

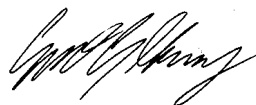
Project Report

Project Title:	Type-Checker Generation
Marking Scheme:	Software Development Based
Student Name:	Grant G Fleming
Registration Number:	201700435
Supervised By:	Dr Conor McBride
Second Marker:	Dr Clemens Kupke

Submitted for the Degree of B.Sc. in Computer Science 2020/2021

Except where explicitly stated all work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context.

Signed:



Grant G Fleming

Date:

March 28, 2021

I agree to this material being made available in whole or in part to benefit the education of future students.

Abstract

Static analysis is an important part of all modern compilers. In this project, we explore how much of the code used to implement type-checkers can be commonly abstracted across a family of type systems.

The aim is to create a generic type checker that is able to type-check code so long as the type system of the language is described to it.

We begin by performing a general type theory literature review: covering the simply typed lambda calculus [5], System F [19] and Hindley-Milner type theory [10, 16, 7] before finally exploring Intuitionistic type theory [13] (sometimes known as Martin L f type theory). We then review some select literature relevant to the implementation of various type-checker components and some relevant meta-theory.

After reviewing our research we will develop a domain-specific language (DSL) for describing type systems, paying special attention to not collect redundant information from the user.

We then use our DSL when we design and implement a type-checker-generator in Agda. This will take the form of a piece of software that receives, as its input, a specification file written in our previously developed DSL and a source code file in the language it describes. The software will then type-check the source code according to the type-system described in the specification.

We conclude by evaluating our DSL, software, and design and implementation process. Finally, we speculate as to how we might improve our software and conclude by outlining possible areas for future work.

Contents

I Background & Literature Review

1	What is a type system?	1
2	An overview of type theory	2
2.1	Background, lambda calculus and System F	2
2.2	Hindley-Milner	3
2.3	Intuitionistic type theory	4
3	Relevant related work	6
3.1	Contexts and type inference	6
3.2	Syntactic universes	6
3.3	Bidirectional dependent type theory	7

II Building A Type-Checker-Generator

4	Project plan	9
4.1	Project specification	9
4.1.1	Overview	9
4.1.2	Goal	9
4.1.3	Artefacts	9
4.1.4	Scope	10
4.1.5	Milestones	10
4.1.6	Acceptance criteria	10
4.2	Timeline	11
4.3	Documentation	11
5	A language for describing type systems	12
5.1	What needs to be described	12
5.2	What is supplied for free	12

5.3	The high-level structure of descriptions	12
5.4	Representing patterns	13
5.5	Representing expressions	14
5.6	Representing premises	15
5.7	Representing η -expansion rules	17
5.8	The DSL	17
6	Building a type checker generator	20
6.1	Implementation Overview	20
6.2	A core language	23
6.3	Thinnings	24
6.4	Substitution	25
6.5	Term substitution	26
6.6	Contexts	27
6.7	Openings	27
6.8	Patterns	27
6.9	Expressions	30
6.10	Premises	31
6.11	Typing rules	32
6.12	η rules	33
6.13	Semantics	34
6.14	Normalisation by evaluation	35
6.15	Checking types	37
6.16	Parser combinators	40
6.17	Parsing the DSL	41
6.18	Parsing the language	41
6.19	Putting it all together	42
7	Testing and verification	43
7.1	Type-led verification	43
7.2	Where type-led verification is insufficient	44
8	Evaluation	45
8.1	DSL evaluation	45
8.2	Software evaluation	46
8.2.1	Forced explicit type information	47
8.2.2	Limited DSL validation	47

8.2.3	Rule matching depends on rule declaration order	47
8.2.4	Missed opportunities for mathematical structure	48
8.2.5	Single universe limitation	48
8.2.6	Code quality	48
8.2.7	Documentation	49
8.3	Personal performance evaluation	49
9	Conclusion	51
9.1	Summary	51
9.2	Future work	51
9.3	Conclusion	52
9.4	Acknowledgements	52
	Appendices	53
.1	Typing rules	54
.1.1	System F	54
.1.2	Hindley-Milner	54
.2	Example specifications	56
.2.1	Simply typed lambda calculus with product types	56
.2.2	System-F-like language	57
.2.3	Lambda calculus variation with dependent types	58
.3	Parser combinators	59
.4	A selection of testing code	63
.4.1	Description of STLC	63
.4.2	Beta reduction and normalisation	67
.4.3	Eta expansion	70
.4.4	Type checking STLC	71
.5	Lattice meet/join algorithms	73
	Bibliography	74

Part I

Background & Literature Review

Chapter 1

What is a type system?

From a computer science perspective, a type is a piece of information that we attach to terms or other constructs of a programming language so that we might verify the validity of their use or otherwise reason about the meaning of the program. The classic example of checking validity is attaching a type to a function that identifies the type of the term it expects as input, and thus we are able to later check that any input supplied conforms to this type.

Type systems use the types assigned to terms to help the programmer avoid certain classes of errors by checking that terms of various types are combined in ways that are valid according to their types and the rules of the type system. Exactly when these checks take place can vary, sometimes occurring before compilation, known as static type checking, and sometimes occurring after compilation during the running of the program, known as dynamic type checking.

We often talk about type constructors as parameterised entities that we use to construct types. When these constructors require no parameters, they are somewhat synonymous with types. In many languages we can supply other types as parameters to type constructors, allowing us to represent data such as typed lists. When we are allowed to supply arbitrary terms in the language as parameters to type constructors, the language is said to be dependently typed as it depends on values in the language.

The features offered by type systems vary greatly. All will involve some incarnation of type checking however, they can often include more advanced features such as type inference, higher-order functions, algebraic data types, various flavours of polymorphism and termination checking [1]. In some applications, the type system takes an interactive role in the production of code as is the case with some theorem provers and dependently typed programming languages.

Type systems are often described in the literature by a collection of inductive rules that describe aspects of the system such as when we can assign some type or when we might consider types equivalent. In the case of dependent types, we also need to provide some semantics so that we might make sense of arbitrary terms in the type. In this case, the type checking process might need to appeal to reduction to perform some check.

Chapter 2

An overview of type theory

2.1 Background, lambda calculus and System F

Type systems, as with so many concepts in computer science, are a concept borrowed from the study of logic. Their initial purpose was to resolve certain inconsistencies in an underlying logic.

Although originally proposed by Bertrand Russell in 1902 to resolve a paradox he had discovered in a formalization of Gottlob Frege's naive set theory in 1901 [3], much introductory material on type theory from a computer science perspective begins with Alonzo Church and his simply typed lambda calculus [5].

Like Russell, Church was searching for a way to make his previously defined system of logic, the lambda calculus, consistent. In 1940 he published his seminal paper that outlined a type system and added extra criteria to which a term must conform to be considered well-formed - it must be typeable.

A concrete, if informal, summation of Church's work here is that in this new system, if a lambda abstraction is applied to a term, the type of the term must be the same as the expected type of the binder in the abstraction. As a consequence, previously well-formed terms which caused inconsistencies in the logic, such as $(\lambda x.xx)(\lambda x.xx)$ were no longer well-formed under the new system.

As the field of computer science expanded rapidly in the late 1960s and 1970s, prominent logicians and computer scientists started to discover the notion of parametric polymorphism. It was noted that some functions had common behaviour over differing types and that the behaviour of these functions did not depend on the types themselves. This led to redundant definitions, such as having to define a function with the same behaviour multiple times, once for each type you wish to operate over. Under type systems akin to the simply typed lambda calculus, the *id* function given by the term $\lambda x_{\mathbb{N}}.x$ is only ever applicable to natural numbers. There was an immense benefit in being able to define functions such as *id* once and have them operate over any type.

One such solution to this problem came to be known as System F or polymorphic lambda calculus. This system was independently discovered by Jean-Yves Girard and John Reynolds in 1972 and 1974 respectively [8, 19].

In lambda calculus, a single binder λ is used to bind variables that range over values. In the simply typed lambda calculus, a lambda term of the form $\lambda x_{\alpha}.M_{\beta}$ (where x_{α} binds

variables x over a type α and M_β is some term of type β) has type $\alpha \rightarrow \beta$ by modern notation. System F introduces a new binder Λ that is used to bind variables that range over types. In this system, terms of the form $\Lambda t.M$ denotes a function that takes, as its first argument, some type and returns a term with all references to t replaced by that type. This function has type $\Delta t.M^t$ where M^t is the type of M .

Under this system, we may write the *id* function as $\Lambda t.\lambda x_t.x$ and thus have it operate over any type t , so long as we supply it.

One disadvantage of this system is that terms become verbose because of their explicit references to type information. Another is that explicit quantification over types can quickly become unwieldy when writing more complex types for polymorphic functions.

2.2 Hindley-Milner

An alternative approach to parametric polymorphism was developed in the late 60s with the Hindley-Milner type system which grew to become the basis of many functional languages such as ML and Haskell.

Hindley introduced a system of *type-schemes* [10] (types that may contain quantified type variables) and defined a type as being a type-scheme that contains no such variables. Milner referred to these concepts as polytypes and monotypes respectively [16].

A given term may have any number of type-schemes, some more general than others, allowing us to commit to stating more or less about our values as the situation requires. Hindley then presents the idea of a *principle type scheme* (p.t.s) where all possible type-schemes for a given term are instances of the p.t.s. (that is, can be constructed by consistent substitution for quantified variables in the p.t.s.).

He then proves that any term for which you can deduce a type scheme, has a p.t.s. It is always possible to work out what the p.t.s. is up to *trivial instances* and he introduces an algorithm to accomplish this type inference - Algorithm W.

As a consequence of this system, there is no need for us to annotate the definitions in our code with type information since we can always infer the p.t.s (most general type-scheme) of our terms. This can result in cleaner looking code in some circumstances however, there is an argument that providing type information explicitly in the syntax of the language serves as important documentation.

The type inference algorithm that is detailed in the Hindley-Milner system is also instrumental in the way types are checked. In a system like System F, whenever a polymorphic function is applied, it must first be provided with the type parameter before being provided with the value parameter so that the value may be type-checked. With Hindley-Milner, as a direct consequence of its type inference features, this is not necessary. Instead, the most general type of the function can be inferred and it is then possible to check that the type of the supplied value parameter is some instance of the input type of the function.

