}\ \mathbf{Empty}}\ \text{Empty-ctx}$$

Figure 4·9: Surface Language Empty

Claim 4.3. The surface language extended with data and a direct elimnator scheme is type sound.

4.2.2 Bidirectional Type Checking

A bidirectional type checking procedure exists for the type assignment rules listed above. An outline of these rules is in Figure 4·10.

The type of data constructors and type constructors can always be inferred. A case with a motive will have its type inferred, and the motive will be used to check every branch in the $\overrightarrow{ty}$-case $<>$ rule. An unmotivated case will be type checked by an argument ignoring type dependency with the $\overleftarrow{ty}$-case rule.

The desired bidirectional properties hold.

Claim 4.4. The data extension to the bidirectional surface language is type sound.

Claim 4.5. The data extension to the bidirectional surface language is weakly annotatable from the data extension of the surface language.

This is a minimal (and somewhat crude) accounting of bidirectional data in the direct eliminator style. It is possible to imagine syntactic sugar that doesn't require the $\overline{N}$, and $\overline{x \Rightarrow}$ the in case expression of the $\overleftarrow{ty}$-case rule. In the rule $\overrightarrow{ty}$-case $<>$ it is also possible to imagine some type constructor arguments being inferred. These features and more will be subsumed by the dependent pattern matching of the next section.

4.3 Pattern Matching

Unfortunately, the direct eliminator style is cumbersome for programmers to deal with. For instance, Figure 4·11 shows how `Vec` data can be directly eliminated to extract the first element of a non-empty list in the definition of `head′`. The `head′` function needs to redirect unreachable vector inputs to a dummy type (`Unit`) and requires several copies of the same `A` variable that are not identified automatically

$$\frac{\mathsf{data}\, D\, \Delta \in \Gamma}{\Gamma \vdash D \overrightarrow{:} \Delta \to *}\ \overrightarrow{ty}\text{-TCon}$$

$$\frac{d\ :\ \Theta \to D\overline{m} \in \Gamma}{\Gamma \vdash d \overrightarrow{:} \Theta \to D\overline{m}}\ \overrightarrow{ty}\text{-Con}$$

$$\frac{\begin{array}{c}\mathsf{data}\, D\, \Delta\, \{...\} \in \Gamma \\ \Gamma \vdash \overline{N} \overleftarrow{:} \Delta \quad \Gamma \vdash n \overleftarrow{:} D\overline{N} \\ \Gamma, \overline{x} : \Delta, z : D\, \overline{x} \vdash M \overleftarrow{:} \star \\ \forall\, d\ :\ \Theta \to D\overline{o} \in \Gamma.\quad \Gamma, \overline{y}_d : \Theta \vdash m_d\, [\overline{x} := \overline{o}'] \overleftarrow{:} M\, [\overline{x} := \overline{o}', z := d\, \overline{y}_d]\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{case}\, \overline{N}, n\ \langle \overline{x} \Rightarrow z : D\, \overline{x} \Rightarrow M \rangle\, \left\{ \overline{\mid \overline{x} \Rightarrow (d\, \overline{y}_d) \Rightarrow m_d} \right\} \\ \overrightarrow{:} M\, \left[\overline{x} := \overline{N}, z := n\right]\end{array}}\ \overrightarrow{ty}\text{-}\mathsf{case}\ <>$$

$$\frac{\begin{array}{c}\mathsf{data}\, D\, \Delta\, \{...\} \in \Gamma \\ \Gamma \vdash \overline{N} \overleftarrow{:} \Delta \quad \Gamma \vdash n \overrightarrow{:} D\overline{N} \\ \Gamma \vdash M \overleftarrow{:} \star \\ \forall\, d\ :\ \Theta \to D\overline{o} \in \Gamma.\quad \Gamma, \overline{y}_d : \Theta \vdash m_d\, [\overline{x} := \overline{o}'] \overleftarrow{:} M\end{array}}{\Gamma \vdash \mathsf{case}\, \overline{N}, n\, \left\{ \overline{\mid \overline{x} \Rightarrow (d\, \overline{y}_d) \Rightarrow m_d} \right\} \overleftarrow{:} M}\ \overleftarrow{ty}\text{-}\mathsf{case}$$

where

$$\overline{o}' = \overline{o}\, [\Theta := \overline{y}_d]$$

Figure 4·10: Surface Language Bidirectional Type Checking

by the eliminator described in the last section. The usual solution is to extend case elimination with **pattern matching** (similar to [Coq92]).

Pattern matching is much more ergonomic than a direct eliminator case. In Figure 4·11, the `head` and `mapVec` functions that are defined though pattern matching is simpler and clearer. Additionally nested constructor matching is now possible as in the `sub3` function.

When pattern matching is extended to dependent types, variables will be assigned their definitions as needed, and unreachable branches can be omitted from code. For this reason, pattern matching has been considered an "essential" feature for dependently typed languages since [Coq92] and is implemented in most popular systems, such as Agda and the user facing language of Coq.

Figure 4·12 shows the extensions to the surface language for data and pattern matching. Our case expression matches a list of scrutinees, allowing us to be very precise about the typing of branches. Additionally, this style allows for syntactic sugar for easy definitions of functions by cases. The direct eliminator style case of the last section is a special case of pattern matching outlined here.

Patterns correspond to a specific form of expression syntax. When an expression matches a pattern it will capture the relevant subexpressions as variables. For instance, the expression, $\texttt{Cons}\,\texttt{Bool}\,\texttt{True}\,3\,(\texttt{Cons}\,\texttt{Bool}\,\texttt{False}\,2\,y')$ will match the patterns:

- $\texttt{Cons}\,w\,x\,y\,z$ with bindings $w = \texttt{Bool}$, $x = \texttt{True}$, $y = 3$, $z = \texttt{Cons}\,\texttt{Bool}\,\texttt{False}\,2\,y'$
- x with bindings $x = \texttt{Cons}\,\texttt{Bool}\,\texttt{True}\,3\,(\texttt{Cons}\,\texttt{Bool}\,\texttt{False}\,2\,y')$
- $\texttt{Cons} - x - (\texttt{Cons} - y - -)$ with bindings $x = \texttt{True}$, $y = \texttt{False}$

When patterns are used in the case construct, the appropriate branch will reduce with the correct bindings in scope.

```
-- direct eliminator style
head' : (A : *) → (n : Nat)
  → Vec A (S n)
  → A ;
head' A n v =
  case A, (S n), v <
    A' ⇒ n' ⇒ _ : Vec A' n' ⇒
      case n' < _ ⇒ *> {
        | (Z ) ⇒ Unit
        | (S _) ⇒ A'
      }
  >{
  | _ ⇒ _ ⇒ (Nil _ ) ⇒ tt
  | _ ⇒ _ ⇒ (Cons _ a _ _) ⇒ a
  } ;

-- pattern match style
head : (A : *) → (n : Nat)
  → Vec A (S n)
  → A ;
head A n v =
  case v < _ ⇒ A > {
  | (Cons _ a _ _) ⇒ a
  } ;

mapVec : (A : *) → (n : Nat) → Vec A n
  → (B : *) → (A → B)
  → Vec B n ;
mapVec A n v B f =
  case v
  < _ : Vec A n ⇒ Vec B n >{
  | (Nil A) ⇒ Nil B
  | (Cons A a pn pv)
    ⇒ Cons B (f a) pn (mapVec A pn pv B f)
  };

sub3 : Nat → Nat ;
sub3 n =
  case n < _ ⇒ Nat > {
  | (S (S (S x))) ⇒ x
  | _ ⇒ 0
  } ;
```

Figure 4·11: Eliminators vs. Pattern Matching

$$
\begin{array}{lcll}
m... & ::= & ... & \\
 & | & \mathsf{case}\,\overline{n},\,\left\{\overline{|\,\overline{pat} \Rightarrow m}\right\} & \text{data elim. without motive} \\
 & | & \mathsf{case}\,\overline{n},\,\langle \overline{x} \Rrightarrow M\rangle\,\left\{\overline{|\,\overline{pat} \Rightarrow m}\right\} & \text{data elim. with motive} \\
\text{patterns,} & & & \\
pat & ::= & x & \text{match a variable} \\
 & | & (d\,\overline{pat}) & \text{match a constructor}
\end{array}
$$

Figure 4·12: Surface Language Data

$$\frac{}{x\ \mathbf{Match}_{\{x:=m\}}\ m}$$

$$\frac{\overline{pat}\ \mathbf{Match}_{\sigma}\ \overline{m}}{d\overline{pat}\ \mathbf{Match}_{\sigma}\ d\overline{m}}$$

$$\frac{pat'\ \mathbf{Match}_{\sigma'}\ n \quad \overline{pat}\ \mathbf{Match}_{\sigma}\ \overline{m}}{pat', \overline{pat}\ \mathbf{Match}_{\sigma' \cup \sigma}\ n, \overline{m}}$$

$$\frac{}{.\ \mathbf{Match}_{\emptyset}\ .}$$

Figure 4·13: Surface Language Match

Therefore the expression $\mathsf{case}\,\mathtt{Cons}\,\mathtt{Bool}\,\mathtt{True}\,3\,(\mathtt{Cons}\,\mathtt{Bool}\,\mathtt{False}\,2\,y')$ $\{\mathtt{Cons} - x - (\mathtt{Cons} - y - -) \Rightarrow x \& y\}$ reduces to `False`.

The explicit rules for pattern matching are listed in Figure 4·13, where σ will hold a possibly empty set of assignments.

It is now easier for case branches to overlap, which could allow non-deterministic reduction. There are several plausible ways to handle this, such as requiring each branch to have independent patterns, or requiring patterns have the same behavior when they overlap [CPD14]. For the purposes of this thesis, we will use the programmatic convention that the first matching pattern takes precedence. For example, the following will type check

$$\mathsf{case}\,4\ \langle s : \mathtt{Nat} \Rightarrow \mathtt{Bool}\rangle\ \{|\mathtt{S}\,(\mathtt{S}\,-) \Rightarrow \mathtt{True}\,|\,- \Rightarrow \mathtt{False}\}$$

and it will reduce to `True`.

While pattern matching is an extremely practical feature, typing these expressions tends to be messy. To implement dependently typed pattern matching, a procedure is needed to resolve the equational constraints that arise within each pattern, and to confirm the impossibility of unwritten branches.

Since arbitrary computation can be embedded in the arguments of a type constructor[4], the equational constraints are undecidable in general. Any approach to constraint solving will have to be an approximation that performs well enough in practice. Usually this procedure takes the form of a first order unification.

4.3.1 First Order Unification

When type checking the branches of a case expression, the patterns are interpreted as expressions under bindings for each variable used in the pattern. If these equations can be unified, then the branch will type-check under the variable assignments, with the additional typing information. For instance,

Example 4.6. Type checking by unification

The pattern $\mathtt{Cons}\,x\ (\mathtt{S}\,y)\ 2\,z$ could be checked against the type $\mathtt{Vec}\,\mathtt{Nat}\,w$.

This implies the typings $x : \star$, $y : \mathtt{Nat}$, $(\mathtt{S}\,y) : x$, $2 : \mathtt{Nat}$, $z : \mathtt{Vec}\,x\,2$, $(\mathtt{Cons}\,x\ (\mathtt{S}\,y)\ 2\,z) : \mathtt{Vec}\,\mathtt{Nat}\,w$.

Which in turn imply the equalities $x = \mathtt{Nat}$,$w = 3$

This is a simple example, in the worst case we may have equations in the form $m\,n = m'\,n'$ which are hard to solve directly (but may become easy to solve if assignment of $m = \lambda x.x$, and $m' = \lambda - .0$ are discovered).

[4]At least in a full spectrum theory, such as the one we study here.

$$\frac{}{U\,(\emptyset, \emptyset)}\ \text{U-emp}$$

$$\frac{U\,(E, S) \quad m \equiv m'}{U\,(\{m \approx m'\} \cup E, S)}\ \text{U-Del}$$

$$\frac{U\,(E\,[x := m]\,, S\,[x := m])}{U\,(\{x \approx m\} \cup E, \{x := m\} \cup S)}\ \text{U-var-L}$$

$$\frac{U\,(E\,[x := m]\,, S\,[x := m])}{U\,(\{m \approx x\} \cup E, \{x := m\} \cup S)}\ \text{U-var-R}$$

$$\frac{U\left(\overline{m \approx m'} \cup E, S\right) \quad n \equiv d\overline{m} \quad n' \equiv d\overline{m'}}{U\,(\{n \approx n'\} \cup E, S)}\ \text{U-DCon-inj}$$

$$\frac{U\left(\overline{m \approx m'} \cup E, S\right) \quad N \equiv D\overline{m} \quad N' \equiv D\overline{m'}}{U\,(\{N \approx N'\} \cup E, S)}\ \text{U-TCon-inj}$$

Figure 4·14: Surface Language Unification

One advantage of the first order unification approach is that if the algorithm succeeds, it will succeed with a unique, most general solution. Since assignments are maximal, we are sure that a unified pattern will still be able to match any well typed syntax.

A simplified version of a typical unification procedure is listed in Figure 4·14. Unification is not guaranteed to terminate since it relies on definitional equalities, which are undecidable in the surface language. The unification procedure should also exclude the possibly cyclic assignments that could occur, such as $x = \mathtt{S}\,x$.

After the branches have type checked we should make sure that they are exhaustive, such that every possible branch will be covered. Usually this is done

by generating a set of patterns that would cover all combinations of constructors and proving that the unlisted branches are unreachable. In general it is undecidable whether any given pattern is impossible or not, so a practical approximation must be chosen. Usually a branch is characterized as unreachable if a contradiction is found in the unification procedure. A programmer will always have the ability to manually include non-obviously unreachable branches and prove their unreachability, or direct those branches to dummy outputs. Though there is a real risk that the unification procedure gets stuck in ways that are not clear to the programmer, and an understandable error message may be very difficult.

But that set of patterns must still be generated, given the explicit branches the programmer introduced. There is no clear best way to do this since a more fine division of patterns may allow enough additional definitional information to show unsatisfiability, while a more coarse division of patterns will be more efficient. Agda uses a tree branching approach that is efficient, but generates course patterns. The implementation of the language in this thesis generates patterns by a system of complements, this system seems slightly easier to implement, more uniform, and generates a finer set of patterns then the case trees used in Agda. However this approach is less performant then Agda in the worse case.

The bidirectional system can be extended with pattern matching with rules that look like

$$\frac{\begin{array}{c}\Gamma \vdash \overline{n} \overrightarrow{:} \Delta \\ \Gamma, \Delta \vdash M \overleftarrow{:} \star \\ \forall\, i\ \left(\Gamma \vdash \overline{pat}_i :_E ?\Delta \quad U\left(E, \sigma\right) \quad \sigma\left(\Gamma, |\overline{pat}_i|\right) \vdash \sigma m \overleftarrow{:} \sigma\left(M\left[\Delta \coloneqq \overline{pat}_i\right]\right)\right) \\ \Gamma \vdash \overline{\overline{pat}} : \Delta \textbf{ complete}\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{case}\, \overline{n},\ \langle \Delta \Rightarrow M \rangle \left\{\overline{|\, \overline{pat} \Rightarrow m}\right\} \\ \overrightarrow{:} M\left[\Delta \coloneqq \overline{n}\right]\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{n} \overrightarrow{:} \Delta \\ \forall\, i\ \left(\Gamma \vdash \overline{pat}_i :_E ?\Delta \quad U\left(E, \sigma\right) \quad \sigma\left(\Gamma, |\overline{pat}_i|\right) \vdash \sigma m \overleftarrow{:} \sigma\left(M\right)\right) \\ \Gamma \vdash \overline{\overline{pat}} : \Delta \textbf{ complete}\end{array}}{\Gamma \vdash \mathsf{case}\, \overline{n},\ \left\{\overline{|\, \overline{pat} \Rightarrow m}\right\} \overleftarrow{:} M}$$

Where $\Gamma \vdash \overline{pat} :_E ?\Delta$ is shorthand for a set of equations that allow a list of patterns to type check under Δ, and $\Gamma \vdash \overline{\overline{pat}} : \Delta$ **complete** is shorthand for the exhaustiveness check.

Conjecture 4.7. *There exists a suitable*[5] *extension to the surface language TAS that supports pattern matching style elimination*

Conjecture 4.8. *The bidirectional extension listed here is weakly annotatable with that extension to the surface language.*

These conjectures are not obvious since pattern matching's unification is not necessarily preserved under reduction, or even well typed substitution. These properties could likely be recovered by limiting the flexible (assignable) variables of unification to those that appear in the pattern. Though doing so seems a little

[5]Supporting at least subject reduction and type soundness.

arbitrary, and is limiting to the programmer. For instance, in `mapVec` the programmer will need to add the length of the `Vec` to the pattern match so that n is flexible. Another possible way to formalize a TAS is to use explicit contextual equalities as in [Sjö15].

The prototype implementation goes further than what is outlined here. For instance, the prototype allows some additional type annotations in the motive and for these annotations to switch the type inference of the scrutinee into a type-check. The implementation also has a simple syntax for modules, and even mutually defined data types. For simplicity these have been excluded from the presentation here.

4.4 Discussion

Pattern matching seems simple, but is surprisingly subtle.

Even without dependent types, pattern matching is a strange programming construct. How important is it that patterns correspond exactly to a subset of expression syntax? What about capture annotations or side conditions? Restricting patterns to constructors and variables means that it is hard to encapsulate functionality, a problem noticed as early as [Wad87]. This has led to making pattern behavior override-able in Scala via Extractor Objects. An extension in GHC allows some computations to happen within a pattern match via the `ViewPatterns` extension. It seems unreasonable to extend patterns to arbitrary computation (though this is allowed in the Curry language[6] as a syntax for its logical programming features).

In the presence of full spectrum dependent types, the perspective dramatically shifts. Any terminating typing procedure will necessarily exclude some type-able patterns and be unable to exclude some unreachable branches. Even though only data constructors are considered, dependent patterns are already attacking a much

[6]https://curry.pages.ps.informatik.uni-kiel.de/curry-lang.org/

more difficult problem than in the non-dependent case. It may make sense to extend the notion of pattern matching to include other useful but difficult features such as the with syntax of [MM04].

Epigram, Agda and Idris make pattern matching more powerful using with syntax that allows further pattern based branching by attaching a computation to a branch. This is justified as syntactic sugar that corresponds to helper functions that can be appropriately elaborated and type checked. The language described in this thesis does not use the with side condition since nested case expressions carry the same computational behavior, and the elaboration to the cast language will allow possibly questionable typing anyway.

More aggressive choices should be explored beyond the with construct. In principle it seems that dependent case expressions could be extended with relevant proof search, arbitrary computation, or some amount of constraint solving, without being any theoretically worse than the usual unification with conversion.

The details of pattern matching change the logical character of the system[CD18]. Since non-termination is allowed in the language described here, the logical issues that arise from patterns are less of a concern than the immediate logical unsoundness that was discussed in Chapter 2. However, it is worth noting that pattern matching as described here validates axiom K and thus appears unsuitable for univalent developments.

This Chapter has glossed over the definitional behavior of cases, since we plan to sidestep definitional issues entirely with the cast language. It is worth noting that there are several ways to set up the definitional reductions. Agda style case trees may result in unpredictable definitional equalities (in so far as definitional behavior is ever predictable). [CPD14] advocates for a more conservative approach that makes function definitions by cases definitional (but shifts the difficulties to

overlapping branches and does not allow the "first match" behavior programmers are used to). Another extreme would be to only allow reductions when the scrutinee is a value, similar to the work in [SCA+12]. Alternatively many partial reduction systems are possible, such that branches are eliminated as they are found unreachable and substitutions made as they are available. This last approach is experimentally implemented for the language defined here. However it is unclear how partial reduction could be handled in the meta theory.

Pattern matching complicates the simple story from Chapter 2, where the bidirectional system made the TAS system checkable by only adjusting annotations. Therefore we have only conjectured the existence of a suitable TAS system for pattern matching. If the definitional equality that feeds the TAS is generated by a system of reductions, any of the reduction strategies listed above will generate a different TAS with subtly different characteristics. For instance, insisting on a call-by-value case reduction will leave many equivalent computations unassociated. If the TAS system uses partial reductions it will need to inspect the constructors of the scrutinee in order to preserve typing when reduction eliminates branches. Agda style reductions need to extend syntax under reduction to account for side conditions.

Ideally the typing rule for pattern matching case expressions in the TAS should not use the notion of unification at all. Instead the rule should characterize the behavior that is required directly and formally[7]. An ideal rule might look like

[7] [Coq92] has a good informal description.

$$\frac{\begin{array}{rl} \Gamma \vdash \overline{n} : \Delta' & (\textit{scrutinees type check}) \\ \Gamma, \overline{x} : \Delta' \vdash M : \star & (\textit{motive exists and is well formed}) \\ \forall i.\, ? & (\textit{every branch is well typed over all possible instantiations}) \\ ? & (\textit{all scrutinees are handled}) \end{array}}{\Gamma \vdash \mathsf{case}\,\overline{n},\, \left\{ \overline{\overline{\mid pat \Rightarrow_i m_i}} \right\} : M\,[\overline{x} \coloneqq \overline{n}]}$$

4.5 Related work

4.5.1 Dependent Systems with Data

Many systems that target data only formalize a representative collection of data types, expecting the reader to be able to generalize the scheme. This data usually covers `Nat`s (for recursive types) and dependent pairs (for dependent types).

Unified Type Theory (UTT)[Luo90, Luo94] is an extension to ECC that specifies a scheme to define strictly positive data types by way of a logical framework defined in MLTT. This scheme generates primitive recursors for schematized data, and does not inherently support pattern matching.

The Calculus of Inductive Constructions (CIC) is an extension to the calculus of constructions that includes a system of first class data definitions. It evolved form Calculus of Constructions, and seems to have been first presented in the Coq manual, where a formulation is still maintained[8]. The meta theory was partially explored [PM93] which presented the Calculus of Constructions with Inductive Definitions (CCID) which is a restricted version of CIC. A bidirectional account of CIC is given in [LB21], though it uses a different style of bidirectionally then discussed here to maintain compatibility with the existing Coq system.

[8] https://coq.github.io/doc/v8.9/refman/language/cic.html

4.5.2 Dependent Pattern matching

The scheme for dependent pattern matching was first presented by Thierry Coquand in [Coq92]. Mcbride and Mckinna extended the power and theory of dependent pattern matching with several additional constructs such as with in [MM04]. Ulf Norell simplified the presentation of pattern matching in his thesis [Nor07]. The subtleties of dependently pattern matching are explored in [CD18], which has many good examples, some of which were motivated by long standing bugs in Agda.

Chapter 5

Data in the Cast Language

Chapter 3 showed how to use the TAS and the bidirectional system as a guide to build a dependently typed language with runtime equality. The TAS inspired the cast system, where the properties and lemmas of the TAS can be extended with casts. While the bidirectional system suggested how to localize uncertain assumptions that can be repurposed by elaboration as equality checks.

In this Chapter we will extend these systems for dependently indexed data and pattern matching. This will turn out to be more complicated than the system in Chapter 3, for two reasons. First, equality was only testable at types in Chapter 3 which allowed for some syntactic and semantic shortcuts. In the presence of dependent data, equality needs to be testable at terms, which will not necessarily have the same type. Second, the subtleties of pattern matching will need to be dealt with. While the intuition built up in Chapter 3 still holds, the cast language will need to be revised.

As before, we will take the (conjectured) surface language of Chapter 4 and construct a cast language with corresponding features. Though it is difficult to formalize a TAS and corresponding bidirectional system that has pattern matching, we will assume the unification of pattern matching belongs in the bidirectional system since it exists only to establish static correctness and is not needed for evaluation. Accordingly we will extend elaboration with a form of unification. Because we will not need to deal with unification in the cast language, the cast language can provide evidence that the cast system is **cast sound**. While the lack of a formal TAS

and bidirectional system in Chapter 4 will make the other properties of Chapter 3 impossible to prove here, we will design the system with an eye towards preserving them.

Despite these caveats, there is an interesting interpretation of data and pattern matching when extended to the cast system.

In a conventionally typed language, the normal forms of data terms have a valid data constructor in the head position (justifying the syntax of pattern matching). In the cast language, the normal form of data can have casts applied to an expression. If the casts are blameless then the constructor in the head position will match the data type. In the cast language pattern matching is extended with a path variable that can represent evidence of equality, then that evidence can be extracted and used in the body of the branch.

As in conventional pattern matching, since the type constructor is known, it is possible to check the coverage against all possible constructors. If every constructor is accounted for, only blameable scrutinees are possible. Quantifying over evidence of equality allows blame to be redirected, so if the program gets stuck in a pattern branch it can blame the original faulty assumption.

To account for "unreachable patterns" that are not stated in the surface language, we can record the proof of inequality for use at runtime. Since in the cast language, it is possible for a case expression to reduce into one of those "unreachable" branches. If this happens blame will be reflected back onto a specific problematic assumption made by the scrutinee. This will involve extending the cast language with operations to manipulate equality evidence directly. Proofs of inequality will be able to appear in terms.

5.1 Examples

Consider some of the following examples of how surface language pattern matches might elaborate.

5.1.1 Head

In the surface language the first element of x can be extracted with,

```
case x <_:Vec Bool (S n)  ⇒  Bool> {
| Cons _ a _ _  ⇒  a
}
```

Where x has the apparent type $\texttt{Vec}\,\texttt{Bool}\,(\texttt{S}\,n)$.

What can go wrong in the presence of casts?