The classic example of such systems is the creation of lists in the standard *cons/[]* way. In a language with a type system akin to System F, this may look something like:

```
(cons Number 1 (cons Number 9 (cons Number 6 [])))
```

whereas if we can infer the most general type of *cons* and subsequently verify that the

supplied input is some instance of this type as in Hindley-Milner, the same expression might be written as:

(cons 1 (cons 9 (cons 6 [])))

We can see this difference reflected in the typing rules of the systems by way of the following example on *id*:

Hindley-Milner	System F
$id : \forall \alpha . \alpha \rightarrow \alpha$	$id : \Delta \alpha . \alpha \rightarrow \alpha$
$id = \lambda x . x$	$id = \Lambda t . \lambda x_t . x$

The type in the Hindley-Milner system can be proven by:

$$\frac{\frac{\frac{x : \alpha \in x : \alpha}{x : \alpha \vdash x : \alpha} \text{ var}}{\vdash \lambda x . x : \alpha \rightarrow \alpha} \text{ abs} \quad \alpha \notin \text{free}(\epsilon)}{\vdash \lambda x . x : \forall \alpha . \alpha \rightarrow \alpha} \text{ gen}$$

The equivalent type in System F can be proven by:

$$\frac{\frac{\frac{x : \alpha \in x : \alpha}{x : \alpha \vdash x : \alpha} \text{ var}}{\vdash \lambda x_t . x : \alpha \rightarrow \alpha} \text{ abs}}{\vdash \Lambda t . \lambda x_t . x : \Delta \alpha . \alpha \rightarrow \alpha} \Delta\text{-abs}$$

In system F, this deduces the only type for $\Lambda t . \lambda x_t . x$ (up to the renaming of bound variables), however, note that in Hindley-Milner we may also deduce other type (schemes) for $\lambda x . x$:

$$\frac{\frac{\frac{x : \alpha \rightarrow \beta \in x : \alpha \rightarrow \beta}{x : \alpha \rightarrow \beta \vdash x : \alpha \rightarrow \beta} \text{ var}}{\vdash \lambda x . x : (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)} \text{ abs} \quad \beta \notin \text{free}(\epsilon)}{\vdash \lambda x . x : \forall \beta . (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)} \text{ gen} \quad \alpha \notin \text{free}(\epsilon)}{\vdash \lambda x . x : \forall \alpha . \forall \beta . (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)} \text{ gen}$$

If you wanted a term with an equivalent type in System F you would need to create a new term $\Lambda t_1 . \Lambda t_2 . \lambda x_{t_1 \rightarrow t_2} . x$

A full list of the typing rules of both type systems is available in appendices .1.1 and .1.2.

2.3 Intuitionistic type theory

Intuitionistic type theory was devised by Per Martin-Löf [13] originally in the 1970s, although several variations have been devised since.

Martin-Löf was motivated, in part, by what we now call the Curry-Howard isomorphism: the idea that statements in logic can be thought of as types, and a proof of a logical statement can be thought of as a program (consequently, simplification of proofs correspond execution of programs) [20]. He noted that despite this isomorphism, computer scientists often developed their own languages and type systems instead of using existing logics and so he designed his theory of Intuitionistic types to have practical uses as both a logic and programming language.

While relating function types to implication was already a well known consequence of the Curry-Howard isomorphism, Martin-Löf was the first person to extend the notion to predicate logic by having his types depend on values, using dependent functions and dependent pairs to encode universal and existential quantification respectively.

It was this introduction of ‘dependent types’ that made this theory different from previous type theories and allowed the development of dependently typed languages with flexible and powerful type systems such as Agda [17].

Although Martin-Löf describes many types in this system, there are a few that are of particular importance which we will briefly describe here:

- Types of the form $(\prod x \in A)B(x)$ describe dependent functions, where the type $B(x)$ of the co-domain depends on the *value* x of the domain.
- Types of the form $(\sum x \in A)B(x)$ describe dependent pairs, where the type $B(x)$ of the second element of the pair depends on the *value* x of the first element.
- Types of the form $A + B$ represent the disjoint union of two types A and B .
- Types of the form \mathbb{N}_n describe types with finite numbers of values. For instance, we could take \mathbb{N}_1 to be unit, \mathbb{N}_2 to be booleans etc.
- Types of the form $I(A, a, b)$ encode propositional equality where values of this type represent proofs that $a \equiv b \in A$.
- W-types have the form $(Wx \in A)B(x)$ and represent *wellorderings* (a somewhat more intuitive explanation is that they represent well-founded trees). This type allows us to define types with complex structures that are inductive in nature. Some of the types that Martin-Löf includes in his type-theory, such as that of Lists or \mathbb{N} , can be defined as W-types.
- Universes are the type of types, allowing the transfinite types needed to talk about a "sets of sets". Martin-Löf recognised the need for such types when reasoning about certain areas of maths such as category theory.

One of the consequences of this powerful type system, is that unlike Hindley-Milner, we can no longer infer the most general type of any term.

Take, for example, a value of some dependent pair $(2, [a :: b :: nil])$ The type of the second element could dependent on the first, and the B in the type $(\sum x \in A)B(x)$ might correspond to some function $\lambda n \rightarrow Vec\ X\ n$, and yet it might not depend on the first argument at all and B could simply correspond to $\lambda n \rightarrow Vec\ X\ 2$. There is no way for us to definitively deduce B from the value alone. To give a more precise definition of the problem we might state that we cannot employ Hindley-Milner style type inference in a dependently typed setting as functions are not always uniquely invertible.

Chapter 3

Relevant related work

In the previous chapter, we explored material designed to give a general background of type-theory. Here we will explore some more modern works which will help us in the construction of a type-checker-generator.

3.1 Contexts and type inference

Gundry, McBride and McKinna present research detailing a novel way of implementing unification and type-inference [9]. An area of particular interest is in their approach to the *context* - traditionally used to track the types of term variables.

In this work, the authors propose a system of explicitly tracking type-variables in the context, even before they are bound to a type-scheme. As well as allocating types to these variables, the unification algorithm is also able to pull these type-variables leftward to scope them appropriately and resolve dependencies, while it tries to solve unification constraints.

This leads to an interesting consequence in their approach to generalisation in let expressions. Now that type variables can exist in the context, we can instantiate a type scheme by introducing a new type variable to the context, and removing the necessary \forall quantifier in the scheme. Consequently, to generalise, we can remove these type-variables from the context and introduce an appropriate \forall quantifier. The authors explicitly place a third element into the context, a marker that delimits generalisation to an appropriate scope.

3.2 Syntactic universes

The concept of a syntactic universe was first introduced to the author by his supervisor, Dr Conor McBride, and further general material on universe construction was obtained when it was covered as a topic in an Advanced Functional Programming class, taught by Dr Fredrik Nordvall Forsberg.

Universe constructions are a way of programming generically by computing over descriptions of data, rather than the data itself. Thus, descriptions are designed carefully to capture the necessary properties of things in the universe and computations are created that operate on these descriptions. As well as creating descriptions it is also possible to retrieve the *meaning* of descriptions - to get ahold of the actual entity they are describing.

We learn more about this concept when we explore work detailing how we might build a universe of syntaxes that are both scope and type-safe and abstractly define common semantics such as substitution (that is defined in this paper to work over arbitrary descriptions) and type-checking (that is defined in this paper to work over a specific description only) [2].

The authors first formalise the concept of well-scoped entities using a family of types, and detail useful combinators for handling scope before providing the required machinery for representing environments. They then introduce the idea of a general notion of semantics for a language and demonstrate its use by defining several operations in terms of this generic semantics.

After exposing the reader to previous work in creating a universe of data types [4] they then go on to modify the *descriptions* defined in this work to instead describe a universe of scope safe and well-kinded syntaxes.

With this concept of descriptions in place, the previous notion of semantics is replaced by a more generic notion which is not syntax-specific and can be used to represent a semantics for any language that we might *describe*. The authors are able to show true syntax-generic programming by defining functionality as semantics over a generic description.

Later in this paper, the authors broach the subject of type-checking and elaboration. In particular, the section on elaboration provides a concrete example of creating an elaborator as semantics over a description of bidirectional lambda calculus.

3.3 Bidirectional dependent type theory

While a bidirectional approach typing is not new [18], formalisation of the metatheory is [14] and we will use this formalisation throughout our project.

The work in this paper formalizes the metatheory for a whole class of bidirectional dependently typed languages. It demonstrates how desirable properties of type systems may be enforced by construction, outlines pitfalls that might be encountered when combining type checking with type synthesis, and provides an object-level syntax split into carefully chosen grammatical classes that we might use to represent arbitrary syntax.

It then emphasises the importance of precisely tracking information flow as it defines judgement forms, patterns, expressions, and typing rules before determining a method for approaching the *schematic variables* that appear in *formal judgements* in the typing rules.

It is from this work we notice that by paying careful attention to the construction of the individual components used to describe a type theory, we may minimize the amount of information the user is required to give, and better validate the information that they do give to ensure the resulting system has desirable properties. It also provides a useful framework for approaching this project as a whole.

Part II

Building A Type-Checker-Generator

Chapter 4

Project plan

4.1 Project specification

4.1.1 Overview

Writing type-checkers for compilers is a key task in the development and evolution of programming languages.

Despite the huge variety in the kinds of languages in existence, type-systems can often be described using the same kinds of basic elements such as typing rules. The descriptions have become standardised enough for us to recognise common syntax for describing these aspects in literature.

This project seeks to capitalise on this by abstracting the common elements and asking the user to provide only what is specific to the type system they seek to represent. Given any description of a type-system in a specified format, a type-checker-generator should be able to generate a type-checker for the type system and use it to type-check provided source code.

4.1.2 Goal

To design a domain-specific language (DSL) capable of describing type systems and to write a piece of software that, given one of these descriptions, can type-check code in the language it describes.

4.1.3 Artefacts

At a minimum, the finished project will consist of:

- A type-checker generator as a standalone binary
- At least two example specifications in the DSL
- At least two source code examples for each specification, one that type-checks successfully and one that does not
- The final report

4.1.4 Scope

In-Scope	Not In-Scope
Notification on whether the type-checking succeeds or fails	Detailed feedback from the type-checking process
Notification on whether or not the specification file is successfully parsed	Detailed feedback on the parsing of the specification file
Example specifications and checkable code for small 'toy' languages	Example specifications and code for large 'real world' languages
Single usable feature of 'running' the generator on a specification and piece of source code	Quality of life/usability features or other niceties

4.1.5 Milestones

1. A DSL that we can use to write *specifications* of type systems.
2. A type-checker able to check types for a generic syntax using a *fixed* set of typing rules where the checkable source code is supplied hard-coded.
3. A type-checker able to check types when full type-system specifications and checkable source code is hard-coded.
4. A type-checker that checks types, reading the specification and source code from files.


4.1.6 Acceptance criteria

1. The software can be run from the command-line by giving a simple command such as:

`TypeCheck < specification-file > < source-code-file >`
2. The software either succeeds with a simple indication message or fails with one of two kinds of error:
 - Failure to parse specification
 - Failure to successfully type-check source code
3. The software passes a fixed set of test cases (specifications and source codes) succeeding, failing, and showing appropriate errors as detailed in the test cases.
4. The software can generate type-checkers capable of checking both non-dependent and dependent types.

4.2 Timeline

The following timeline assigns provisional deadlines to key stages in the project. We will use this information later when we retroactively evaluate our performance.

- 
- 16/11/2020 - Complete initial readings on type theory.
 - 30/11/2020 - Complete specific readings relating to the design and implementation of various type systems and components thereof.
 - 07/12/2020 - Identify all elements needed to fully and unambiguously describe a type system.
 - 28/12/2020 - Design a DSL, formalising a syntax for expressing these elements.
 - 04/01/2021 - Produce sample specifications for testing using the DSL.
 - 01/03/2021 - Design and implement a type-checker-generator.
 - 15/03/2021 - Verify and test the implementation.
 - 22/03/2021 - Conduct a qualitative evaluation of the DSL and resulting software.
-

4.3 Documentation

The process of creating documentation for the project is intertwined with the process of developing the code. We plan to create the source code in literate Agda files, where source files are latex files interspersed with areas identified as code.