- A blamable cast may have made x appear to be a `Vec` even when it is not. For instance, $\texttt{True} ::_\ell \texttt{Vec}\,\texttt{Bool}\,3$ (in Chapter 3 notation).
- The vector may be empty but cast to look like it is inhabited. For instance, $\texttt{Nil}\,\texttt{Bool} ::_\ell \texttt{Vec}\,\texttt{Bool}\,5$.
- The vector may contain elements that are not `Bool`. For instance, $\texttt{Cons}\,\texttt{Nat}\,3\,\ldots\,\ldots ::_\ell \texttt{Vec}\,\texttt{Bool}\,5$.

To handle these issues, elaboration can generate the following cast language term,

```
case x {
| (Cons A a y _) :: p  ⇒  a::(TCon0p)
| (Nil A) :: p  ⇒  !TCon1p
}
```

The elaborated `case` expression covers all possible constructors for the data type constructor `Vec`, including patterns that did appear in the surface term. Then unification solves the constraints to help elaborate the body.

In the first branch, the pattern captures typed variables, $A : \star$, $a : A$, $y : \texttt{Nat}$, while p is a path variable that contains evidence that the type of $\texttt{Cons}\,\texttt{A}\,\texttt{a}\,\texttt{y} -$ is $\texttt{Vec}\,\texttt{Bool}\,(\texttt{S}\,\texttt{n})$. So we will say, $p : \texttt{Vec}\ A\ (\texttt{S}\,y) \approx \texttt{Vec}\,\texttt{Bool}\,(\texttt{S}\,n)$. $TCon_0\,p$ extracts the 0th argument from the type constructor $p : \texttt{Vec}\ \underline{A}\ (\texttt{S}\,y) \approx \texttt{Vec}\,\underline{\texttt{Bool}}\,(\texttt{S}\,n)$ resulting in the type $TCon_0 p\ :\ A \approx \texttt{Bool}$. The body of the branch casts a along $TCon_0\,p$ to `Bool`. Casts will need to be generalized from Chapter 3 to contain evidence of equality.

In the the second branch, the pattern match gives $A : \star$, $p : \texttt{Vec}\ A\ \texttt{Z} \approx \texttt{Vec}\,\texttt{Bool}\,(\texttt{S}\,y)$. The body of that branch encodes the contradiction using explicit blame syntax (!) by observing $\texttt{Z} \neq \texttt{S}\,y$ with $TCon_1 p$. Any match in that branch must be blameable.

Since there is no assertion made in either branch, no warnings will be reported for this elaborated `case` term. Any failure that arises will be redirected to the scrutinee, which must have made a blameable assertion.

Again consider the ways x could go wrong:

- If the user tries to eliminate $x = \texttt{True} :: \texttt{Vec}\,\texttt{Bool}\,3$, the type constructor is not matched so the faulty assumption can be blamed automatically.

- If the scrutinee is an empty `Vec`, we will fall into the `Nil` branch, which will reflect the underlying faulty assumption, via the explicit blame syntax.

- If the `Vec` is inhabited by an incorrect type, such as $\texttt{Cons}\,\texttt{Nat}\,3 \ldots \ldots ::_\ell \texttt{Vec}\,\texttt{Bool}\,5$, the `case` will return $3 ::_{\ell,\ldots} \texttt{Bool}$ with a cast that rests on the blamable assertion of $\texttt{Vec}\,\texttt{Nat}\,5 \approx \texttt{Vec}\,\texttt{Bool}\,5$. When exactly this blame will surface depends on the evaluation and checking strategies. In the implemented language call-by-value and check-by-value are used at runtime and the blame will surface before the pattern match. Using a weak-head-normal-form strategy

the blame will be embedded in the resulting term and discovered whenever the "`Bool`" is eliminated.

5.1.2 Sum

The body of a pattern match may need to make use of type level facts discovered from the pattern match. For instance, in the surface language we can sum the two numbers in a `Vec` of length 2 with

```
case x <_:Vec Nat 2  ⇒  Nat> {
| Cons _ i _ (Cons _ j _ _)  ⇒  i+j
}
```

The elaboration procedure will produce

```
case x {
| (Cons Nat' i n' (Cons Nat'' j n'' rest):: p1):: p2  ⇒
  i::(TCon0p2) + j::(TCon0p1 ∪ TCon0p2)
| (Nil Nat') :: p  ⇒
  !TCon1p
| (Cons Nat' i n' (Nil Nat''):: p1):: p2  ⇒
  !(TCon1p1 ∪ DCon0(TCon1p2))
}
```

- In the first branch we have the variables in scope, $Nat' : \star$, $Nat'' : \star$, $i : Nat'$, $j : Nat''$, $p1 : \texttt{Vec}\ Nat''\ (\texttt{S}\ n'') \approx \texttt{Vec}\ Nat'\ n'$, and $p2 : \texttt{Vec}\ Nat''\ (\texttt{S}\ n') \approx \texttt{Vec Nat 2}$.
  - This means the elaborator can construct $TCon_0(p2) : Nat' \approx \texttt{Nat}$, and $TCon_0(p1) : Nat'' \approx Nat'$. Thes facts can be combined to show $TCon_0(p1) \cup TCon_0(p2) : Nat'' \approx \texttt{Nat}$.
  - The elaborator knows what the type of every sub expression is supposed to be, so casts can be injected onto the i and j terms using evidence from the pattern.

- In the 2nd branch we have, $p : \texttt{Vec}\ Nat''\ 0 \approx \texttt{Vec}\ Nat'\ 2$.
  - Which is contradictory, by $TCon_1 p\ :\ 0\ \approx\ 2$.
- In the 3rd branch, $p1 : \texttt{Vec}\ Nat''\ 0 \approx \texttt{Vec}\ Nat'\ n'$, $p2 : \texttt{Vec}\ Nat'\ (\texttt{S}\ n') \approx \texttt{Vec Nat}\ 2$.
  - Which is unsatisfiable by $TCon_1(p1) \cup DCon_0(TCon_1\, p2)\ :\ 0 \approx 1$. We don't need to know which sub path is problematic beforehand, only that the combination causes trouble. If this branch is reached, we can observe a problem in at least one path.

5.1.3 Missing Branches

What about unstated branches that cannot be excluded with type information? Consider this partial pattern match where $\texttt{rept}\ :\ (x : \texttt{Nat}) \to \texttt{Vec Bool}\, x$,

```
case x <x: Nat  ⇒  Vec Bool x> {
| 2  ⇒  rept 2
}
```

will elaborate to

```
case x  {
| S (S (Z :: _) :: _) :: _  ⇒  rept 2
? Z :: _
? S (Z :: _) :: _
? S (S (S _ :: _) :: _)
}
```

Substitution can confirm that the explicit branch has exactly the type of the motive and does not need a cast[1]. Additionally the elaborator will form a covering of implicit patterns that handle any possible constructor. Since the unifier cannot find

[1] While it is possible that blame was embedded in the $(\texttt{S}(\texttt{S}(\texttt{Z} :: -) :: -) :: -)$ term, the cast system will allow $(\texttt{S}(\texttt{S}(\texttt{Z} :: -) :: -) :: -) \equiv 2$.

a contradiction for any of these cases, the user will be warned of possible runtime errors.

5.1.4 Congruence (embedding equalities in terms)

This surface expression that takes in a propositional proof that $2 = 2$ and uses the named witness to generate a vector of length 2, demonstrates some of the subtler possibilities that arise in dependently typed pattern matching.

```
case x <_:Id Nat 2 2  ⇒  Vec Bool 2> {
| refl _ a  ⇒  rep Bool True a
}
```

This will elaborate to

```
case x {
| (refl N a)::p  ⇒
 (rep Bool True (a :: (TCon0 p)))
   :: Vec Bool (TCon1 p)
}
```

In the branch, $N : \star$, $a : N$, and p : `Id` N a $a \approx$ `Id Nat` 2 2. Since we have p : `Id` N a $a \approx$ `Id Nat` 2 2, we can derive $TCon_0(p)$: $N \approx$ `Nat`. Which can be used in $a :: (TCon_0(p))$ to cast a from N to `Nat`. But then we need evidence that `Vec Bool` $(a :: (TCon_0(p))) \approx$ `Vec Bool` 2 to avoid a sperous assertion. First, we need to select the subterm of interest, `Vec Bool` $\underline{(a :: (TCon_0(p)))} \approx$ $Vect$ `Bool` $\underline{2}$. Equality evidence is constructed specifically so that it can be embedded into terms. If we have evidence, q, such that $q : (a :: (TCon_0(p))) \approx 2$ then `Vec Bool` $\underline{q}$: `Vec Bool` $\underline{(a :: (TCon_0(p)))} \approx$ `Vec Bool` $\underline{2}$.

The cast system will only require that terms are equated up to a definitional equality that disregards casts so instead of needing to show $a :: (TCon_0(p)) \approx 2$, we only have to show $a \approx 2$. Which we have in $TCon_1(p) : a \approx 2$ and $TCon_2(p) : a \approx 2$. The elaborator can choose either to get a well cast term, and

```
peek : Id Nat 0 1 → Nat
peek x =
case x <_: Id Nat 0 1 ⇒ Nat> {
  | (refl _ x :: w) ⇒ x :: (TCon_0 w)
}

-- under weak head evaluation
peek (refl 4 :: Id Nat 0 1) = 4
```

Figure 5·1: Cast Pattern Matching

while the pattern will behave consistently on blameless terms, different behavior is possible when blame is discoverable.

For instance, given the elaboration above,

- if x is `refl Nat 2 :: Id Nat 0 2 :: Id Nat 2 2` then blame will be discoverable from the $TCon_1$ observation.
- if x is `refl Nat 2 :: Id Nat 2 0 :: Id Nat 2 2` then blame will not be discoverable and a blameless `Vec` is constructed.

In general there is no way around this, equality evidence may be constructable in subtle ways. Not everything can be checked.

5.1.5 Peeking

Another example of a term that might potentially lead to unexpected behavior is the peek function defined in Figure 5·1. `peek` will ignore several discrepancies in the index of the `Id` type, if run in weak-head-normal-form[2]. As in Chapter 3, our formalism uses a minimal amount of checking to maintain cast soundness, though more eager checking is implemented in the prototype.

[2]The example can be extended to call-by-value with functions, $\texttt{peek}' : \texttt{Id}(\texttt{Unit} \to \texttt{Nat})(\lambda - \Rightarrow 0)(\lambda - \Rightarrow 1) \to \texttt{Unit} \to \texttt{Nat}$.

5.2 Syntax

The syntax for the cast language can be seen in Figure 5·2. There are several differences to note from Chapter 3.

In Chapter 3 casts did double duty in both asserting an equality and changing the type of an underlying term. Now casts (::) will hold evidence that may change the type of a term, and assertions ($\sim$) will assert the specific equalities. For instance, if we wanted to assume `Nat` and `Bool` are the same we could write $\texttt{Nat} \sim^{\star}_{\ell} \texttt{Bool}$ given a location ℓ to blame, since $\star$ is the type of both `Nat` and `Bool`. The cast operation will allow 1 to be used as a `Bool` by casting the assumption, $1 :: (\texttt{Nat} \sim^{\star}_{\ell} \texttt{Bool})$.

These assertions are written $a \sim^{C}_{\ell,o} b$ and will evaluate a and b in parallel until a head constructor is reached on each branch. If the constructor is the same it will commute out of the term. If the head constructor is different the term will get stuck with the information for the final blame message. For instance, $(1 \sim^{\texttt{Nat}}_{\ell} 2) \rightsquigarrow_{*} \texttt{S}\left(0 \sim^{\texttt{Nat}}_{\ell.DCon_0} 1\right)$, where $DCon_0$ records that the issue occurs in the 0th argument of the outer `S` constructor.

The presence of assertions allows an expression to have different interpretations. For instance, $(1 \sim^{\texttt{Nat}}_{\ell} 2)$, can be interpreted as 1, 2 or evidence that $1 \approx 2$. We call the concrete interpretations **endpoint**s, and they will be formalized in the next section.

Assertions can be chained together with $\cup$ when two expressions share an endpoint. For instance, if we have $\texttt{Nat} \sim^{\star}_{\ell} \texttt{Bool}$ and $\texttt{Bool} \sim^{\star}_{\ell'} \texttt{Unit}$, then we can associate `Nat` and `Unit`, with $(\texttt{Nat} \sim^{\star}_{\ell} \texttt{Bool}) \cup^{\star} (\texttt{Bool} \sim^{\star}_{\ell'} \texttt{Unit})$.

Additionally, the syntax needs a way to point out that when a pattern is ununifiable. This is what the explicit blame syntax (!) is for. For instance, if the unifier could derive $1 = 2$ from evidence $p_1 : 1 = x$ and $p_2 : 2 = x$ then elaboration can force blame with $!^{\texttt{Nat}}_{p_1 :\cup^{\texttt{Nat}} p_2}$. Holding the type information in the superscript isn't absolutely needed, but allows us to provide better error messages and reflects the

variables,			
x, f, p, q			
pattern,			
$patc$	$::=$	x	
	$\mid$	$\left(d\overline{patc} :: p\right)$	
cast expressions,			
a, b, A, B, C, L	$::=$	x	
	$\mid$	$a :: C$	cast
	$\mid$	$\star$	
	$\mid$	$(x : A) \to B$	
	$\mid$	$\mathsf{fun}\, f\, x \Rightarrow b$	
	$\mid$	$b\, a$	
	$\mid$	$a \sim^{C}_{\ell,o} b$	assertion
	$\mid$	$a \cup^{C} b$	union
	$\mid$	$!^{L'}_{L}$	force blame
	$\mid$	D_{Δ}	type cons.
	$\mid$	$d_{\Delta \to D\overline{a}}$	data cons.
	$\mid$	$\mathsf{case}\, \overline{a},\, \left\{\overline{\mid \overline{patc} \Rightarrow b}\ \overline{?_{\ell}\, \overline{patc}}\right\}$	data elim.
	$\mid$	$TCon_i\ L$	
	$\mid$	$DCon_i\ L$	
observations,			
o	$::=$	.	
	$\mid$	$o.Arg$	
	$\mid$	$o.Bod_a$	
	$\mid$	$o.App_a$	
	$\mid$	$o.TCon_i$	type cons. index
	$\mid$	$o.DCon_i$	data cons. arg.
contexts,			
Γ	$::=$	.	
	$\mid$	$x : A$	
	$\mid$	$x_p : a : A \approx b : B$	
	$\mid$	$\Gamma, \mathsf{data}\, D\ :\ \Delta \to \star\ \left\{\overline{\mid d\ :\ \Theta \to D\overline{a}}\right\}$	data definition
	$\mid$	$\Gamma, \mathsf{data}\, D\ :\ \Delta \to \star$	abstract data

Figure 5·2: Cast Language Syntax

implementation.

The syntax allows data, though with some differences from Chapter 4. Data constructors and data type constructors now carry their explicit types, to emphasize that the type information will be needed for some reductions[3]. Motives are no longer needed on `case` expressions, and type/cast tracking will be handled implicitly. In addition to the handled patterns, `case` expressions also will contain a covering of unhandled patterns, for use in warnings and errors. Further, we allow two new observations, to observe the indices of data type constructors and term constructors respectively. These observations correspond directly to injectivity steps of the unification procedure.

Pattern matching is extended with a new path variable position. These variables are bound from the extended notion of patterns, and contain evidence that the data expression has the expected type.

Pattern matching across dependent data types will allow for observations that were not possible in Chapter 3. For instance, it is now possible to observe specific arguments in type constructors and term constructors. Since function terms can appear as indices in data constructors and data type constructors it is now possible to observe functions though application App_a.

5.3 Endpoint Rules

Care must be taken so that typing is still sensible when an expression could have multiple interpretations. To do this we construct a new cast system to include term level endpoint information that will generalize the cast relation of Chapter 3.

Endpoint judgments are written in the form $\Gamma \vdash a \sqsupseteq a' : A$ which means we can interpret a as $a' : A$ in variable context Γ. For instance, $\texttt{Nat} \sim^{\star}_{\ell} \texttt{Bool} \sqsupseteq \texttt{Nat} : \star$ and

[3]This also reflects our implementation. A more efficient implementation would probably not include them in the term syntax.

$\texttt{Nat} \sim^{\star}_{\ell} \texttt{Bool} \sqsupseteq \texttt{Bool} : \star$.

5.3.1 Function Fragment

The rules for non-data terms are listed in Figure 5·3.

The $\sqsupseteq$-var and $\sqsupseteq$-$\star$ rules extend the usual type assignment judgments to endpoints. The $\sqsupseteq$-fun-ty and $\sqsupseteq$-fun-app rules makes sure different endpoints are used in a type consistent way. For example, $((\lambda x.x + 1) \sim not)\,(1 \sim \texttt{False}) \sqsupseteq \texttt{True}, 2$ and $(1 \sim 2) + (10 \sim 3) \sqsupseteq 4, 5, 11, 12$. The $\sqsupseteq$-: : rule allows using a term at a different type if the cast has compatible endpoints. The next rules allow endpoints to be extracted out of the left or right of $\sim$, and $\cup$ if they are well formed.

In addition to the usual type conversion rule we also have a term conversion rule. Term conversion is important so that $\cup$ can associate terms under reduction. For instance, $(x \sim 2) \cup (1 + 1 \sim y) \sqsupseteq x, y$. Additionally term conversion will allow for the equivalence of type cast information. For instance, $1 :: (\texttt{Nat} \sim \texttt{Bool}) \sqsupseteq 1 : \texttt{Nat}$.

Endpoints can be used to recover the more familiar notions in Figure 5·4. A term is well-cast when it is its own endpoint (formalized in the ty-def rule). Therefore a well-cast term can only have assert and unify syntax in proper positions of casts. Also we can suggestively write two endpoints as an equality with $\approx$.

As usual, the system needs a suitable definition of $\equiv$. We use an equivalence relation that respects reductions and substitutions while ignoring casts. Ignoring casts keeps terms from being made distinct by different casts, and prevents reductions from getting stuck for the purpose of equality. Note that unlike Chapter 3 this means equivalence will not preserve blame information and we will need to rely on a more specific reduction relation that preserves both equivalence and blame.

5.3.2 Data Endpoints

The rules for data endpoints are listed in Figure 5·5.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \sqsupseteq x : A} \sqsupseteq\text{-var} \qquad \frac{}{\Gamma \vdash \star \sqsupseteq \star : \star} \sqsupseteq\text{-}\star$$

$$\frac{\Gamma \vdash A \sqsupseteq A' : \star \quad \Gamma, x : A' \vdash B \sqsupseteq B' : \star}{\Gamma \vdash (x : A) \to B \sqsupseteq (x : A') \to B' : \star} \sqsupseteq \text{-fun-ty}$$

$$\frac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star \quad \Gamma, x : A \vdash b \sqsupseteq b' : B}{\Gamma \ \vdash \ \mathsf{fun}\, f\, x \Rightarrow b \ \sqsupseteq \ \mathsf{fun}\, f\, x \Rightarrow b \ : \ (x : A) \to B} \sqsupseteq \text{-fun}$$

$$\frac{\Gamma \vdash b \sqsupseteq b' : (x : A') \to B' \quad \Gamma \vdash a \sqsupseteq a' : A'}{\Gamma \vdash b\, a \sqsupseteq b'\, a' : B' \, [x \coloneqq a']} \sqsupseteq \text{-fun-app}$$

$$\frac{\begin{array}{c}\Gamma \vdash L \sqsupseteq A : \star \\ \Gamma \vdash L \sqsupseteq B : \star \\ \Gamma \vdash a \sqsupseteq a' : A\end{array}}{\Gamma \ \vdash \ a :: L \ \sqsupseteq \ a' :: L \ : \ B} \sqsupseteq \text{-::}$$

$$\frac{\begin{array}{c}\Gamma \vdash a \sqsupseteq a' : A' \\ \Gamma \vdash L \sqsupseteq A' : \star \\ \Gamma \vdash L \sqsupseteq C : \star\end{array}}{\Gamma \vdash a \sim^{L}_{\ell,o} b \sqsupseteq a' :: L \ : \ C} \sqsupseteq \text{- } \sim \text{L} \qquad \frac{\begin{array}{c}\Gamma \vdash b \sqsupseteq b' : B' \\ \Gamma \vdash L \sqsupseteq B' : \star \\ \Gamma \vdash L \sqsupseteq C : \star\end{array}}{\Gamma \vdash a \sim^{L}_{\ell,o} b \sqsupseteq b' :: L \ : \ C} \sqsupseteq \text{- } \sim \text{R}$$

$$\frac{\begin{array}{c}\Gamma \vdash a \sqsupseteq a' :: L : C \\ \Gamma \vdash a \sqsupseteq c' :: L : C \\ \Gamma \vdash b \sqsupseteq c' :: L : C\end{array}}{\Gamma \vdash a \cup^{L} b \sqsupseteq a' :: L \ : \ C} \sqsupseteq \text{- } \cup \text{L} \qquad \frac{\begin{array}{c}\Gamma \vdash b \sqsupseteq b' :: L : C \\ \Gamma \vdash a \sqsupseteq c' :: L : C \\ \Gamma \vdash b \sqsupseteq c' :: L : C\end{array}}{\Gamma \vdash a \cup^{L} b \sqsupseteq b' :: L \ : \ C} \sqsupseteq \text{- } \cup \text{R}$$

$$\frac{\begin{array}{c}\Gamma \vdash a \sqsupseteq a' : A' \\ A' \equiv B\end{array}}{\Gamma \vdash a \sqsupseteq a' : B} \sqsupseteq \text{-conv-ty} \qquad \frac{\begin{array}{c}\Gamma \vdash a \sqsupseteq a' : A' \\ a' \equiv b \\ \Gamma \vdash b : A'\end{array}}{\Gamma \vdash a \sqsupseteq b : A'} \sqsupseteq \text{-conv-trm}$$

$$\frac{\Gamma \vdash a \sqsupseteq a : A}{\Gamma \vdash a : A}\ \text{ty-def}$$

$$\frac{\Gamma \vdash a \sqsupseteq b : B \quad \Gamma \vdash a \sqsupseteq c : C}{\Gamma \ \vdash \ a \ : \ \ b \ : \ B \ \approx \ c \ : \ C}\ \approx\text{-def}$$

Figure 5·4: Definitions

Assertion variables have endpoints at each side of the equality given from the typing judgment. Data type constructors and data term constructors have the function type implied by their definitions (and annotations that match).

The $\sqsupseteq$-case states that a case expression has a corresponding case expression endpoint:

- The endpoints of the scrutinees correspond to an appropriate telescope that is compatible with the patterns[4].
- A motive B' must exist under the context extended with that telescope.
- Every pattern must cast-check against the telescope.
- The body of every pattern must have an endpoint consistent with the motive and pattern.
- The constructors of the patterns must form a complete covering of the telescope.

In $\sqsupseteq$-! direct blame allows evidence with obviously contradictory endpoints to inhabit any type.

Data types and data terms can be inspected with $\sqsupseteq$-TCon and $\sqsupseteq$-DCon. However the prior indices are needed to compute the types, so an index meta-function $@_i$ is defined (in Figure 5·6). For instance, $TCon_1(\texttt{Id}\,(\texttt{Nat} \sim \texttt{Bool})\,(1 \sim \texttt{True})\,(1 \sim \texttt{True})) \sqsupseteq 1 : \texttt{Nat}$ while also ... $\sqsupseteq \texttt{True} : \texttt{Bool}$.

[4]The endpoint and typing judgment can be extended to lists and telescopes.

$$\frac{p \ : \ a : A \approx b : B \ \in \ \Gamma}{\Gamma \vdash p \sqsupseteq a : A} \sqsupseteq\text{-var-L} \qquad \frac{p \ : \ a : A \approx b : B \ \in \ \Gamma}{\Gamma \vdash p \sqsupseteq b : B} \sqsupseteq\text{-var-R}$$

$$\frac{\mathsf{data}\ D \ : \ \Delta \to \star \ \in \ \Gamma}{\Gamma \vdash D_\Delta \sqsupseteq D_\Delta : \Delta \to \star} \sqsupseteq\text{-TCon} \qquad \frac{d \ : \ \Delta \to D\overline{a} \ \in \ \Gamma}{\Gamma \vdash d_{\Delta \to D\overline{a}} \sqsupseteq d_{\Delta \to D\overline{a}} : \Delta \to D\overline{a}} \sqsupseteq\text{-Con}$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{a} \sqsupseteq \overline{a'} : \Delta \\ \Gamma \vdash \Delta \ \mathbf{ok} \\ \Gamma, \Delta \vdash B' : \star \\ \forall \overline{p} \in \overline{|\ \overline{patc}}, \ \Gamma \vdash \overline{p} : \Delta \\ \forall \overline{p} \in \overline{|\ \overline{patc}}, \ \Gamma, (\overline{p} : \Delta) \vdash b \sqsupseteq b' : B' \\ \forall \overline{p} \in \overline{?\,\overline{patc'}}, \ \Gamma \vdash \overline{p} : \Delta \\ \Gamma \vdash \overline{|\ \overline{patc}}\,\overline{?\,\overline{patc'}} : \Delta \ \mathbf{Complete}\end{array}}{\Gamma \vdash \mathsf{case}\,\overline{a},\ \left\{\overline{|\ \overline{patc} \Rightarrow b}\,\overline{?\,\overline{patc'}}\right\} \sqsupseteq \mathsf{case}\,\overline{a'},\ \left\{\overline{|\ \overline{patc} \Rightarrow b'}\,\overline{?\,\overline{patc'}}\right\} : B'\ \left[\Delta \coloneqq \overline{a'}\right]} \sqsupseteq\text{-}\mathsf{case}$$

$$\frac{\begin{array}{c}\Gamma \vdash L' \sqsupseteq A : \star \\ \Gamma \vdash L' \sqsupseteq B : \star \\ \Gamma \vdash L \sqsupseteq a : A \\ \Gamma \vdash L \sqsupseteq b : B \\ \mathbf{head}\, a \neq \mathbf{head}\, b \\ \Gamma \vdash B : \star\end{array}}{\Gamma \vdash !_L^{L'} \sqsupseteq !_L^{L'} \ : \ B} \sqsupseteq\text{-!}$$

$$\frac{\Gamma \vdash b \sqsupseteq D_\Delta \overline{a} : \star \quad \mathbf{length}\,\Delta = \mathbf{length}\,\overline{a}}{\Gamma \vdash TCon_i\ b \sqsupseteq \Delta @_i \overline{a}} \sqsupseteq\text{-TCon}$$

$$\frac{\Gamma \vdash b \sqsupseteq d_{\Delta \to D\overline{b}}\overline{a} \ : D\overline{c} \quad \mathbf{length}\,\Delta = \mathbf{length}\,\overline{a}}{\Gamma \vdash DCon_i\ b \sqsupseteq \Delta @_i \overline{a}} \sqsupseteq\text{-DCon}$$