By developing the code in this manner, we somewhat blur the lines between implementation, documentation and reporting as we develop aspects of all three simultaneously, resulting in code that is self-documented. We do not consider a user-guide necessary since running the software consists of executing a binary and supplying paths to some specification file and some source code file. We will ensure the software provides appropriate feedback if such files are not identified.

Chapter 5

A language for describing type systems

5.1 What needs to be described

To build a type checker for a language, the user must have the means to provide the following information:

- Types
- Language constructs for each type, defining the 'shape' of all values of the associated type in weak head normal form
- A type-checking rule for each new language construct
- Eliminators
- Type synthesis rules for each elimination
- β -reduction rules for each elimination
- Optional η -expansion for each type

5.2 What is supplied for free

We provide means to describe a class of languages where all types exist in a single universe. Although we could allow the user to provide some name for the universe, we opt not to for simplicity and instead mandate the construction "set" to mean the universe.

When a user describes types, they are not required to specify that they exist in the universe. In a single-universe type system, there is nowhere else they could live and so we give this functionality for free in the resulting software.

5.3 The high-level structure of descriptions

We say that the structure of a description of a type-system is as follows:

- One or more type descriptions where each is:
 - type pattern
 - premises for determining it is a type
 - zero or more elimination descriptions where each is:
 - * eliminator pattern
 - * premises for checking the type of the eliminator
 - * type expression of the whole elimination
 - zero or more canonical forms where each is:
 - * value pattern
 - * premises for checking the type of the value
 - * β -reduction expression for every elimination described
 - zero or one η -expansion rules

We place additional conditions on when we can supply an η -rule (which we will discuss further in a future section), however, for now we focus on the important concepts of patterns and expressions as these are critical to describing and building terms in our DSL.

5.4 Representing patterns

In the high-level structure, we make frequent reference to patterns. This is a concept we borrow from Conor McBride's previous work in formalising the meta-theory of bi-directional dependent type systems [14]. Patterns are tools we use to describe when arbitrary code matches a certain structure.

To define a pattern we require meta-level variables to represent arbitrary subterms in a pattern. We adopt the convention that meta-level variables are always uppercase and so when we speak of identifiers in this section, we are referring to strings of uppercase alphabetic characters. Newly bound variables are denoted by an identifier postfixed with a period and meta-level term variables are denoted by an identifier in isolation. All meta-level variables must be separated from the rest of the pattern by whitespace. All other characters are assumed to be literals and are parsed as atoms except for parenthesis. We may, for example, describe a pattern representing lambda terms as follows:

$$\backslash X. - > BODY$$

Where " \backslash " and " $->$ " are interpreted as atoms, " $X.$ " is interpreted as a newly bound variable and " $BODY$ " represents an arbitrary sub-term. The scope of the newly bound variable is assumed to extend rightward across the remainder of the term, if this is not desirable then we may manipulate the structure of the tree with parenthesis as one might expect, limiting the scope as in the following example where " X " is not in scope in " $ANOTHERSUBTERM$ ".

$$\backslash (X. - > BODY) \text{ and } ANOTHERSUBTERM$$

5.5 Representing expressions

An expression introduces a way for users to build terms from patterns that we decide are trusted. We find this concept in the same meta-theory that gave us our patterns [14].

In defining an expression, we define atoms of the expression in the same way that we did for patterns. We may place multiple elements side by side separated by whitespace (again in the same way that we did for our patterns) and use the same encoding for variables, newly bound variables are encoded with an identifier postfixed with a period. We may refer to bound variables by giving the appropriate identifier *prefixed* with a period as in the following example that constructs the identity function:

$$\backslash X. - > \{.X\}$$

In this case, we require that the bound variable reference is surrounded by curly braces because we are placing a computation in a place that expects a construction. We will consider the meaning of this later.

Since we require no meta-level sub-expression variables, we assume that any identifier standing alone and un-post/prefixed is intended to refer to a meta-level term variable of some pattern that is in-scope. When this expression is used in some concrete scenario, all the in-scope patterns will have been matched against terms. These variables are then used to pull out subterms that we use to construct the new term. The following example demonstrates the construction of such an expression when we assume the pattern to the left of the \Rightarrow is in-scope:

$$L \text{ and } R \Rightarrow \backslash X. - > L \text{ and } \{.X\}$$

We now consider the case where we refer to a sub-term that exists under some number of binders in the pattern. If we wish to use such a term then we must provide a substitution for all of the free variables in the subterm and so variables in an expression can optionally be followed by a "/" and some substitution. A complete substitution for all free-variables is mandated in the DSL to avoid unintended variable capture and the introduction of free variables as in the following case that may occur if we did not perform this check:

$$\backslash X. \backslash Y. - > M \quad \Rightarrow \quad \backslash X. - > \{.X\} \text{ and } M$$

We must also devise a syntax to describe substitutions themselves. We do so by giving substitutions as a list surrounded by square brackets, each substitution is separated by a comma and is given as an expression so long as that expression represents a term with a synthesizable type (a computation). When we later talk about how we build our type-checker we will discuss this further, but for now the following justification should be sufficient. We may synthesize the type of a free variable (by looking up its type in the context) and so if we are to substitute this variable for another term we should make sure that it is possible to synthesize its type also incase this is a property that is needed where the variable was used.

Substitutions are given in order according to their de Bruijn index with the "least local" variable leftmost in the list and the most local variable rightmost. This list must also begin with a comma. We do not supply an empty list in the case that a meta-term-variable does not contain free variables, we simply omit it. The following example shows how we use

substitutions in this way by repeating the previous example, except that now we make the captured "X" explicit and remove the free "Y":

$$\backslash X. \backslash Y. - > M \quad \Rightarrow \quad \backslash X. - > \{.X\} \text{ and } M/[, .X, .X]$$

In the substitution, we do not require that the ".X" bound variable references are surrounded by curly braces, as they are not being embedded in some construction. So far we have given the various ways in which we might build expressions that represent constructions. We have shown how we may reference bound variables, how these are computations and how we might embed a computation in a construction by surrounding it in curly braces.

There are two other kinds of computations we can encode in our expressions. We may write some expression representing a term followed by a colon and another expression representing its type (both as constructions) to encode type annotation information. We might encode elimination by providing an expression representing some computation, followed by some whitespace and a construction.

We give some examples of expressions below giving the in-scope patterns to the left of the "⇒" separated by commas and the expression to the right of it. The patterns are only given in this manner to give context for the benefit of the reader, the ⇒ notation is not part of our DSL.

$$\begin{aligned} A \times B , L \text{ and } R , \backslash X. - > \backslash Y. - > M & \Rightarrow \quad \backslash Z. - > M/[, (L : A), .Z] \\ TY , TM , E & \Rightarrow \quad (TM : TY) E \\ A , \backslash X. - > M & \Rightarrow \quad \backslash X. - > ((M/[, .X] : A) \text{ atm}) \end{aligned}$$

It is important to note the subtleties in scope when constructing expressions. In the last example, the "X." in the expression does not shadow the "X." in the pattern, they exist in different scopes. The "X." in the pattern technically does not exist at all, there is no way for us to reference it, either in the pattern itself (where variables do not exist) or in an expression where it is not in scope. It is merely a mental aid to construction.

5.6 Representing premises

As with patterns and expressions, our treatment of premise and premise chains in this section is heavily influenced by the same source [14].

There are four kinds of premise representable in the DSL. Premises are listed together one after another, to give the conditions under which we may make some judgement. We will first detail each type of premise before discussing what it means to chain the premise together. We will later detail what is considered in-scope when defining an expression as part of a premise.

The type premise is written by stating the word "type" followed by the identifier corresponding to some meta-level term variable in scope. This premise states the condition that the syntax referred to by some meta-level variable must represent a type.

$$\text{type } \text{SOMEMETAVAR}$$

The type check premise is written as some construction expression, followed by "<-" and some identifier corresponding to a meta-level term variable in scope. This premise states the condition that the syntax referred to by some meta-level variable must be of a particular type.

$$\text{someexpression} < - \text{SOMEMETAVAR}$$

The equivalence premise is written as a construction expression followed by "=" and another construction expression. This premise states the conditions that two pieces of syntax should be equivalent.

$$\text{someexpression} = \text{anotherexpression}$$

The context extension premise is written as some identifier, followed by ":" and a computation expression followed by "|-" and another premise. I.e. it allows us to present an arbitrary premise in an extended context.

$$X : \text{someexpression} \mid - \text{some} < - \text{OTHERPREMISE}$$

Here we note that the expression must be a computation in case we use it in a place where type synthesis is required. The context extension premise is somewhat different. Its use is mandated if we wish to present a premise talking about some meta-level term variable that exists under a binder in the original pattern. That is, we are only allowed to present a term with free variables along with a context expanded to type them. The name for the free variables is assigned in the context expansion section of the premise.

A chain of premises is always considered in the context of some pattern, the subject. A chain of premises also begins with some trusted pattern, the nature of this pattern depends on where we are using the chain of premises and will be detailed later. The subject contains meta-level term variables which we do not yet trust, but seek to trust, and the trusted pattern contains meta-level term variables which we trust already. The purpose of a chain of premises is to obtain trust in the subject by describing how to obtain trust in each of the meta-level term variables in it. There are two ways that we can discharge such a variable as trusted: we can state the type of the term variable with a type check premise, or we can state that the term variable is a type with the type premise. While we may use context extension and equivalence checking at will, neither of these premises discharge anything from what remains to be trusted in the subject.

At the start of a premise chain, only what is initially given as trusted is in-scope (meaning that we may refer to its meta-level term variables when building expressions). As we process each link in the chain, we may potentially move something from the subject (to be trusted) to the input (which we trust) bringing it in-scope from the next link in the chain until the end. A chain may only end if there remains nothing in the subject to be trusted (something we enforce by construction in our software).

The initial scope of a premise chain, that is the pattern where we may reference meta term variables without establishing any trust, differs depending on the purpose for which we are building the chain. If we are supplying the premises for a new type, then we supply nothing as initially trusted and the pattern representing the new type is the subject that we seek to trust. If we are supplying the premises for checking some language construction is of a certain type, then we trust the type as input, and we seek to trust the new data constructor pattern that

we take as a subject. The type of the target is also in scope when we give the premises that seek to trust some eliminator we supply in an elimination typing rule.

5.7 Representing η -expansion rules

For us to perform η -expansion, there are two pieces of information that we must have.

Firstly, we must know the top-level data constructor of a newly η -expanded term. For this reason, we restrict η -expansion to terms that contain only a single data constructor.

Secondly, for each argument to the data constructor, we must know the eliminator which we might use such that when we eliminate the original term with the eliminator, we get the term that is supplied at that place.

We can therefore give enough information to facilitate η -expansion by choosing a pattern representing some value, and giving an environment for the pattern, where an environment is structurally identical except that it has a term everywhere the original pattern had a term variable.

For instance, given the product type $A \times B$ that describes values (X, Y) and has eliminators fst and snd then we might give (fst, snd) to inform the construction of an η -expansion rule. Similarly, for a function type $A \rightarrow B$ which describes values $\lambda X. \rightarrow M$ we might give $\lambda Y. \rightarrow Y$ to inform the construction of an η -expansion rule which when applied to some target T would result in an η -expanded term $\lambda Y. \rightarrow (T Y)$.