Figure 5·5: Cast Data Endpoint Rules

$$\begin{array}{lcl} ((x : A) \to \Delta)\,@_0\,(a, \overline{b}) & = & a : A \\ ((x : A) \to \Delta)\,@_i\,(a, \overline{b}) & = & (\Delta\,[x := a])\,@_{(i-1)}\overline{b} \end{array}$$

Figure 5·6: Typed Index Function

5.4 Reductions

5.4.1 Function Fragment

A selection of reduction operations is listed in Figure 5·7.

Function reduction happens as usual. Type universes reduce away in the following three rules. Function types commute through $\sim$ and $\cup$ when their type annotations are the type universe. When the type position is resolved to function type, arguments can be applied under :: and into $\sim$ and $\cup$. This allows data types to be treated like functions. For instance, $(\lambda x \Rightarrow \mathtt{S}x) \sim \mathtt{S}$ will never cause a blameable error.

Finally, there are reductions to consolidate cast bookkeeping. In addition to the listed reductions, a reduction can happen in any sub-position.

5.4.2 Data Reductions

Some reductions for data are listed in Figure 5·8. Elimination of data types is delegated to the **Match** judgment (that is unlisted). $TCon_i$, and $DCon_i$ observations reduce to the expected index. Data types and data terms consolidate over $\sim$ and $\cup$ when a head constructor is shared and the type annotation has resolved, this needs to be computed by applying the indices point-wise against the typing telescope (see Figure 5·9). This is why the telescope annotation is explicit in the formalism (so reductions can happen independent of a context).

The important properties of reduction are

- Assertions emit observably consistent constructors, and record the needed observations.

$$\overline{(\mathsf{fun}\, f\, x \Rightarrow b)\, a \rightsquigarrow b\,[f \coloneqq (\mathsf{fun}\, f\, x \Rightarrow b)\,, x \coloneqq a]}$$

$$\overline{A :: \star \rightsquigarrow A} \qquad \overline{\star \sim^{\star}_{\ell,o} \star \rightsquigarrow \star} \qquad \overline{\star \cup^{\star} \star \rightsquigarrow \star}$$

$$\overline{\left(((x : A) \to B) \sim^{\star}_{\ell,o} ((x : A') \to B')\right) \rightsquigarrow \left(x : \left(A \sim^{\star}_{\ell,o.Arg} A'\right)\right) \to \left(B \sim^{\star}_{\ell,o.Bod_x} B'\right)}$$

$$\overline{(((x : A) \to B) \cup^{\star} ((x : A') \to B')) \rightsquigarrow (x : (A \cup^{\star} A')) \to (B \cup^{\star} B')}$$

$$\overline{(b :: ((x : A) \to B))\, a \rightsquigarrow (b\,(a :: A)) ::\ B\,[x \coloneqq a :: A]}$$

$$\overline{\left(c \sim^{(x:A)\to B}_{\ell,o} b\right) a \rightsquigarrow \left(c\,(a :: A) \sim^{B[x:=a::A]}_{\ell,o.App_a} b\,(a :: A)\right)}$$

$$\overline{(c \cup^{(x:A)\to B} b)\, a \rightsquigarrow (c\,(a :: A)) \cup^{B[x:=a::A]} (b\,(a :: A))}$$

$$\overline{(a :: L') \sim^{L}_{\ell,o} b \rightsquigarrow a \sim^{L' \cup^{\star} L}_{\ell,o} b} \qquad \overline{a \sim^{L}_{\ell,o} (b :: L') \rightsquigarrow a \sim^{L \cup^{\star} L'}_{\ell,o} b}$$

$$\overline{(a :: L') \cup^{L} b \rightsquigarrow a \cup^{L' \cup^{\star} L} b} \qquad \overline{a \cup^{L} (b :: L') \rightsquigarrow a \cup^{L \cup^{\star} L'} b}$$

$$\overline{((a :: L) :: L') \rightsquigarrow a :: (L \cup^{\star} L')}$$

Figure 5·7: Cast Language Small Step (select rules)

- Sameness assertions will get stuck on inconsistent constructors.

5.4.3 Endpoint Preservation

The system preserves typed endpoints over reductions.

$$\frac{a \rightsquigarrow b \quad \Gamma \vdash a \sqsupseteq a' : A'}{\Gamma \vdash b \sqsupseteq a' : A'}$$

For the fragment without data, this can be shown with some modifications to the argument in Chapter 3. We conjecture that the proof could be extended to support data.

5.5 Cast soundness

For the fragment without data, we can show cast soundness. Again we conjecture that the proof could be extended to support data.

If $\Gamma \vdash a \sqsupseteq a' : A'$, Γ **Empty** then either a **Consistent**, a **Blame**$_{\ell,o}$, or $a \rightsquigarrow b$. Again progress can be shown by extending the usual TAS progress argument over the new constructs.

The **Consistent** judgment (Figure 5·10) generalizes being a value, to elements over $\cup$ and $\sim$. This only matters for functions, since they are treated extensionally, in the sense that blame should only be possible from a function if the same input results in different outputs. This relation can be restricted further to be more like a conventional value judgment. For instance, enforcing the arguments of a data type are consistent, but these restrictions are not needed here.

The a **Blame**$_{\ell,o}$ judgment means a witness of error can be extracted from the term a pointing to the original source location ℓ with observation o. The most important rule is

$$\frac{\overline{a}\ \left(\overline{|\ \overline{patc} \Rightarrow b}\right)\ \mathbf{Match}\ b'}{\mathsf{case}\,\overline{a},\ \left\{\overline{|\ \overline{patc} \Rightarrow b}\,\overline{?\,\overline{patc}}\right\} \rightsquigarrow b'}$$

$$\overline{TCon_i\ \ (D_\Delta \overline{a}) \rightsquigarrow a_i}$$

$$\overline{DCon_i\ \ (d_{\Delta \to D\bar{e}} \overline{a}) \rightsquigarrow a_i}$$

$$\frac{\mathbf{length}\,\Delta = \mathbf{length}\,\overline{a} = \mathbf{length}\,\overline{b}}{\left((D_\Delta \overline{a}) \sim^{\star}_{\ell,o} \left(D_\Delta \overline{b}\right)\right) \rightsquigarrow D_\Delta \left(\Delta @^{T \sim \ell,o} \left(\overline{a}, \overline{b}\right)\right)}$$

$$\frac{\mathbf{length}\,\Delta = \mathbf{length}\,\overline{a} = \mathbf{length}\,\overline{b}}{\left((d_{\Delta \to D\bar{e}} \overline{a}) \sim^{C}_{\ell,o} \left(d_{\Delta \to D\bar{e}} \overline{b}\right)\right) \rightsquigarrow d_{\Delta \to D\bar{e}} \left(\Delta @^{\mathbf{D} \sim \ell,o} \left(\overline{a}, \overline{b}\right)\right) :: C}$$

$$\frac{\mathbf{length}\,\Delta = \mathbf{length}\,\overline{a} = \mathbf{length}\,\overline{b}}{\left((D_\Delta \overline{a}) \cup^{\star} \left(D_\Delta \overline{b}\right)\right) \rightsquigarrow D_\Delta \left(\Delta @^{T \cup} \left(\overline{a}, \overline{b}\right)\right)}$$

$$\frac{\mathbf{length}\,\Delta = \mathbf{length}\,\overline{a} = \mathbf{length}\,\overline{b}}{\left((d_{\Delta \to D\bar{e}} \overline{a}) \sim^{C}_{\ell,o} \left(d_{\Delta \to D\bar{e}} \overline{b}\right)\right) \rightsquigarrow d_{\Delta \to D\bar{e}} \left(\Delta @^{\mathbf{D} \cup} \left(\overline{a}, \overline{b}\right)\right) :: C}$$

Figure 5·8: Data Reductions

$$\begin{array}{lcl}
\Delta @_0^{k \sim \ell,o} \left(\overline{a}, \overline{b}\right) & = & \Delta @^{k \sim \ell,o} \left(\overline{a}, \overline{b}\right) \\
\left((x : A) \to \Delta\right) @_i^{k \sim \ell,o} \left(\left(a, \overline{a}\right), \left(b, \overline{b}\right)\right) & = & \left(a \sim^{A}_{\ell,o.TCon_i} b\right), \\
 & & \left(\Delta \left[x \coloneqq \left(a \sim^{A}_{\ell,o.kCon_i} b\right)\right]\right) @_{i+1}^{\sim \ell,o} \left(\overline{a}, \overline{b}\right) \\
(.)\ @_i^{k \sim \ell,o} (., .) & = & . \\
\Delta @_0^{k \cup} \left(\overline{a}, \overline{b}\right) & = & \Delta @^{k \cup} \left(\overline{a}, \overline{b}\right) \\
\left((x : A) \to \Delta\right) @_i^{k \cup} \left(\left(a, \overline{a}\right), \left(b, \overline{b}\right)\right) & = & \left(a \cup^{A} b\right), \left(\Delta \left[x \coloneqq \left(a \cup^{A} b\right)\right]\right) @_{i+1}^{k \cup} \left(\overline{a}, \overline{b}\right) \\
(.)\ @_i^{k \cup} (., .) & = & .
\end{array}$$

Figure 5·9: Pointwise Indexing

$$\frac{}{\star\ \mathbf{Consistent}}$$

$$\frac{}{(x : A) \to B\ \mathbf{Consistent}}$$

$$\frac{}{(\mathsf{fun}\, f\, x \Rightarrow b)\ \ \mathbf{Consistent}_{\mathsf{fun}}}$$

$$\frac{a\ \mathbf{Consistent}_{\mathsf{fun}} \quad b\ \mathbf{Consistent}_{\mathsf{fun}}}{a \sim_{\ell,o}^{(x:A)\to B} b\ \mathbf{Consistent}_{\mathsf{fun}}}$$

$$\frac{a\ \mathbf{Consistent}_{\mathsf{fun}} \quad b\ \mathbf{Consistent}_{\mathsf{fun}}}{a \cup^{(x:A)\to B} b\ \mathbf{Consistent}_{\mathsf{fun}}}$$

$$\frac{a\ \mathbf{Consistent}_{\mathsf{fun}}}{a :: (x : A) \to B\ \mathbf{Consistent}_{\mathsf{fun}}}$$

$$\frac{\mathbf{length}\, \overline{a} < \mathbf{length}\, \Delta}{D_\Delta\, \overline{a}\ \mathbf{Consistent}_{\mathsf{fun}}}$$

$$\frac{\mathbf{length}\, \overline{b} < \mathbf{length}\, \Delta}{d_{\Delta \to D\overline{a}}\, \overline{b}\ \mathbf{Consistent}_{\mathsf{fun}}}$$

$$\frac{\mathbf{length}\, \overline{a} = \mathbf{length}\, \Delta}{D_\Delta\, \overline{a}\ \mathbf{Consistent}}$$

$$\frac{\mathbf{length}\, \overline{b} = \mathbf{length}\, \Delta}{d_{\Delta \to D\overline{a}}\, \overline{b}\ \mathbf{Consistent}}$$

Figure 5·10: Consistent

$$\frac{\mathbf{head}\ a \neq \mathbf{head}\ b}{a \sim^{L}_{\ell,o} b\ \mathbf{Blame}_{\ell,o}}$$

Blame can be recursively extracted out of every sub expression. For instance, $\left(\left(1 \sim^{\texttt{Nat}}_{\ell,app_1} 0\right) + 2\right)\ \mathbf{Blame}_{\ell,app_1}$.