5.8 The DSL

When designing the DSL there were three main problems that we wished to address. The first is that type-theory is an inherently jargon-heavy topic and so here we attempt to keep our DSL as jargon-free as possible, in particular favouring syntax that can be read linearly. The challenge in describing a type-system should be constructing the appropriate expressions, not understanding how the high-level syntax of the DSL is laid out.

The second issue is that of brevity. For a given type, it is standard to supply many rules to check types, define reduction, define η -expansion and more. Typically much information is repeated across a set of such rules. It is for this reason we propose a hierarchical structure in our DSL. The top-level element is a "type" that is described by a certain pattern. Any rules pertaining to this type sit inside this top-level element and can access the same pattern. There are similar hierarchical gains to be had elsewhere in the DSL resulting in a concise language that we believe is still approachable to read and understand.

The third and final problem that we seek to address is that the freedom afforded with most methods of describing type-systems make dependencies non-obvious. For instance, it is easy to mistakenly convince yourself that the type of some elimination depends on the syntax of the target in some way, this is not the case. Dependencies are made clear in our DSL using a combination of hierarchical structure and a carefully chosen order in which we require the information to be given.

As previously explained, the top-level definition in our DSL associates a pattern with a type. If premises are required (that is to say, if the pattern contains meta-level term variables that we must trust) then we add an "if:" clause, and then we list the premises, with each

premise on a line of its own. Inside some type description, we may give values of the type, again these are given as patterns as described in 5.4.

A description of a simple base type with a single value might look as follows:

```
type: alpha
value: a
```

We will now outline how we could build more complex types by describing functions in the simply typed lambda calculus. We will present the description in four stages. The first stage is defining the syntax of the type itself. Since our type contains meta-variables then we must provide a means to trust these, and so premises are required for this type.

```
type: A -> B
if:
  type A
  type B
```

This is an area in which we evidence the jargon-reduction and readability goals of our DSL design.

In the second stage, we supply the required information to type eliminations. Since this is not dependent on the specifics of the actual values of a type, we give the required information for this process here before we even define what the values could be making this non-dependency far more clear. We define the eliminator by giving some pattern, trusting the pattern and finally showing how we might build the final type of the elimination with an expression.

```
type: A -> B
if:
  type A
  type B
eliminated-by: E
if:
  (A) <- E
resulting-in-type: B
```

We may give any number of eliminators here so long as when we later give values, we give the appropriate number of reductions, one for each of the eliminations described here, and we give them in the same order. Such an example can be seen in the description of product types in appendix .2.1.

In the third stage, we define what patterns of syntax correspond to values of this type and provide the necessary premises to identify when this is the case. In our example we are forced to trust a single meta-variable so we need to supply at least one premise to achieve this.


```

type: A -> B
  if:
    type A
    type B
  eliminated-by: E
    if:
      (A) <- E
    resulting-in-type: B
value: \ X. -> M
  if:
    X : (A) |- (B) <- M
  reduces-to: M/[ , E:A]

```

Note that since we gave one eliminator earlier, we are forced to give exactly one reduction inside the value description which we do by supplying an expression that substitutes for the free variable in M.

Again we emphasise the continued prioritisation of a jargon-free linear reading experience and the use of hierarchy to make dependency explicit.

Finally, we give the required information required to construct the η -expansion rules as described in 5.7 resulting in a complete description of simple non-dependent function types.

```

type: A -> B
  if:
    type A
    type B
  eliminated-by: E
    if:
      (A) <- E
    resulting-in-type: B
value: \ X. -> M
  if:
    X : (A) |- (B) <- M
  reduces-to: M/[ , E:A]
  expanded-by: \ Y. -> Y

```

We have given this description in four stages intentionally to show that at each stage we had a valid, parsable description of a type. We allow types with no values as was shown the first stage. We allow the elimination typing rule to be supplied even if there are no values as was shown in the second. We then allow descriptions of the values of the type in stage three, and finally, we allow the addition of η -expansion information in the last stage. The resulting DSL is quite modular and this makes definitions concise where all of the features are not required, as can be seen in the base type *alpha* we introduced initially. This was an intentional decision that we made to address the problems with brevity that we identified earlier in this section.

A final point to make is that of whitespace. Indentation is irrelevant in our DSL to allow maximum flexibility. We choose to indent our descriptions to emphasise the structure of the language. There are some cases where new lines are required, after a keyword (something ending with ":") or if the keyword is identifying some kind of entity described by a pattern then the new line must follow the pattern. Also, as previously mentioned, when listing premises each one must appear on its own line.

Chapter 6

Building a type checker generator

Now that a user has the means of describing a type system, we will discuss how the generic type-checking software is constructed.

We will describe key definitions and types to communicate a high-level understanding of how the software works, however, we will often omit implementation details in areas where the code is less important, tedious or verbose.

The general tactic for type-checking some generic syntax will be to parse the required typing, β and η rules from the descriptions provided using our DSL. Using this information we can then parse user source code and represent it in a fairly language-agnostic internal object syntax before using the rules to type-check our internal syntax.

We will begin by describing the internal syntax, talk about how we might represent the various tools and rules that we need to type check this syntax and define the type checking process before briefly covering our approach to parsing this information from the user-supplied input files. We purposely keep our parsing conversations brief to prioritize the presentation of challenges in other areas.

Our approach in this section is heavily influenced by the work of Dr Conor McBride in *The types who say 'ni'* [14]. Although the code we provide is strictly our own in all cases, there are some areas where this code is a near-direct translation of the mathematics in this paper. As we point out in the previous chapter: patterns, expressions and the treatment of premises are all concepts that we take from this work. In addition, our internal syntax is very like that described in the paper and we use the same approach for thinnings, substitution, contexts and much of our treatment of scope.

6.1 Implementation Overview

When we provide implementation details in this chapter, we must provide this information in a 'bottom-up' fashion so that the reader has the required knowledge to make sense of the later code examples. To provide context to the reader in the earlier parts of this chapter we will now provide a high-level description of what we seek to accomplish.

Our software operates over an internal syntax that is split into two grammatical classes, constructions and computations. Constructions have their types checked, whereas computations have their types synthesised. The overall aim is to produce two functions, one to check

types and the other to infer types.

To check and infer types in some arbitrary type system, we need access to information that describes the system: typing rules, β -rules and η -rules. These concepts are represented by types in our implementation.

We will find that these rules are all constructed from the same basic building blocks. Since we closely follow the approach in [14] then we require two distinct forms for describing syntax in rules. We require the ability to match patterns of syntax as information flows into a rule, and so we provide patterns for this purpose. We also require the ability to build terms conditionally and define the information that flows out of a rule and so we define expressions to accomplish this. These concepts are then used together since expressions are formed in the presence of some pattern that defines a trusted input that we may use when building a term. This allows us to create synthesis rules as functions from input patterns to output expressions.

By separating these concerns we can ensure differing characteristics in each case. Patterns cannot encode computation and further computation in some term cannot unmatch it from a pattern. In contrast, we can build computation freely using expressions.

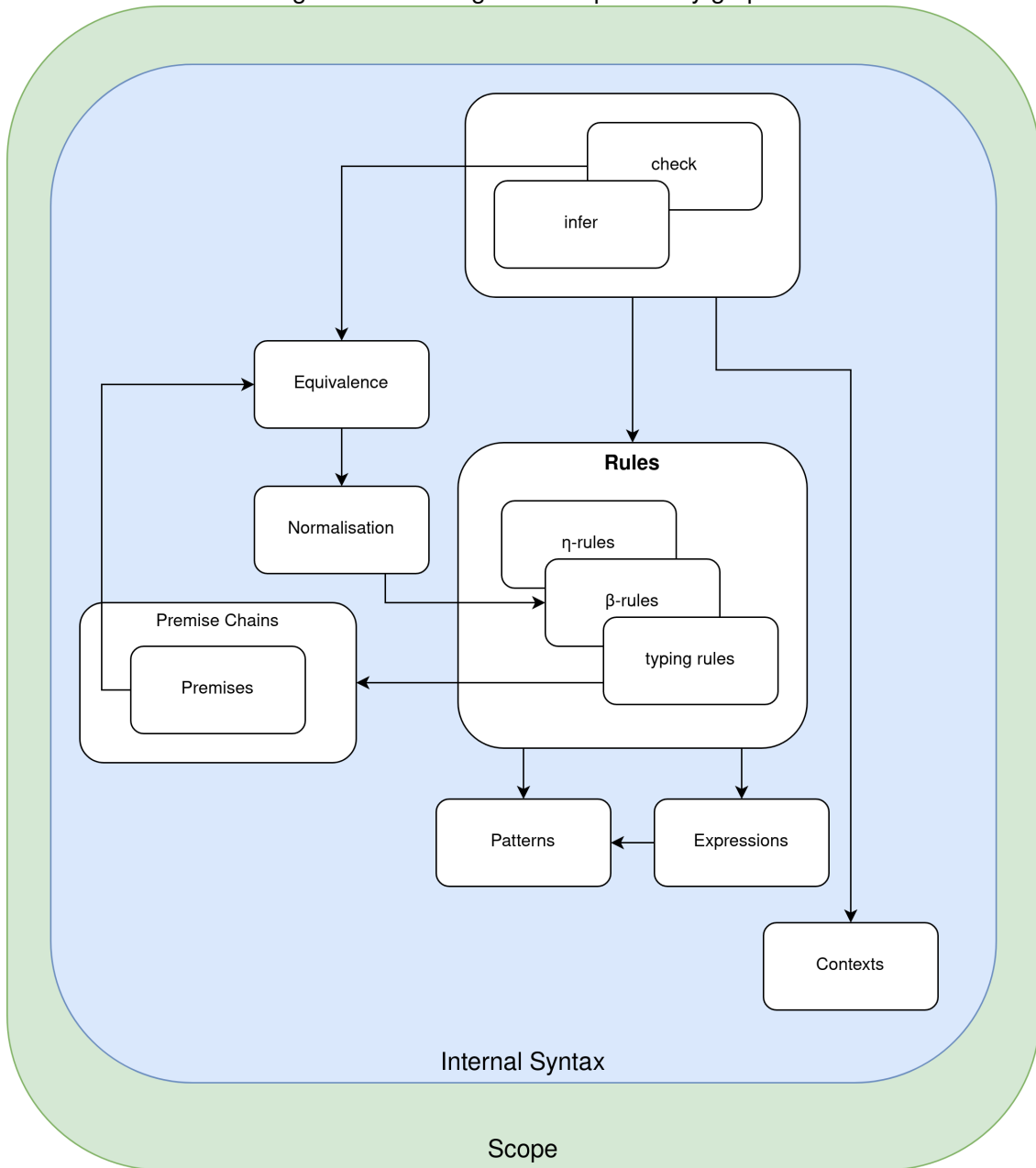
To finish our rules we require some way to impose conditions on a pattern so that we might encode the premises of our rules. For this purpose we provide premises and premise chains, taking care to ensure that a premise chain is guaranteed to establish a minimum level of trust in some pattern.

Since our implementation accepts dependent types, we will need to provide operational semantics and normalisation (which in turn necessitates some implementation of substitution). We will need these concepts to check type equivalence.

We are explicit in handling scope throughout the project, so before we describe the implementation of everything we have introduced so far, we must begin by defining our representation of scope and provide tools for its manipulation. Using our tools we can create well-scoped contexts which help to eliminate a large class of potential errors from the type checking process.

To help visualize the structure of the software, we provide a high-level dependency graph in figure 6.1. We have arranged the graph so that the dependencies roughly flow from top to bottom, and so we will have to implement the software from the bottom up according to this figure.

Figure 6.1: The high-level dependency graph



6.2 A core language

First, we introduce the concept of scope and what it means to be scoped. In our system, all variables are de Bruijn indexed and our scope can be represented by a single natural number. We take 0 to mean nothing in scope, 1 to mean a single thing in scope and so on. A scoped entity is one that is defined in terms of a scope.

```
Scope =  $\mathbb{N}$ 
Scoped = Scope  $\rightarrow$  Set
```

A variable is our first example of such a scoped entity which we construct in order to maintain the invariant that a variable should always refer to something in scope.

```
data Var : Scoped where
  ze : Var (suc  $\gamma$ )
  su : Var  $\gamma \rightarrow$  Var (suc  $\gamma$ )
```

Our internal syntax is split into two broad grammatical classes: constructions and computations. Constructions will have their type checked, while computations have their type synthesized.

```
data Const  $\gamma$  where
  '      : String  $\rightarrow$  Const  $\gamma$ 
  _•_    : Const  $\gamma \rightarrow$  Const  $\gamma \rightarrow$  Const  $\gamma$ 
  bind   : Const (suc  $\gamma$ )  $\rightarrow$  Const  $\gamma$ 
  thunk  : Compu  $\gamma \rightarrow$  Const  $\gamma$ 

data Compu  $\gamma$  where
  var    : Var  $\gamma \rightarrow$  Compu  $\gamma$ 
  elim   : Compu  $\gamma \rightarrow$  Const  $\gamma \rightarrow$  Compu  $\gamma$ 
  _::_   : Const  $\gamma \rightarrow$  Const  $\gamma \rightarrow$  Compu  $\gamma$ 

data Dir : Set where const compu : Dir

Term : Dir  $\rightarrow$  Scoped
Term const = Const
Term compu = Compu
```

This syntax gives us the means to represent atoms of original syntax with ' , pairs of syntax elements, binding sites (thus increasing the scope for a subterm), variables, type annotated terms with "::" and eliminations - the sites of β -reductions. We are also able to embed computations into constructions using thunk, as being able to synthesize the type of a term guarantees that we are able to check it.

Note that blindly embedding synthesizable terms with a thunk is not always the best course of action. In the case of an annotated term, we already have a suitable construction under the annotation. For convenience, we provide a function to perform this embedding and hence a function to take *any* computation to a construction and another to take any term to a construction. We also prove the opposite functionality, allowing us to take any term to a computation, however in this case we need to supply a construction that we claim to be the type. This may or may not be used in constructing the computation.

```

→ : Compu γ → Const γ
→ (t :: T) = t
→ x = thunk x

→→ : Term d γ → Const γ
→→ {const} = id
→→ {compu} = →

←← : Term d γ → Const γ → Compu γ
←← {const} (thunk x) T = x
←← {const} t T        = t :: T
←← {compu} t _        = t

```

6.3 Thinnings

A key concept that will be used throughout this implementation is that of a thinning. Thinnings describe embeddings between scopes and are denoted $\delta \sqsubseteq \gamma$ where they embed some scope δ into another scope γ and as such it must be that $\delta \leq \gamma$.

Some thinning $\delta \sqsubseteq \gamma$ can be represented a bit-vector γ digits long where each digit might be 0 indicating that the variable is new in γ or 1 indicating that it previously existed in δ . A thinning may also be interpreted as a selection from some scope γ , and we may use singleton thinnings in order to represent variables in this way. Our implementation maintains the $\delta \leq \gamma$ invariant by construction and is shown here.

```

data _⊆_ : Scope → Scope → Set where
  ε  : 0 ⊆ 0
  _O : γ ⊆ δ → γ ⊆ suc δ
  _I : γ ⊆ δ → suc γ ⊆ suc δ

```

We define some elements for later use: the identity thinning, the empty thinning and what it means to append two thinnings as well as injections to a concatenation in both directions. Their type signatures are as follows.

```

ι      : γ ⊆ γ
∅      : 0 ⊆ γ
_++_   : δ ⊆ γ → δ' ⊆ γ' → (δ + δ') ⊆ (γ + γ')
_<_    : (γ δ : Scope) → γ ⊆ (γ + δ)
_>_    : (γ δ : Scope) → δ ⊆ (γ + δ)

```

There are various scoped entities on which it makes sense for thinnings to act by lifting the entity into a wider scope. To capture this commonly used behaviour we introduce the concept of Thinnable and also provide its counterpart Selectable for the opposite action of using some thinning to somehow narrow the scope.

```

Thinnable : Scoped → Set
Thinnable X = ∀ {δ} {γ} → X δ → (δ ⊆ γ) → X γ

```

$\text{Selectable} : \text{Scoped} \rightarrow \text{Set}$
 $\text{Selectable } X = \forall \{\delta\} \{\gamma\} \rightarrow (\delta \sqsubseteq \gamma) \rightarrow X \gamma \rightarrow X \delta$

There are many scoped entities that we will wish a thinning to act on, and so we adopt the convention that all functions detailing an action of thinnings begin with "<" except the action of a thinning on another thinning which equates to composition, and so we use the more traditional \circ notation.

$_ \circ _ : \text{Thinnable } (\delta \sqsubseteq _)$
 $_ \langle \text{var} _ : \text{Thinnable Var}$
 $_ \langle \text{term} _ : \text{Thinnable (Term } d)$

While Selectable is used far less often, there are key areas where it makes sense, for instance in using a thinning to select elements from a vector.

$_ ! _ : \text{Selectable (BwdVec } X)$

A weakening is a special case of a thinning where the scope is extended by one at its most local position, for example when passing under a binder. This concept is captured here, as well as the relevant type that details the action of a weakening.

$\uparrow : \gamma \sqsubseteq (\text{suc } \gamma)$
 $\uparrow = \iota \circ$
 $\text{Weakenable} : \text{Scoped} \rightarrow \text{Set}$
 $\text{Weakenable } T = \forall \{\gamma\} \rightarrow T \gamma \rightarrow T (\text{suc } \gamma)$
 $\text{weaken} : \text{Thinnable } T \rightarrow \text{Weakenable } T$
 $\text{weaken } \langle t = \langle t \uparrow$

When providing Weakenables, we adopt the naming convention of beginning their identifiers with "^".

$_ ^ : \text{Weakenable } (\gamma \sqsubseteq _)$
 $_ ^ \text{var} : \text{Weakenable Var}$
 $_ ^ \text{term} : \text{Weakenable (Term } d)$

6.4 Substitution

Substitutions are defined as backward vectors: vectors that grow by appending elements to the right-hand side as opposed to the left. A substitution is defined in terms of two scopes, the scope of the target of substitution, and the scope of the entities we will substitute into the target.

$_ \Rightarrow [_] : \text{Scope} \rightarrow \text{Scoped} \rightarrow \text{Scope} \rightarrow \text{Set}$
 $\gamma \Rightarrow [X] \delta = \text{BwdVec } (X \delta) \gamma$

We are able to look up individual variables in a substitution, in section 6.3 we explained that this is just a special case of using a thinning to select from a substitution. We also capture a key notion that there is a type that describes scoped things on which we may perform substitutions.

Substitutable : **Scoped** \rightarrow **Set**
Substitutable $T = \forall \{\gamma\} \{\gamma'\} \rightarrow T \gamma \rightarrow \gamma \Rightarrow [T] \gamma' \rightarrow T \gamma'$

Finally, we acknowledge that given two scope transformations, we can define a type to represent composition, which leads us nicely to a definition for composition of substitutions.

Composable : (**Scope** \rightarrow **Scope** \rightarrow **Set**) \rightarrow **Set**
Composable $X = \forall \{\gamma\} \{\gamma'\} \{\gamma''\} \rightarrow (X \gamma \gamma') \rightarrow (X \gamma' \gamma'') \rightarrow (X \gamma \gamma'')$

$[] \circ \sigma$: **Substitutable** $T \rightarrow$ **Composable** $\Rightarrow [T]$
 $[/] \varepsilon \circ \sigma \sigma' = \varepsilon$
 $[/] (\sigma \dashv, x) \circ \sigma \sigma' = ([/] \sigma \circ \sigma \sigma') \dashv, / x \sigma'$

Here we have paid attention to the types to aid the readability of later definitions. This can be seen in the type of our final definition above which states that if some T is substitutable, then we are able to compose the substitutions.

6.5 Term substitution

Following our generic notion of substitution, we now specialise our definition to substitute computations. We supply the usual thinning and weakening mechanics that we rely on as well as defining the identity substitution.

\Rightarrow : **Scope** \rightarrow **Scope** \rightarrow **Set**
 $\gamma \Rightarrow \delta = \gamma \Rightarrow [Compu] \delta$
 $\langle \sigma$: **Thinnable** ($\gamma \Rightarrow$)
 \wedge : **Weakenable** ($\gamma \Rightarrow$)
 id : $\gamma \Rightarrow \gamma$

We also define the action of such a substitution on a term, where most cases merely recurse on direct substructures except for two cases of interest that we show here. The first is that we must ensure we alter the substitution accordingly as we pass under binders, introducing an identity substitution for the newly bound variable after weakening the scope of the substitution we already have. The second is where we find some variable and perform the actual substitution.

$/term$: **Term** $d \gamma \rightarrow \gamma \Rightarrow \delta \rightarrow$ **Term** $d \delta$
 $/term \{const\} (bind \ t) \ \sigma = bind \ (t /term \ (\sigma \wedge \dashv, var \ ze))$
 $/term \{compu\} (var \ v) \ \sigma = lookup \ (Term \ compu) \ \sigma \ v$

6.6 Contexts

We define contexts to be substitutions of constructions. We ensure that the target of the substitution is scoped identically to the things we will substitute, keeping our contexts in a pre-thinned state. This allows us to use terms directly from it without having to fix up the scope first.

$\text{Context} : \text{Scope} \rightarrow \text{Set}$
 $\text{Context } \gamma = \gamma \Rightarrow [\text{Const}] \gamma$

We provide a special function to extend contexts that appends a new element to the context (making it no longer a context as $\text{succ} \gamma \neq \gamma$ in $(\text{succ} \gamma) \Rightarrow [\text{Const}] \gamma$) before weakening the whole substitution yielding the context $(\text{succ} \gamma) \Rightarrow [\text{Const}] (\text{succ} \gamma)$. The extra effort exerted here keeps our contexts in the previously explained pre-thinned state.

$_ , _ : \text{Context } \gamma \rightarrow \text{Const } \gamma \rightarrow \text{Context } (\text{succ } \gamma)$
 $\Gamma, t = (\Gamma -, t) \wedge \Gamma$

If we only ever use ε (for creating an empty backwards vector) and $_ , _$ to construct our contexts, we can be assured that they will always be valid and pre-thinned. No such assurances can be made if the raw vector appending data constructor $_ -, _$ is used.