There are two new sources of blame from the case construct. The cast language records every unhandled branch and if a scrutinee hits one of those branches the case will be blamed for in-exhaustiveness[5]. If a scrutinee list primitively contradicts the pattern coverage blame can be extracted from the scrutinee. Since our type system will ensure complete coverage (based only on constructors), if a scrutinee escapes the complete pattern match, it must be that there was blamable a cast in the scrutinee.

5.6 Elaboration

5.6.1 Unification

To handle elaboration over pattern matching, we will need to extend unification from Chapter 4 to accommodate casts (Figure 5·11). We will now track not just equational constraints, but also why the constraints hold, and why the types are the same. For instance, in the constraint notation $1 \approx^{L}_{C} \texttt{True}$ means we have the constraint $1 = \texttt{True}$ because of C and $\texttt{Nat} = \texttt{Bool}$ because of L. The $a \approx^{L}_{C} b$ constraint can be thought of as stating the term C has endpoints a, b. Solutions record the reasoning behind an assignment. For instance, $x :=_{C} 3$ means x will be assigned 3 because of C.

When terms are substituted over equations, E, the terms are substituted into terms and causes are substituted into causes. For instance, $\left(x \approx^{\texttt{Nat}}_{TCon_1 x} x + x\right)[x :=_{p} 3] = 3 \sim^{\texttt{Nat}}_{TCon_1 p} 3 + 3$, which is sensible when constraints are considered as a notation for endpoints.

[5]This runtime error is conventional in ML style languages, and is even how Agda handles incomplete matches.

$$\frac{}{U\left(\emptyset, \emptyset\right)}$$

$$\frac{U\left(E, S\right) \quad a \equiv a'}{U\left(\{a \approx_C^L a'\} \cup E, S\right)}$$

$$\frac{U\left(E\left[x :=_C a :: L\right], S\left[x :=_C a :: L\right]\right)}{U\left(\{x \approx_C^L a\} \cup E, \{x :=_C a :: L\} \cup S\right)}$$

$$\frac{U\left(\Delta@^{\approx TCon\_C}(\overline{b}, \overline{b'}) \cup E, a\right) \quad a \rightsquigarrow^* D_\Delta \overline{b} \quad a' \rightsquigarrow^* D_\Delta \overline{b'}}{U\left(\{a \approx_C^L a'\} \cup E, a\right)}$$

$$\frac{U\left(\Delta@^{\approx DCon\_C}(\overline{b}, \overline{b'}) \cup E, a\right) \quad a \rightsquigarrow^* d_{\Delta \to D\overline{d}} \overline{b} \quad a' \rightsquigarrow^* d_{\Delta \to D\overline{d}} \overline{b'}}{U\left(\{a \approx_C^L a'\} \cup E, a\right)}$$

$$\frac{U\left(\left\{a \approx_C^{L \cup^\star D} b\right\} \cup E, S\right)}{U\left(\{a \approx_C^L b :: D\} \cup E, S\right)}$$

Figure 5·11: Cast Language Unification Rules (select)

Additionally term constructor and type constructor injectivity is handled through an indexing operator $@^{\cdots}$ similar to evaluation. Finally we add rules to peel off casts so that unification remembers why terms have the appropriate type, and so unification will not get stuck.

5.6.2 Elaboration

Without data, elaboration can be extended from Chapter 3 by asserting a cast where an infer meets a check.

$$\frac{\Gamma \vdash m \,\mathbf{Elab}\; a \overrightarrow{:} A}{\Gamma \vdash m \overleftarrow{:_{\ell,o}} B \;\mathbf{Elab}\; \left(a :: \left(A \sim^{\star}_{\ell,o} B\right)\right)}\; \overleftarrow{\mathbf{Elab}}\text{-cast}$$

However, the situation becomes more complicated with pattern matched data. With the information extracted from unification, terms can be elaborated with blameless casts that mimic the behavior of pattern matching in Chapter 4. The information can also be used to redirect blame from "impossible" branches. However, the elaboration procedure also needs to be extended with an additional context of assignments to cast into and out of making a formal presentation difficult.

5.7 Prior Work

There is very little prior work that can be cited for this Chapter.

While univalent type theories are interested in tracking (possibly unique) equalities over data, those systems tend to make users deal with equalities explicitly. Since the goal here is an easy to use language, their approach did not seem applicable.

The most relevant work to this Chapter is [LBMTT22], which uses a gradual typing methodology to gradualize a version of the CIC. Their are a number of relevant differences between these works: That work needs to support a term level wildcard ?, where this chapter maintains the surface syntax. This Chapter supports fully indexed

data, that work only supported parameterized data.

5.8 Future Work

The system presented here improves on several earlier systems and implementation experiments. There is reason to believe things could be improved further. For instance, when pattern matching an uncast data the pattern must have a cast variable (so one is synthesized). It may be more efficient (and cleaner) to have a term that would correspond to reflexivity, as some of our earlier experimental systems had.

Currently the implementation is fairly consevative with pattern matches. Only variables that appear in case patterns are used as flexible variables for unification. This results in some unintuitive behavior, such as needing to pattern match on the length of the list in append.

```
append : (A : *) → (n : Nat) → Vec A n
 → (m : Nat) → Vec A m
 → Vec A (add n m) ;
append A n vn m vm =
 case n, vn < n' ⇒ _ : Vec A n' ⇒ Vec A (add n' m) >{
   | (Z)    ⇒ (Nil _) ⇒ vm
   | (S _)  ⇒ (Cons _ a pn pvn)
      ⇒ Cons A a (add pn m) (append A pn pvn m vm)
 };
```

In Chapter 4 we noted that there are different ways one could handle flexible variables in type checking. But elaboration opens up new possibilities. For instance, we have enough information that the elaborator could add n to the scrutinee list, and refine the pattern appropriately.

The system here hints at an extension to pattern matching syntax that could be explored more generally. It seems useful to be able to read equational information out of patterns, especially in settings with rich treatment of equality. Matching equalities directly could be a useful feature in Agda, or in univalent type theories

where manipulating equations is more critical.

Chapter 6

Notes and Future Work

The content of this thesis was achieved through much trial and error. There were several experiments that while interesting and promising, did not cleanly fit into the narrative of the first 5 Chapters. This Chapter contains an idealized[1] review of what was tried, and hopefully provide hints about how one might productively try again.

The goal was always to make a dependently typed language as approachable as python[2]. While I believe a full spectrum dependently typed language with runtime definitional equality checking, dependent pattern matching and with no restrictions on recursion are part of that practical language, there are still lingering usability issues. For those who share the dream of more reliable software through easier-to-use dependently typed languages, here are some lines of work for your consideration.

6.1 Automatic Testing

One of the advantages of type checking is the immediacy of feedback. This thesis outlines a system that will give warning messages immediately, but requires evaluation to give the detailed error messages that are most helpful when correcting a program. This is especially important if the user wants to use the system as a proof language, and will not generally execute their proofs. An automatic testing system recaptures

[1]For instance, automated testing procedure was originally specified and implemented on a different language than the cast language described in this thesis. Accordingly the notations have changed, to be sensible in the context of the rest of the thesis. Further, automated testing procedure has not yet been reimplemented on the current version of the prototype.

[2]We have perhaps succeeded in making a dependently typed language as approachable as Haskell.

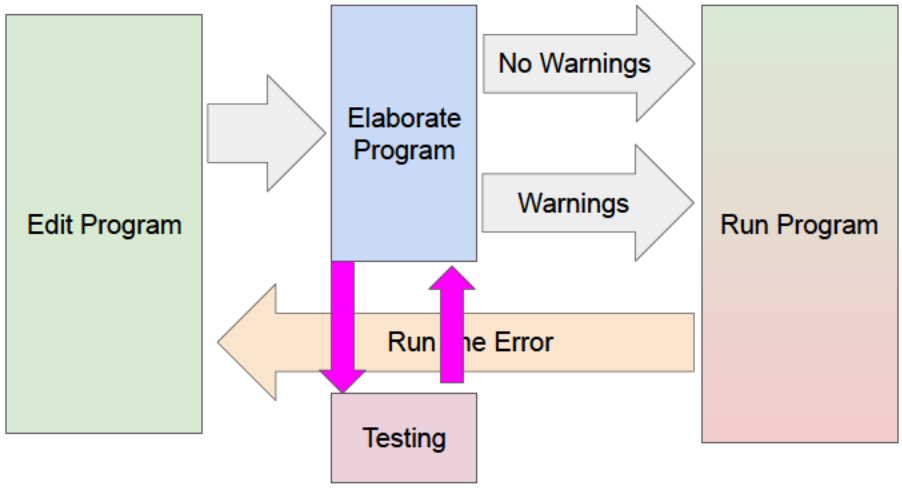


Figure 6·1: Ideal Workflow

some of that quicker feedback, by having a system that passively tries to find errors. This ideal workflow appears in Figure 6·1.

Adding a testing system to the prototype is more complicated than originally expected, and a good theory of dependently typed testing is incomplete.

Given those caveats, we do have a (practical?) testing procedure and some examples that might inform future theoretical work.

6.1.1 Observable Blame

Since this procedure operates over the cast language, we must decide what constitutes a reasonable testing environment

A one hole context $C[-]$ can be defined for the cast languages presented in this thesis. We can say that blame is observable if $\vdash a : A$ and $\vdash C[a] : B$ and $C[a] \rightsquigarrow_* b$ and $b\,\mathbf{Blame}_{\ell,\,o}$ for some $\ell \in \mathit{lables}\,(a)$ but $\ell \notin \mathit{lables}\,(C[-])$[3].

This error observability is semi-decidable in general (by enumerating all well typed

[3]Even here it is questionable which specific reduction rule to use (I suspect call-by-value would make the most sense for the implementation) and if the context should be limited to terms that elaborate from the surface language or all well cast terms (I use all well cast terms for convenience).

test variables,
X, Y, F, x, y, f, p, q
environment,

$$
\begin{array}{lcll}
I & ::= & \Gamma & \text{context (to hold data definitions)} \\
 & | & I, X : \star & \\
 & | & I, X \Rightarrow \star & \\
 & | & I, X \Rightarrow (x : Y) \to Fx & \\
 & | & I, f : (z : A) \to B & \\
 & | & I, fa \Rightarrow y & \\
 & | & I, X \Rightarrow D_{\Delta}\overline{x} & \\
 & | & I, x \Rightarrow (d_{\Delta \to D\overline{A}}\overline{y}) :: D\overline{p} & \\
 & | & I, p : a \approx_q b &
\end{array}
$$

Environmental Reduction,
$I \vdash a : A \ \rhd b : B$

Figure 6·2: Test Environment

syntax). But testing every context is infeasibly inefficient, especially if we try to synthesize terms of open ended types, like functions. An approximate approach can build partial testing contexts based on fixing observations to test variables.

6.1.2 Test Environment

The syntax for a test environment is listed in Figure 6·2. The first clause allows a cast context that is intended to contain data definitions for the term under test. We can declare test variables of type ($\star$) of partial function type and partial data type. We call them partial because they immediately defer to other test variables. The syntax allows functions to assign an output test variable to any input. The last 3 rules support data which as usual complicates things far more than you might expect. Since we use endpoints to handle data, we will need to handle them symbolically, they turn out to act like constraints that the testing environment should respect (where q represents evidence that the types of a and b are the same).

$fa \Rightarrow g, ga' \Rightarrow h$	written	$f\,a\,a' = h$
$X \Rightarrow (x : Y) \to \star, Y \Rightarrow a$	written	$X \Rightarrow (x : a) \to \star$
$X \Rightarrow (x : Y) \to Fx, Y \Rightarrow a$	written	$X \Rightarrow (x : a) \to Fx$
$f : (x : Y) \to Fx, Y \Rightarrow a$	written	$f : (x : a) \to Fx$
...		
any blamable term	written	!!

Figure 6·3: Environment and Observation Abbreviations

$$\frac{I \vdash c : C \rhd a : A \quad x \Rightarrow b \in I}{I \vdash c : C \rhd a\,[x \coloneqq b] : A\,[x \coloneqq b]}$$

$$\frac{I \vdash c : C \rhd a : A \quad xb \Rightarrow y \in I}{I \vdash c : C \rhd a\,[xb \coloneqq y] : A\,[xb \coloneqq y]}$$

$$\frac{I \vdash c : C \rhd a : A \quad a \rightsquigarrow a'}{I \vdash c : C \rhd a' : A}$$

Figure 6·4: Symbolic reduction

Some abbreviations are listed in Figure 6·3. The syntax is presented to encourage the use of fresh variables, but these variables will be collapsed into more readable examples. Additionally, throught this Chapter, we will use !! as a short hand for some term that contains obvous blame.

These testing contexts are used with the symbolic reductions rules listed in Figure 6·4 to simplify terms under test. Finally, environments can shift focus into subterm positions and apply test variables to functions (these rules are unlisted).