6.7 Openings

A useful operation later will be to open up scoped entities. We take this to mean lifting an entity to include some scope γ without capturing anything scoped in γ . When opening some entity scoped in δ by γ , the result is an entity scoped in $\gamma + \delta$. Any reference to something in scope refers to the same thing it did before it was opened. If some entity scoped in δ has an action of a thinning defined for it, opening it by γ may be defined as the action of the thinning $\gamma \triangleleft \delta$.

$\text{Openable} : (T : \text{Scoped}) \rightarrow \text{Set}$
 $\text{Openable } T = \forall \{\delta\} \rightarrow (\gamma : \text{Scope}) \rightarrow T \delta \rightarrow T (\gamma + \delta)$

As with previous concepts, we adopt a naming convention when constructing openings where we begin identifiers with " \otimes ".

6.8 Patterns

A key concept we will need to implement our generic type-checker is that of a pattern. Our rules are defined not in terms of concrete pieces of syntax, but in terms of patterns, which we then match against concrete syntax.

Our concept of a pattern is structurally identical to that of a construction, except that we exclude thunks and introduce the notion of a *place* that may stand for an arbitrary construction so long as we can thin it to the required scope.

The dual concept of a pattern is that of an environment. It is structurally similar to a pattern except where a pattern may have a *place*, an environment answers this call with a

thing that fits in the place. As always, we are careful to handle scope correctly in the case of *bind* when constructing environments so that the underlying entity is defined in the weakened scope.

Environments are indexed by a pattern so that we can ensure that they always match exactly the pattern intended (they offer a *thing* for every *place*). Consequently, this allows us a non-failable operation to generate a term from pattern p and its associated p -Env

```
data Pattern (γ : Scope) : Set where
  '      : String → Pattern γ
  _•_    : Pattern γ → Pattern γ → Pattern γ
  bind   : Pattern (suc γ) → Pattern γ
  place  : {δ : Scope} → δ ⊆ γ → Pattern γ
  ⊥      : Pattern γ

data _-Env {γ : Scope} : Pattern γ → Set where
  '      : {s : String} → (' s) -Env
  _•_    : q -Env → r -Env → (q • r) -Env
  bind   : t -Env → (bind t) -Env
  thing  : {θ : δ ⊆ γ} → Const δ → (place θ) -Env
```

We define the γ opening of a pattern by recursing structurally until we reach some *place* and prepending the thinning by the identity thinning length γ .

$_ \otimes _ : \text{Openable Pattern}$

We now have the required machinery to define pattern matching. We do not define matching over some term and pattern scoped identically, but more generally over some term that might be operating in a wider scope. This is crucial as a pattern is often constructed in the empty scope so that we might not refer to arbitrary free variables when defining formal rules, however, these rules may then be applied in some non-empty scope.

```
match : Const (δ + γ) → (p : Pattern γ) → Maybe ((δ ⊗ p) -Env)
match {γ = γ} t (place {δ'} θ) with γ ≡n δ'
... | yes refl = just (thing t)
... | no      = nothing
match (' a) (' c) with a == c
... | true  = just '
... | false = nothing
match (s • t) (p • q) = do
  x ← match s p
  y ← match t q
  just (x • y)
match (bind t) (bind p) = do
  x ← match t p
  just (bind x)
match _ _ = nothing
```

When constructing formal rules, it is critical that we are able to refer to distinct places in a pattern. For this purpose, we define a schematic variable. This variable is able to trace a path

through the pattern that indexes it, terminating with a \star to mark the *place* we are referring to. Since environments follow the same structure as their pattern we might also use schematic variables here in the same way. We index this type with the pattern in which it identifies a place and use the index to guarantee the validity of a schematic variable with regards to its pattern.

```

data svar : Pattern  $\gamma$   $\rightarrow$  Scope  $\rightarrow$  Set where
   $\star$       :  $\{\theta : \delta \sqsubseteq \gamma\} \rightarrow \text{svar } (\text{place } \theta) \delta$ 
   $\_ \bullet$      :  $\text{svar } p \delta \rightarrow \text{svar } (p \bullet q) \delta$ 
   $\bullet \_$      :  $\text{svar } q \delta \rightarrow \text{svar } (p \bullet q) \delta$ 
  bind     :  $\text{svar } q \delta \rightarrow \text{svar } (\text{bind } q) \delta$ 

 $\_!!\_$  :  $\text{svar } p \delta \rightarrow (\gamma' \otimes p) \text{-Env} \rightarrow \text{Const } (\gamma' + \delta)$ 
 $\star$    !! thing  $x = x$ 
 $(v \bullet)$  !!  $(p \bullet q) = v !! p$ 
 $(\bullet v)$  !!  $(p \bullet q) = v !! q$ 
bind  $v !! \text{bind } t = v !! t$ 

```

We define a few less interesting but critical utility functions for later use. We give the means to remove a place from a pattern, replacing it with a trivial atom. Similarly, we extend the same functionality to environments. We also provide the means to build a term from a pattern and some appropriate environment.

```

 $\_ - \_$       :  $(p : \text{Pattern } \gamma) \rightarrow \text{svar } p \delta \rightarrow \text{Pattern } \gamma$ 
 $\_ \text{-penv } \_$  :  $p \text{-Env} \rightarrow (\xi : \text{svar } p \delta) \rightarrow (p - \xi) \text{-Env}$ 
termFrom :  $(p : \text{Pattern } \gamma) \rightarrow (\delta \otimes p) \text{-Env} \rightarrow \text{Const } (\delta + \gamma)$ 

```

We will later find it useful to traverse a pattern and build a potential svar 'on the way down' so that when we reach a *place* we have the svar that refers to it and so we construct a type to help us.

The notion is that instead of encoding some path to a *place* in the pattern, we instead encode some path between the pattern p and some subpattern q . X encodes an empty path between a pattern and itself, then we may extend it with " $_ \bullet$ ", " $\bullet _$ " or " bind " so that we now show the path between some term and a direct substructure. We can continue in a similar fashion, building some path, and if the path ends in a *place*, then we know we might convert it to the appropriate svar.

```

data svar-builder : Pattern  $\gamma$   $\rightarrow$  Pattern  $\delta$   $\rightarrow$  Set where
  X      : svar-builder  $p p$ 
   $\_ \bullet$     : svar-builder  $p p' \rightarrow \text{svar-builder } (p \bullet q) p'$ 
   $\bullet \_$     : svar-builder  $q p' \rightarrow \text{svar-builder } (p \bullet q) p'$ 
  bind   : svar-builder  $p p' \rightarrow \text{svar-builder } (\text{bind } p) p'$ 

```

We intend to use this concept by initially selecting both the indexed patterns to be the pattern we are about to traverse, then as we traverse the structure of the pattern we use the combinators defined below to strip constructors off the second index while simultaneously encoding the removed constructor in the path being constructed. We give the first combinator in full here as an example, and the types of the other two.

```

 $\Leftarrow$  : svar-builder  $p (lf \bullet rt) \rightarrow \text{svar-builder } p lf$ 
 $\Leftarrow X$       =  $X \bullet$ 

```

```

⇐ (v •)    = (⇐ v) •
⇐ (• v)    = • (⇐ v)
⇐ (bind v) = bind (⇐ v)

⇒ : svar-builder p (lf • rt) → svar-builder p rt
↳ : svar-builder p (bind bn) → svar-builder p bn

```

Finally, we are able to build an actual svar from a builder only if it encodes a path in some pattern p that leads to a place in p .

```

build : {θ : δ ⊆ γ} →
      svar-builder p (place θ) →
      svar p δ
build X      = ★
build (v •)  = (build v) •
build (• v)  = • (build v)
build (bind v) = bind (build v)

```

6.9 Expressions

Expressions allow us to define a way that we might construct a term from a pattern that represents something that we trust. For example, if the expression is showing how one might build the required term as the output of some type-synthesis rule for elimination, what is trusted might be the type of the target and whatever we learn to trust in the premises. These trusted patterns contain places that sculpt out component parts in a piece of matched syntax and allow us to use these parts in constructing a new term.

Expressions mirror the structure of terms except that they include the option to reference some place in a pattern and instantiate it with some substitution of its free variables. This means that we might build a term based on some piece of syntax that we do not yet have until the pattern is later matched.

```

data Con (p : Pattern δ) (γ : Scope) : Set
data Com (p : Pattern δ) (γ : Scope) : Set

data Con p γ where
  '      : String → Con p γ
  _•_    : Con p γ → Con p γ → Con p γ
  bind  : Con p (suc γ) → Con p γ
  thunk : Com p γ → Con p γ
  _/_   : svar p γ' → γ' ⇒ [Com p] γ → Con p γ

data Com p γ where
  var  : Var γ → Com p γ
  elim : Com p γ → Con p γ → Com p γ
  _::_ : Con p γ → Con p γ → Com p γ

Expr : Pattern δ → Dir → Scoped

```

```

Expr p const γ = Con p γ
Expr p compu γ = Com p γ

```

We are able to generate a term from an expression and some opening of an environment for the trusted pattern. Most of the cases recurse structurally, while variables are simply opened to encompassing scope, however we demonstrate below how svar instantiations are processed.

```

toTerm : (γ ⊗ p) -Env → Expr p d γ' → Term d (γ + γ')
toTerm {γ = γ} {d = const} {γ' = γ} penv (ξ / σ)
  = let σ' = map-toTerm σ penv in
    let id-fv = id (σ (γ ◁ γ')) in
    (ξ !! penv) /term (id-fv ++ σ')

```

We first resolve the substitution of expressions to one of terms (here we must inline a specialisation of map to satisfy Agda's termination checker) before extending it with the identity substitution for all the free variables and finally performing the substitution on the term taken from the environment. We substitute the bound variables in the term but ensure not to affect any free variables.

6.10 Premises

A key concept we will use when defining rules is that of a premise. We use premises to accumulate trust in some pattern, potentially discharging a place in the pattern as 'trusted'. In our implementation of premise, p is some pattern that identifies places that we already trust while q is the pattern of untrusted places. The premise establishes trust in some p' and shows what remains untrusted with q' . While we can encode various checks with our premises, only two increase the accumulated trust: either a statement that some $place$ is a type or a statement that it is *of* some type.

In the definition of formal rules, we cannot refer to arbitrary free variables and will give a chain of premises where the patterns are defined in the empty scope (unless we have ducked under the \vdash premise). We will later require the ability to open such premises when using them to establish trust in an arbitrary scope and so we define such an opening.

```

data Prem (p q : Pattern δ) (γ : Scope) :
  (p' : Pattern γ) → (q' : Pattern δ) → Set where
type   : (ξ : svar q δ') → (θ : δ' ⊆ γ) → Prem p q γ (place θ) (q - ξ)
_≲'_[] : (T : Expr p const γ) → (ξ : svar q δ') → (θ : δ' ⊆ γ) → Prem p q γ (place θ) (q - ξ)
_≡'_   : Expr p const γ → Expr p const γ → Prem p q γ ("T") q
univ   : Expr p const γ → Prem p q γ ("T") q
_⊢'_   : Expr p const γ → Prem p q (suc γ) p' q'' → Prem p q γ (bind p') q''

⊗premise : (δ : Scope) →
  Prem p q γ p' q' →
  Prem (δ ⊗ p) (δ ⊗ q) (δ + γ) (δ ⊗ p') (δ ⊗ q')

```

Before showing how premises may be chained together, we must present supporting definition. This is used to prove that a pattern contains no places. We index this type with the pattern in question, mandate that the type follows the pattern structurally but provide no way to encode a place.

```

data _Placeless {γ : Scope} : Pattern γ → Set where
  '      : (s : String) → ' s Placeless
  _•_    : {l r : Pattern γ} → (l Placeless) → (r Placeless) → (l • r) Placeless
  bind   : {t : Pattern (suc γ)} → (t Placeless) → (bind t) Placeless
  ⊥      : ⊥ Placeless

```

We now use these concepts to define a chain of premises that is guaranteed to establish complete trust in some pattern.