Example 6.1. Empty Type.

The "empty" type $((x : \star) \to x)$ can be inhabited by the cast language term $\lambda x \Rightarrow \star ::_\ell x$. According to the blame rules of Chapter 3 this term does not immediately produce blame without an input. We can observe blame by giving the function an argument, specifically by applying a function type as input. For instance, $(\lambda x \Rightarrow \star ::_\ell x)(\star \to \star) \rightsquigarrow_* \star ::_\ell (\star \to \star)$, which is blameable.

Our symbolic execution procedure would be able to discover that example by noting the function type $(x : \star) \to x$ and applying a test variable x of type $\star$. After normalizing it is easy to see that any x of the shape $x \;\Rightarrow\; - \to -$, would cause visible blame.

Example 6.2. Higher Order Functions.

Higher order functions can be handled similarly. For instance, if we have the cast language term

$$\lambda f \Rightarrow \star :: ((f \star) \sim_\ell \star) \quad : (\star \to \star) \to \star$$

our symbolic execution procedure would apply a test variable f of type $f : \star \to \star$. Then to observe a blamable term, symbolic execution would partially define the behavior of f, $f\star \;\Rightarrow\; - \to -$.

Example 6.3. Dependent types.

In the presence of dependent types, we must also consider blameable terms embedded in type formers. For instance, $!! \to \star \quad : \star$, can observe an error by inspecting its input, which would correspond to the filled context $((\lambda x \Rightarrow x :: \star) :: ([!! \to \star] \sim_\ell (\star \to \star)))\star \;\rightsquigarrow_*\; \star :: (!! \sim_\ell \star)$. Which is blamable.

The syntax of this system will be able to uncover every instance of observable blame. Since it has the data definitions it needs for the term's pattern matching and every unlisted data type can be simulated by a church like encoding. However it will also uncover many unobservable instances of blame.

Here are some additional constraints that can be applied to test environments:

- Γ holds only the well formed data definitions, for the data related to the inspected term.
- Variables, assignments, and paths are type and endpoint correct (with suitable extensions to the endpoint system that supports test environments).
- Test paths are blameless.

- Observably different outputs must come from observably different inputs (in the case of dependent function types, the argument should be considered as an input).

- Test variables are always fresh. For instance, $f : \texttt{Nat} \to \texttt{Nat}, f2 \Rightarrow x, f3 \Rightarrow x$ would not be allowed since $f2 = f3$ and is over specified.

We will call an environment where these constraints hold **plausible**.

Together these constraints comprise a decent testing procedure because:

- Testing can guide toward the labels of interest. For instance, we can move to labels that have not yet observed a concrete error. Terms without labels can be skipped entirely.

- Testing can choose assignments strategically avoiding or activating blame as desired.

- Since examples are built up partially the partial contexts can avoid introducing their own blame by construction.

- Testing can handle higher order functions, recursions, and self reference gracefully. For instance, the equations $f : \texttt{Nat} \to \texttt{Nat}$, $f\,(f\,0) \Rightarrow 1$ and $f\,(3) \Rightarrow 3$ can be in the context if there is an assignment that implies $f\,0 \neq 3$

However even plausible environments can flag blame that is not observable,

- Since there is no way for a term within the cast language to "observe" a distinction between some type formers, plausible environments cannot always be realized back to a term that would witness the bad behavior. For instance, the environment $F : \star \to \star, F \star \Rightarrow \star \to \star, F\ (\star \to \star) \Rightarrow \star$ will not correspond to an observable instance of blame, since there is no tem F that can be constructed

with those properties. In this way the test environment can make stronger observations than the cast language. The environment reflects a term language that has a type case construct.

- A version of parallel-or can be specified by the assignment context even though such a term is unconstructable in the language, *por* : `Bool` → `Bool` → `Bool`, *por loop* `True` ⇒ `True`, *por* `True` *loop* ⇒ `True`, *por* `False` `False` ⇒ `False`. Here all assignments are well typed, and each output can be differentiated by a different input.

6.1.3 Related and Future Work

Formalizing a complete and efficient testing procedure along these lines is still future work. However there have been other attempts to automatically test functional code that are worth mentioning.

Testing

Many of the testing strategies for typed functional programming trace their heritage to **property-based** testing in QuickCheck [CH01]. Property based testing involves writing functions that encode the properties of interest, and then randomly testing those functions. It is only natural that some of that research has been extended to dependent types:

- QuickChick[4] [DHL+14, LPP17, LGWH+17, Lam18] uses type-level predicates to construct generators with soundness and completeness properties, but without support for higher order functions. However, testing requires building type classes that establish the properties needed by the testing framework such as decidable equality. This is presumably out of reach of novice Coq users.

[4] `https://github.com/QuickChick/QuickChick`

  - Current work in this area uses coverage guided techniques in [LHP19] like those in symbolic execution.
  - More recently Benjamin Pierce has used American Fuzzy Lop, a binary fuzzer, on compiled Coq code as a way to generate counter examples[5].

- [DHT03] added QuickCheck style testing to (version 1 of) Agda.

Symbolic Execution

There are likely insights to be gained from the research on symbolic execution, especially work that deals with typed higher order functions. Symbolic execution is a technique to efficiently extract errors from programs. Usually this happens in the context of an imperative language with the assistance of an SMT solver. Symbolic execution can be supplemented with other techniques and an extensive literature exists on the topic.

The situation described in this section is unusual from the perspective of symbolic execution:

- The number of blamable source positions is limited by the location tags. Thus the search in this section is blame guided, rather than coverage guided.
- The language is dependently typed. Often the languages studied with symbolic execution are untyped or simply typed.
- The language in this section needs higher order functions. Often the research in this area focuses on base types that are efficiently handleable with an SMT solver, such as integer arithmetic.

This limits the prior work to relatively few papers

[5] `https://www.youtube.com/watch?v=dfZ94N0hS4I`

- A symbolic execution engine for Haskell is presented in [HXB+19], but at the time of publication it did not support higher order functions.
- A system for handling higher order functions is presented in [NTHVH17], however the system is designed for Racket and is untyped. Additionally it seems that there might be a state space explosion in the presence of higher order functions.
- [YFD21] extended and corrected some issues with [NTHVH17], but still works in a untyped environment. The authors note that there is still a lot of room to improve performance.
- Closest to the goal here, [LT20] uses game semantics to build a symbolic execution engine for a subset of ML with some nice theoretical properties.
- An early version of the procedure described in this section was presented as an extended abstract in [LZB20]. However conjectures made in that preliminary work were false (the procedure would flag unreachable errors, in the sense described above).

The appearance of *por* hints that the approach presented here could be revised in terms of games semantics. Though a dependently typed game semantics also seem to have a number of unanswered questions, that correspond roughly to the issues listed above. Though game semantics for dependent types is a complicated subject in and of itself, and has only begun to be studied in [VJA18].

6.2 Runtime Proof Search

Just as "obvious" equalities are missing from the definitional relation, "obvious" proofs and programs are not always conveniently available to the programmer. For

instance, in Agda it is possible to write a sorting function quickly using simple types. With effort is it possible to prove that sorting procedure correct by rewriting it with the necessarily dependently typed invariants. However, very little is offered in between. The problem is magnified if module boundaries hide the implementation details of a function, since those details are often exactly what is needed to make a proof! This is especially important for larger scale software where a library may require proof terms that while true are not provable from the exports of other libraries.

The solution proposed here is additional syntax that will search for a term of the type when resolved at runtime. Given the sorting function

$$\texttt{sort} : \texttt{List Nat} \to \texttt{List Nat}$$

and given the first order predicate that

$$\texttt{IsSorted} : \texttt{List Nat} \to *$$

then it is possible to assert that `sort` behaves as expected with

$$\lambda x.? : (x : \texttt{List Nat}) \to \texttt{IsSorted}\,(\texttt{sort}x)$$

This term will act like any other function at runtime, given a `List` input the function will verify that the `sort` correctly handled that input, or the term will give an error, or non-terminate.

Additionally, this would allow prototyping form first order specification. For instance,

$$\begin{array}{l} \mathit{data}\ \mathtt{Mult} : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat} \to * \ \mathit{where} \\ \quad \mathtt{base} : (x : \mathtt{Nat}) \to \mathtt{Mult}\ 0\ x\ 0 \\ \quad \mathtt{suc} : (x\, y\, z : \mathtt{Nat}) \to \mathtt{Mult}\, x\, y\, z \to \mathtt{Mult}\ (1 + x)\ y\, (y + z) \end{array}$$

can be used to prototype

$$\mathtt{div} = \lambda z.\lambda x.\mathtt{fst}\left(? : \sum y : \mathtt{Nat}.\mathtt{Mult} x\, y\, z\right)$$

The testing system in the last section could direct the computation of these solutions in advance. In some cases it is possible to find and report a contradiction.

Experiments along these lines have been limited to ground data types, and fix an arbitrary solution for every type problem. Ground data types do not need to worry about the path equalities since all the constructors will be concrete.

Non ground data can be very hard to work with when functions, function types or universes are considered. For instance,

$$? : \sum f : \mathtt{Nat} \to \mathtt{Nat}.\mathtt{Id}\,(f, \lambda x.x + 1)\,\&\mathtt{Id}\,(f, \lambda x.1 + x)$$

It is tempting to make the ? operator sensitive to more than just the type. For instance,

```
n : Nat;
n = ?;


pr : Id Nat n 1;
pr = refl Nat 1;
```

Will likely give the warning message "`n =?= 1 in Id Nat n 1`". It will then

likely give the runtime error "`0 =! = 1`". Since the only information to solve ? is the type `Nat` and an arbitrary term of type `Nat` will probably be solved with 0. Most users would expect the context to be considered and n to be solved with 1.

However constraints assigned in this manner can be extremely non-local. For instance,

```
n : Nat;
n = ?;

...

pr : Id Nat n 1;
pr = refl Nat 1;

...

pr2 : Id Nat n 2;
pr2 = refl Nat 2;
```

And things become even more complicated when solving is interleaved with computation. For instance,

```
n : Nat;
n = ?;

prf : Nat → Nat ;
prf x = (\ _ ⇒ x) (refl Nat x : Id Nat n x);
```

6.2.1 Prior Work

Proof search is often used for static term generation in dependently typed languages (for instance Coq tactics). A first order theorem prover is attached to Agda in [Nor07]. However it is rare to make those features available at runtime.

Logic programing languages such as Prolog[6], Datalog[7], and miniKanren[8] use "proof search" as their primary method of computation. Dependent data types can be seen as a kind of logical programming predicate. The Twelf project[9] makes use of runtime proof search and has some support for dependent types, but the underlying theory cannot be considered full spectrum. The Curry Language[10] performs logic programming in a Haskell-like language. Gradual dependent type research is working towards a similar goal [ETG19], but [LBMTT22] has a good explanation of why extending graduality to dependent indexed types is difficult.

6.3 Future work

6.3.1 Effects

The last and biggest hurdle to bring dependent types into a mainstream programming language is by providing a reasonable way to handle effects. Though dependent types and effects have been studied I am not aware of any full spectrum system that has implemented those theories. It is not even completely clear how best to add an effect system into Haskell, the closest "mainstream" language to the one studied here.

While trying carefully to avoid effects in this thesis, we still have encountered 2 important effects: blame-based error and non-termination.

Errors

The current system implements blame-based runtime errors and static warnings in a unique way. There is no control flow for errors built into the reduction or call-by-value relations, and there is no way to handle an error within the program. Every potential

[6] `https://www.swi-prolog.org`
[7] `https://docs.racket-lang.org/datalog`
[8] `http://minikanren.org`
[9] `http://twelf.org/wiki/Main\_Page`
[10] `https://curry.pages.ps.informatik.uni-kiel.de/curry-lang.org/`

error is linked to a static warning. There are a few features that would be good to experiment with.

Ideally we could allow users to provide proofs of equality to remove warnings by having them define and annotate an appropriate identity type. This would allow the language to act more like an Extensional Type Theory. Programmers could justify these proofs as a way to remove runtime checks and make code (and testing) faster. Just as with ETT many desirable properties such as function extensionality would still not be provable. We have pushed this to future work since there are already many explored strategies for dealing with equality proofs in an Intensional Type Theory that are suitable for avoiding warnings in the current implementation.

Currently blame-based errors aren't handled[11]. Programmers may want to use the information from a bad cast to build the final output, it might even be possible to capture a well typed term that witnesses the inequality. For instance,

```
f : Vec String 1 → String;
f x = case x {
  Cons _ a _ _  ⇒  a
}

h : (x : Nat) → String;
h x =
  handle{
    f (rep String "hi" x)
  } pr : x != 1  ⇒  "whoops" ;
```

Though additional research would be needed for exactly the form the contradictions should take if they are made available to the handler.

Handling effects in a dependent type system is subtle[12] since the handling construct can observe differences that should not otherwise be apparent. This is

[11] Or caught.