We may string together premises, threading what is left to trust after each premise into what is still to trust in the next. We collect everything that we learn to trust along the way, allowing later premises to refer to entities whose trust was established by earlier premises.

If we wish to end a chain of premises, we must show that there is nothing left that requires trusting by proving that what we have been asked to trust is a pattern that contains no places.

```

data Prems (p0 q0 : Pattern γ) : (p2 : Pattern γ) → Set where
  ε      : (q0 Placeless) → Prems p0 q0 p0
  _⇒_    : Prem p0 q0 γ pg q1' →
    Prems (p0 • pg) q1' p2' →
    Prems p0 q0 p2'

⊗premises : (δ : Scope) → Prems p q p2 → Prems (δ ⊗ p) (δ ⊗ q) (δ ⊗ p2)

```

6.11 Typing rules

A TypeRule is used to establish the conditions under which a piece of syntax, the subject, is determined to be a type. The rule must contain a premise chain that establishes complete trust in this subject from no prior trusted knowledge. Matching this rule is matching a piece of syntax against the subject and results in some environment for the subject.

```

record TypeRule : Set where
  field
    subject : Pattern 0
    premises : Σ[ p' ∈ Pattern 0 ] Prems (' "⊤" ) subject p'

```

The type-checking rule involves both an input and a subject. For $T \ni t$ we take T to be a trusted input and seek to establish trust in t. Our premise chain reflects this by using the input as its trusted pattern and seeking trust in the subject. Matching occurs on both the input and the subject. If successful, it returns a pair of environments.

```

record ∋rule : Set where
  field
    subject : Pattern 0
    input    : Pattern 0
    premises : Σ[ p' ∈ Pattern 0 ] Prems input subject p'

```

We may also use this rule to reverse engineer the type of any place in the subject, taking advantage of the fact that our premise chain can only establish trust in a place by ultimately

making a statement either about its type or about it being a type. This function turned out to be non-trivial, and many proofs were required to convince Agda to accept the implementation. An alternative approach might have been to construct a pattern environment where each *thing* corresponded to the type of the corresponding *place* in the pattern. We could have even gone so far as to generalise what may be stored at places in patterns and teased out some applicative structure.

```

typeOf : (r :  $\exists$ rule)       $\rightarrow$ 
        (s : svar ( $\gamma \otimes$  subject r)  $\delta$ )  $\rightarrow$ 
        ( $\gamma \otimes$  input r) -Env       $\rightarrow$ 
        (( $\gamma \otimes$  (subject r)) -Env)  $\rightarrow$ 
        Const ((bind-count s) +  $\gamma$ )

```

Our elimination typing rules work slightly differently from those which operate on constructions above. Eliminations use some *eliminator* to eliminate some *target*. An elimination rule, if it matches, is used to try and *supply* us with the type of the resulting term.

To build the output, we don't need to hold any description of the target term, it is enough that we describe what type we require it to be. In our elimination typing rules, we take a pattern to match against the type of the target, another to match against the *eliminator*, and seek to establish trust in the *eliminator* under the assumption that we trust the target type. We use an Expression to build the type of the elimination from everything in which the premise chain has established trust. Matching an elimination rule requires matching both the target pattern and the eliminator.

```

record ElimRule : Set where
  field
    targetPat : Pattern 0
    eliminator : Pattern 0
    premises  :  $\Sigma$ [  $p' \in$  Pattern 0 ] Prems targetPat eliminator  $p'$ 
    output    : Expr (proj1 premises) const 0

```

6.12 η rules

Later we will show how we might fully normalise a term using a technique known as normalisation by evaluation. To do this, we will find that we require the ability to perform η -expansion and so we provide a rule to help us achieve this.

In our η -rule, we store only the eliminator for each place in the pattern. To generate the η -expanded form, we map the elimination of the target over this environment of eliminators to get the full eliminations that are destined for each place in the pattern. This is very straightforward as a concept but we have to fix up our types a little to convince Agda of the well-scopedness. We use a mapping function we defined to easily map over an environment and help us concisely build the eliminations from the environment of eliminators.

```

record  $\eta$ -Rule : Set where
  open  $\exists$ rule

  field

```

```

checkRule :  $\exists$ rule
eliminators : subject checkRule -Env
headForm = subject checkRule

eliminations : (type target : Const  $\gamma$ )  $\rightarrow$  ( $\gamma \otimes$  headForm) -Env
eliminations { $\gamma$ } type target
= map
  ( $\lambda$  { $\delta$ } const  $\rightarrow$  thunk (elim (( $\leftarrow\leftarrow$  target type) (term ( $\gamma \triangleleft \delta$ )) ( $\gamma \otimes$  term const))))
  eliminators

 $\eta$ -expand : (type target : Const  $\gamma$ )  $\rightarrow$  Const  $\gamma$ 
 $\eta$ -expand ty tm = termFrom headForm (eliminations ty tm)

```

6.13 Semantics

When constructing our elimination typing rule, we noted that we did not require information about the syntax of the target itself, only its type. For β -reduction we now require this information and so we create a type that wraps an elimination typing rule but provides the extra information we need: some pattern for the target of elimination and some expression showing how we construct the reduced term.

Matching a β -rule involves matching the enclosed elimination rule and the pattern of the target, when such a rule is matched we might then appeal to β -reduce. We are guaranteed to be able to return a computation here since we have kept the required type information to hand and so we make use of the $\leftarrow\leftarrow$ operator that we introduced in section 6.2.

```

record  $\beta$ -rule : Set where
  open ElimRule
  field
    target    : Pattern 0
    erule     : ElimRule
    redTerm   : Con (target • targetPat erule • eliminator erule) 0
  redType = output erule
  eprems   = proj2 (premises erule)

 $\beta$ Rule-Env : { $\gamma$  : Scope}  $\rightarrow$  Set
 $\beta$ Rule-Env { $\gamma$ } = ( $\gamma \otimes$  target) -Env  $\times$  ERuleEnv erule

 $\beta$ -reduce :  $\beta$ Rule-Env  $\rightarrow$ 
  ( $\gamma \otimes$  (proj1 (premises erule))) -Env  $\rightarrow$ 
  Compu  $\gamma$ 
 $\beta$ -reduce (tenv, tyenv, eenv) p'env
=  $\leftarrow\leftarrow$  (toTerm (tenv • tyenv • eenv) redTerm)
  (toTerm p'env redType)

```

We now have what is required to find a matching rule by matching the various patterns that it may contain. We iterate over the list of rules and return the first match. We chose a 'first match' approach here for simplicity of implementation, however it is acknowledged that other approaches are far more suitable. We discuss this later in our evaluation of the software.

In this type, we first encounter the failure handling monad that we will use to propagate errors. It works as expected, either succeeding with a value or failing with an error message.

```
findRule : List  $\beta$ -rule  $\rightarrow$ 
  (tar type elim : Const  $\gamma$ )  $\rightarrow$ 
  Failable (  $\Sigma$  [  $r \in \beta$ -rule ]  $\beta$ Rule-Env  $r$  )
```

To achieve reduction we will need to check the premise chain, in the enclosed elimination to access the required environments. The mechanics of checking premise chains is not the business of β -reduction code and so we instead give a type to represent a function that can do this for us and insist that the caller provide it. We also provide a similar type appended with ' , that captures the same intuition except that it is "pre-loaded" with the context.

```
PremChecker =  $\forall \{ \delta \} \rightarrow$  Context  $\delta \rightarrow$ 
  {  $p \ q \ p' : \text{Pattern } \delta$  }  $\rightarrow$ 
   $p$ -Env  $\rightarrow q$ -Env  $\rightarrow$ 
  Prems  $p \ q \ p' \rightarrow$ 
  Failable ( $p'$ -Env)
```

Reduction with regards to a list of rules can now be defined in three stages; we first attempt to find a matching rule, then if this succeeds we check the premise chain and finally if both the previous stages succeed then we make our call to β -reduce.

```
reduce : List  $\beta$ -rule  $\rightarrow$ 
  PremChecker'  $\gamma \rightarrow$ 
  (tar type elim : Const  $\gamma$ )  $\rightarrow$ 
  Failable (Compu  $\gamma$ )
reduce { $\gamma$ } rules checkprems ta ty el
= do
  (rule , t , ty , e)  $\leftarrow$  findRule rules ta ty el
  p'env  $\leftarrow$  checkprems ty e ( $\otimes$ premises  $\gamma$  (eprem rule))
  succeed ( $\beta$ -reduce rule (t , ty , e) p'env)
where open ElimRule
```

β -reduction failure depends solely on the existence of the rule, and the successful checking of the premise chain. The call to β -reduce is not in itself a failable operation.

6.14 Normalisation by evaluation

We previously gave a type describing a premise-chain-checker, now we give similar types for things that will reduce for us, and things that might infer types. What is noticeably absent from these types is the existence of any kind of rules. Our normalisation procedure should be rule-agnostic and so we achieve this by being very careful where we choose to mention them, and, critically, where we choose *not* to. We must, however, bear the rules in mind when we normalise and so a conscious decision is made here for normalisation to be a non-failable operation. In a world of user-supplied typing rules, we decide that flexibility is important and so when we encounter problems we take a "do as much as you can" approach to normalisation.

```
Reducer =  $\forall \{ \gamma \}$   $\rightarrow$ 
  PremChecker'  $\gamma \rightarrow$ 
```

```

      (tar type elim : Const  $\gamma$ ) →
      Failable (Compu  $\gamma$ )
Inferer =  $\forall \{\gamma\}$  →
      Context  $\gamma$  →
      (term : Term compu  $\gamma$ ) →
      Failable (Term const  $\gamma$ )

```

To implement normalisation by evaluation, we must first define an evaluation function to reduce any eliminations. Most cases proceed as one might expect, traversing the structure of the term to find eliminations that we might hope to reduce. We reduce the target and eliminator in some elimination even if we fail to infer the type of the target (and so fail to reduce the elimination). This is to keep separate the concerns of type-checking and normalisation and it is our opinion that this evaluation should not fail because of a type-error. Instead, the process continues with some *unknown* type placeholder, meaning that it will not match any rules that might produce erroneous results. The responsibility of type-checking the term falls to the caller.

```

_<_>_<_>_<_> : Reducer × Inferer × PremsChecker →
  Context  $\gamma$  →
  Term  $d$   $\gamma$  →
  Const  $\gamma$ 
(rd , inf , pc) -  $\Gamma$  <_> T <_> = <_> T <_>  $\Gamma$ 
where
  <_> : Term  $d$   $\gamma$  → Context  $\gamma$  → Const  $\gamma$ 
  - ...
  <_> {compu} { $\gamma$ } (elim t e)  $\Gamma$  with inf  $\Gamma$  t
  ... | fail n = thunk (elim (←← (<_> t <_>  $\Gamma$ ) (' "unknown")) (<_> e <_>  $\Gamma$ ))
  ... | succeed ty with rd (pc  $\Gamma$ ) (<_> t <_>  $\Gamma$ ) ty (<_> e <_>  $\Gamma$ )
  ... | succeed x = <_> x <_>  $\Gamma$ 
  ... | fail x = thunk (elim (←← (<_> t <_>  $\Gamma$ ) ty) (<_> e <_>  $\Gamma$ ))
  - ...