[12] Everything about dependent types is subtle.

most clearly seen in the generalization of Herbelin's paradox presented in [PT19]. The problem is that the value of a `Bool` term may depend on effects that cause logical unsoundness (or worse). The paradox can be presented in our system with an additional handling construct,

```
h : (u : Unit) → Bool;
h u =
  handle{
    case u {
      | tt ⇒ true
    }
  } _ ⇒ false ;

hIsTrue : (u : Unit) → Id Bool true (h u);
hIsTrue u =
  case u <u → Id Bool true (h u)>{
    | tt ⇒ refl Bool true
  };

hIsTrue !! : Id Bool true false
```

Interestingly this term is not as bad as the paradox would be in other settings. A warning is given so we would not expect logical soundness. If evaluated in weak-head-normal-form the term will produce blame witnessing the static warning given.

Non-termination

Non-termination is allowed, but it would be better to have it work in the same framework as equational warnings, namely warn when non-termination is possible, and try to find slow running code via automated testing (runtime errors for non termination are clearly not possible). Then we could say without caveat "programs without warnings are proofs". It might be possible for users to supply their own external proofs of termination[CSW14], or termination metrics.

Other effects

One of the difficulties of an effect system for dependent types is expressing the definitional equalities of the effect modality. Is `print"hello";print"world"` $\equiv$ `print"helloworld"` at type `IO Unit`? By delaying equality checks until runtime these issues can be avoided until the research space is better explored. Effects risk making computation mathematically inelegant. In this thesis we avoided this inelegance for an error effect with the blame relation. Something analogous could perhaps be applied to more interesting effect systems.

Both the symbolic execution and search above could be considered in terms of an effect in an effect system. Proof search could be localized though an effect modality, better communicating its non locality.

6.4 User studies

The main proposition of this work is that it will make dependent types easier to learn and use. This should be demonstrated empirically with user studies. Since the surface language has been implemented independently of the cast language, we have a rare opportunity to test the usefulness of elaboration on two nearly identical systems.

6.5 Semantics

This thesis has explored its systems using operational semantics, this has led to serviceable, but cumbersome, proofs. Ideally less syntactic semantics of a typed language in this style should be explored.

For instance, the entire system is designed with an unformalized notion of **observational equivalence**[13] in mind. While there has been some exploration into observational equivalence for dependent types in [Sjö15, JZSW10], it uses untyped

[13] Also called **contextual equivalence**.

observational equivalence, which is a weak version of the relation. A version of typed operational equivalence is considered in [VJA18] though they consider definitional distinctions observable. There is a difficult circularity when trying to define typed observational equivalence in a dependently typed setting. A good exploration of dependently typed observational equivalence would be an interesting and helpful direction for further study.

Chapter 7

Conclusion

This thesis has attempted to articulate and address a common hesitation around dependent types. Programmers do not want to be interrupted. Especially if the interruption is about a chance of an error. Addressing this legitimate concern has led to a new way of treating warnings in a dependent type system. By creating a parallel system where checks are made and given runtime behavior, programmers still get all of the benefits, but fewer drawbacks of dependent type systems.

This turned out to be surpassingly more subtle than expected. As we saw in Chapter 3, the programmer's intent needs to be inferred, so that a reasonable check can be localized. This is possible through an extension to bidirectional type checking. Runtime errors complicate the semantics, this issue was sidestepped by applying a new relation that extracts blame. Checks need their own runtime behavior, which is possible in the pure functional setting.

Further, user defined data turned out to be far more complicated than expected. Extending pattern matching to track equalities seems like a clever idea, however the formalism in Chapter 5 is still more complicated then we would like. It is unclear if a simpler approach is possible.

Finally, in Chapter 6, there are several ways to improve the current system and build towards future work: automated testing, runtime proof search and most of all a convenient embedding of effects.

The approach to warnings presented in this thesis may be more generally

applicable. Type systems can still be designed to harshly avoid errors, but by creating a parallel system where checks are made and given runtime behavior, the type system will be less imposing to new users. For instance, many interesting linear type systems are currently being explored, allowing warnings may make these systems more usable to programmers who are not used to those restrictions.

Dependent types have seemed on the verge of mainstream use for decades. While dependent types are not there yet, they have the unique potential to bridge the gap between those who program and those who prove. Each community has built invaluable expertise that could benefit the other. Once that connection is made solid, more robust software is the least we should expect.

While this thesis has not single handedly made this connection, I think it is a necessary piece of the puzzle.

Bibliography

[ADLO10] Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. $\pi\sigma$: Dependent types without the sugar. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming*, pages 40–55, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[AH04] David Aspinall and Martin Hofmann. Dependent types. In *Advanced Topics in Types and Programming Languages*, pages 45–86. MIT Press, 2004.

[AJSW17] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: Parametricity, with and without types. *Proceedings of the ACM on Programming Languages*, 1(ICFP), August 2017.

[Aug98] Lennart Augustsson. Cayenne a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. Association for Computing Machinery.

[BB08] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, pages 365–379, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BL84] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In Gilles Kahn, David B. MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 1–50, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013.

[Car86] Luca Cardelli. A polymorphic [lambda]-calculus with type: Type. Technical report, DEC System Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, May 1986.

[Cas14] Chris Casinghino. *Combining proofs and programs.* PhD thesis, University of Pennsylvania, 2014.

[CD18] Jesper Cockx and Dominique Devriese. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *Journal of Functional Programming*, 28:e12, 2018.

[CDT12] The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at `http://coq.inria.fr/doc`.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, February 1988.

[CH01] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *ACM SIGPLAN Notices*, 46(4):53–64, 2001.

[Chl17] Adam Chlipala. Formal reasoning about programs. *url: http://adam. chlipala. net/frap*, 2017.

[Chr13] David Raymond Christiansen. Bidirectional typing rules: A tutorial. Technical report, 2013.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, 1992.

[Coq96] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996.

[CPD14] Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns. In Zhong Shao, editor, *Programming Languages and Systems*, pages 87–106, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[CSW14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. *ACM SIGPLAN Notices*, 49(1):33–45, 2014.

[CTW21] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: A type theory with computational assumptions. In *ACM Symposium on Principles of Programming Languages*, 2021.

[DFFF11] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. *ACM SIGPLAN Notices*, 46(1):215–226, January 2011.

[DHL+14] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. In *The Coq Workshop*, 2014.

[DHT03] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In *International Conference on Theorem Proving in Higher Order Logics*, pages 188–203. Springer, 2003.

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5), May 2021.

[DTHF12] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 214–233, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[ETG19] Joseph Eremondi, Éric Tanter, and Ronald Garcia. Approximate normalization for gradual dependent types. *Proceedings of the ACM on Programming Languages*, 3, July 2019.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. Association for Computing Machinery.

[Fla06] Cormac Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages*, POPL '06, pages 245–256, New York, NY, USA, 2006. Association for Computing Machinery.

[GCT16] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *ACM Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. Association for Computing Machinery.

[Gri17] Radu Grigore. Java generics are turing complete. *SIGPLAN Notices*, 52(1):73–85, jan 2017.

[Hof97a] Martin Hofmann. *Extensional constructs in intensional type theory.* Springer Science & Business Media, 1997.

[Hof97b] Martin Hofmann. *Syntax and Semantics of Dependent Types*, pages 79–130. Publications of the Newton Institute. Cambridge University Press, 1997.

[HXB+19] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 411–424, New York, NY, USA, 2019. Association for Computing Machinery.

[JZSW10] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. *ACM SIGPLAN Notices*, 45(1):275–286, January 2010.

[KF09] Kenneth Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification*, PLPV '09, pages 27—38, New York, NY, USA, 2009. Association for Computing Machinery.

[KF10] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2), feb 2010.

[KSW20] Wen Kokke, Jeremy G. Siek, and Philip Wadler. Programming language foundations in agda. *Science of Computer Programming*, 194:102440, 2020.

[Lam18] Leonidas Lampropoulos. *Random Testing for Language Design*. PhD thesis, University of Pennsylvania, 2018.

[LB21] Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[LBMTT22] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. Gradualizing the calculus of inductive constructions. *ACM Transactions on Programming Languages and Systems*, 44(2), apr 2022.

[Lei86] Gottfried Wilhelm Leibniz. Discours de métaphysique, 1686.

[Lem19] Mark J Lemay. Understanding java usability by mining github repositories. In *9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[LF21] Mark Lemay and Qiancheng Fu, Qiancheng amd Fu. Gradual correctness: a dynamically bidirectional full-spectrum dependent type theory (extended abstract). 2021.

[LGWH+17] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 114–129, 2017.

[LHM+17] Mark Lemay, Wajih Ul Hassan, Thomas Moyer, Nabil Schear, and Warren Smith. Automated provenance analytics: A regular grammar based approach with applications in security. In *9th USENIX Workshop on the Theory and Practice of Provenance*, 2017.

[LHP19] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[LPP17] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[LT17] Nico Lehmann and Éric Tanter. Gradual refinement types. *ACM SIGPLAN Notices*, 52(1):775–788, January 2017.

[LT20] Yu-Yang Lin and Nikos Tzevelekos. Symbolic Execution Game Semantics. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. 1994.

[LZB20] Mark Lemay, Cheng Zhang, and William Blair. Developing a dependently typed language with runtime proof search (extended abstract). *Workshop on Type-Driven Development*, 2020.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Miq01] Alexandre Miquel. The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, pages 344–359, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[ML71] Per Martin-Löf. A theory of types. Technical report, University of Stockholm, 1971.

[ML72] Per Martin-Löf. An intuitionistic theory of types. Technical report, University of Stockholm, 1972.

[MM04] Conor Mcbride and James Mckinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[MU21] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[NM05] Aleksandar Nanevski and Greg Gregory Morrisett. Dependent type theory of stateful higher-order functions. 2005.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[NTHVH17] Phuc C Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *Journal of Functional Programming*, 27, 2017.

[OTMW04] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US.

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 328–345, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Pro13] The Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. 2013.

[PSH90] Erik Palmgren and Viggo Stoltenberg-Hansen. Domain interpretations of martin-löf's partial type theory. *Annals of Pure and Applied Logic*, 48(2):135–196, 1990.

[PT18] Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option. In *European Symposium on Programming*, pages 245–271. Springer, 2018.

[PT19] Pierre-Marie Pédrot and Nicolas Tabareau. The fire triangle: How to mix substitution, dependent elimination, and effects. *Proceedings of the ACM on Programming Languages*, 4(POPL), December 2019.

[Rei89] Mark B. Reinhold. Typechecking is undecidable when 'type' is a type. Technical report, 1989.

[Ros99] Andreas Rossberg. Undecidability of ocaml type checking, 1999.

[SCA+12] Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. *Mathematically Structured Functional Programming*, 76:112–162, 2012.

[Sjö15] Vilhelm Sjöberg. *A dependently typed language with nontermination*. PhD thesis, University of Pennsylvania, 2015.

[SJW16] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 228–242, New York, NY, USA, 2016. Association for Computing Machinery.

[Smi88] Jan M. Smith. The independence of peano's fourth axiom from martin-löf's type theory without universes. *The Journal of Symbolic Logic*, 53(3):840–845, 1988.

[SSK19] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: Reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 166–180, New York, NY, USA, 2019. Association for Computing Machinery.

[Str06] Thomas Streicher. *Domain-theoretic foundations of functional programming*. World Scientific Publishing Company, 2006.

[STS15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. Autosubst: Reasoning with de bruijn terms and parallel substitutions. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 359–374, Cham, 2015. Springer International Publishing.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

[SVCB15] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[SW15] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 369–382, 2015.

[Tak95] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, 1995.

[Vel03] Todd L Veldhuizen. C++ templates are turing complete. Technical report, Indiana University Computer Science, 2003.

[VJA18] Matthijs Vákár, Radha Jagadeesan, and Samson Abramsky. Game semantics for dependent types. *Information and Computation*, 261:401–431, 2018. ICALP 2015.

[VPJMa12] Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 341–352, New York, NY, USA, 2012. Association for Computing Machinery.

[Wad87] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. Association for Computing Machinery.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and*

Computer Architecture, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery.

[Wad15] Philip Wadler. A Complement to Blame. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 309–320, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[WF94] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WF09] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 1–16, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Xi07] Hongwei Xi. Dependent ml an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 03 2007. Copyright - 2006 Cambridge University Press; Last updated - 2010-06-08.

[YFD21] Shu-Hung You, Robert Bruce Findler, and Christos Dimoulas. Sound and complete concolic testing for higher-order functions. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 635–663, Cham, 2021. Springer International Publishing.

[ZMMW20] Jakub Zalewski, James McKinna, J. Garrett Morris, and Philip Wadler. λdb: Blame tracking at higher fidelity. In *Workshop on Gradual Typing*, WGT20, pages 171–192, New Orleans, January 2020. Association for Computing Machinery.

CURRICULUM VITAE