```

The final step in normalisation by evaluation is trying to build the correct head form for the associated types of the term and its subterms. We do this using our η -Rule mechanics from 6.12, our svar-builder from 6.8 and our method of computing the type of a place in a pattern given some \ni -rule from 6.11.

Here we match some η -rule giving us the pattern for the head form, and also the eliminations that will occur at each place in that pattern. We then navigate down through the structure of the head pattern in our helper function so that we get to each place in the pattern, and when we do, we have built the svar that identifies it. This allows us to use the \ni rule to get the type of that particular place, which in turn facilitates the recursive call to qt to generate the head form at the nested place.

To satisfy the well-scopedness of the subterms, we must maintain a proof that we are constructing a well-scoped svar-builder. When we traverse under a binder in a head form pattern, we are forced to add such a binder to our svar-builder to maintain its well-scopedness. Our proof achieves this by maintaining the invariant that the scope of the subterm currently identified by the svar-builder is always the original scope plus however many binders we have added when constructing it.

```

qt : List  $\eta$ -Rule  $\rightarrow$  (ty tm : Const  $\gamma$ )  $\rightarrow$  Const  $\gamma$ 
qt { $\gamma = \gamma$ } rs ty v with EtaRule.findRule rs ty
... | fail x          = v
... | succeed (r , i) = let elms = eliminations r ty v in
                        helper ( $\gamma \otimes$  headForm r) (i • elms) X refl elms

where
   $\exists$ rl = checkRule r
  helper :  $\forall \{ \gamma \} (q : \text{Pattern } \gamma) \rightarrow$ 
    (( $\gamma \otimes$  input  $\exists$ rl) • ( $\gamma \otimes$  subject  $\exists$ rl)) -Env  $\rightarrow$ 
    (v : svar-builder ( $\gamma \otimes$  subject  $\exists$ rl) q)  $\rightarrow$ 
     $\gamma' \equiv$  bind-count-bl v +  $\gamma$   $\rightarrow$ 
    q -Env  $\rightarrow$ 
    Const (bind-count-bl v +  $\gamma$ )
  helper (' x) is v prf '
    = ' x
  helper (s • t) is v prf (es • et)
    = helper s is ( $\Leftarrow$  v) prf es • helper t is ( $\Rightarrow$  v) prf et
  helper (bind q) is v prf (bind et)
    = bind (helper q is ( $\vdash$  v) (cong suc prf) et)
  helper (place  $\theta$ ) (i • s) v prf (thing el)
    = qt rs
      (typeOf (checkRule r) (build v) i s)
      (subst Const prf (el <term  $\theta$ ))

```

Normalisation is then a process of first evaluating the term before building suitable head forms.

```

normalize : List  $\eta$ -Rule  $\rightarrow$  List  $\beta$ -rule  $\rightarrow$ 
  Inferer  $\times$  PremsChecker  $\rightarrow$ 
  Context  $\gamma$   $\rightarrow$ 
  (type : Const  $\gamma$ )  $\rightarrow$ 
  (term : Term d  $\gamma$ )  $\rightarrow$ 
  Const  $\gamma$ 
normalize  $\eta$ s  $\beta$ s i&c  $\Gamma$  ty = (qt  $\eta$ s ty)  $\circ$  ((reduce  $\beta$ s , i&c) -  $\Gamma$   $\llcorner \llcorner$ )

```

6.15 Checking types

We now have all of the required definitions to construct the process of type checking. This process is a set of mutually recursive definitions which we will describe here.

When we check types, it is convenient to collect all the rules together so that we might easily pass them around. We do so with the following type, although in our evaluation we will reveal why it was a mistake to collect these rules together using lists.

```

data RuleSet : Set where
  rs : List TypeRule  $\rightarrow$ 
    List UnivRule  $\rightarrow$ 
    List  $\exists$ rule  $\rightarrow$ 

```

```

List ElimRule →
List  $\beta$ -rule  →
List  $\eta$ -Rule  →
RuleSet

```

Firstly, we present three functions, one for checking each of the user-provided typing rules. Each definition proceeds in the same way, attempting to match an appropriate rule, if a match is found, the rule is run. If no rule is found to match, and there are no rules left to check, these functions fail with a descriptive error.

```

 $\exists$ -check  : Context  $\gamma$           →
           RuleSet                →
           (term type : Const  $\gamma$ ) →
           Failable  $\top$ 

type-check : Context  $\gamma$           →
           RuleSet                →
           (type : Const  $\gamma$ ) →
           Failable  $\top$ 

elim-synth : Context  $\gamma$           →
           RuleSet                →
           (target-type eliminator : Const  $\gamma$ ) →
           Failable (Const  $\gamma$ )

```

We define a notion of term equivalence that we will need later to check the type of computations embedded in constructions. This is a purely syntactic action. If terms are to have their normal forms compared then they must be normalised before checking their equivalence here.

```

 $\equiv_v$  : Var  $\gamma \rightarrow$  Var  $\gamma \rightarrow$  Failable  $\top$ 
 $\equiv^t$  : Term  $d \gamma \rightarrow$  Term  $d \gamma \rightarrow$  Failable  $\top$ 

```

The high-level functionality of our type checker is given by two functions, check and infer. We check the type of constructions and infer the type of computations, however since we can embed computations in constructions using a thunk, we may accept a call to check the type of any term.

When checking constructions, we make a distinction between two cases, either we duck under a thunk to check the computation beneath it, or we delegate immediately to \exists -check. When checking computations under a thunk, we always start by synthesizing the type with a call to *infer* safe in the knowledge that if it succeeds, we are guaranteed that what is given is a type. We then check equivalence between what was inferred, and the type we are checking after normalising each of them.

When inferring the type of computations, there are three cases that we consider separately. The types of variables are looked up in the context. Terms with type annotations are checked to determine that the annotation is indeed the type of the term and if this is so then

the normalised annotation is returned. Lastly, the types of eliminations are inferred by synthesizing the type of the target before delegating to *elim-synth* so that it might determine to the appropriate elimination typing rule to run. Any term given back in this process is checked to be a type before *infer* returns it.

```

check : RuleSet      →
      Context  $\gamma$   →
      (type : Const  $\gamma$ ) →
      (term : Term  $d \gamma$ ) →
      Failable  $\top$ 

infer  : RuleSet      →
      Context  $\gamma$   →
      (term : Compu  $\gamma$ ) →
      Failable (Const  $\gamma$ )

```

Before we discuss the running of rule instances, we must first describe how to check a premise chain. The result of matching against a given rule provides us with the necessary environments for us to resolve the schematic variables that might exist in a premise. In our implementation, checking the premises turns out to be mostly trivial, if verbose. Each data constructor of a Premise resolves to calling a single function that we have already provided (such as *type-check*, *check* or *__≡^t__*) before building the required environments for what is newly trusted and what remains to be trusted.

We provide just the definition for the ' \in ' premise by way of example here to illustrate how the concrete terms are realised from expressions and schematic variables before we delegate to *check*.

```

check-premise : Context  $\gamma$       →
              RuleSet          →
               $p$ -Env  $\rightarrow q$ -Env →
              Prem  $p q \gamma p' q'$  →
              Failable ( $p'$ -Env  $\times q'$ -Env)

check-premise  $\Gamma$  rules  $penv qenv$  ( $T \ni \xi$  [ $\theta$ ])
= do
  _ ← check rules  $\Gamma$  (toTerm  $penv T$ ) (( $\xi$  !!  $qenv$ ) (term  $\theta$ )
  succeed ((thing ( $\xi$  !!  $qenv$ )) , ( $qenv$ - $penv$   $\xi$ ))

```

Checking a whole chain of premise proceeds as one might expect; each premise is checked in order. The environments for the things we trust accumulate while the environments for the things that remain to be trusted are whittled away. If the premise chain is empty, we simply return what we already trust.

In direct correspondence to our initial three functions that check the various user-defined rules, we provide three more, one for running each type of rule should such a matching rule be found. We make sure to address the appropriate conditions here, such as checking that the type in a \ni rule is indeed a type.

We give two examples here, that of running a \ni rule and that of running an elimination rule. The first merely tries to succeed, the second must return the synthesised type.

We do not request environments for the exact patterns in a rule, but instead for their opening into the current scope. Similarly, when we retrieve the premise chain from a rule, we open this to the current scope before checking it. This method allows us to mandate that rules are defined in the empty scope while allowing them to be applied in any scope.

```

run- $\exists$ rule : Context  $\gamma$            $\rightarrow$ 
              RuleSet              $\rightarrow$ 
              (rule :  $\exists$ rule)       $\rightarrow$ 
              (( $\gamma \otimes$  (input rule)) -Env)  $\rightarrow$ 
              (( $\gamma \otimes$  (subject rule)) -Env)  $\rightarrow$ 
              Failable  $\top$ 
run- $\exists$ rule { $\gamma$ }  $\Gamma$  rules  $\exists$ -rule input-env subject-env
= do
  ty  $\leftarrow$  return (termFrom (input  $\exists$ -rule) input-env)
  _  $\leftarrow$  type-check  $\Gamma$  rules ty
  chain  $\leftarrow$  return ( $\otimes$ premises  $\gamma$  (proj2 (premises  $\exists$ -rule)))
  _  $\leftarrow$  check-premise-chain rules  $\Gamma$  input-env subject-env chain
  succeed tt

run-erule : Context  $\gamma$            $\rightarrow$ 
              RuleSet              $\rightarrow$ 
              (rule : ElimRule)     $\rightarrow$ 
              ( $\gamma \otimes$  targetPat rule) -Env  $\rightarrow$ 
              ( $\gamma \otimes$  eliminator rule) -Env  $\rightarrow$ 
              Failable (Const  $\gamma$ )
run-erule { $\gamma$ }  $\Gamma$  rules elim-rule input-env subject-env
= do
  chain  $\leftarrow$  return ( $\otimes$ premises  $\gamma$  (proj2 (premises elim-rule)))
  p'env  $\leftarrow$  check-premise-chain rules  $\Gamma$  input-env subject-env chain
  elim-ty  $\leftarrow$  return (toTerm p'env (output elim-rule))
  _  $\leftarrow$  type-check  $\Gamma$  rules elim-ty
  succeed elim-ty

```

Our type checking process is a highly mutually recursive process by nature. We can leverage Agda's termination checker to give us feedback that we might use to guarantee that the process terminates. We are unable to guarantee termination in this case as we do not impose the required restrictions on user-supplied rules. If a user supplies a rule containing a circular check, then the type checking process will not halt. We talk more about this limitation later in our evaluation of the software.

6.16 Parser combinators

We acknowledge that parsing is an entire field of study in its own right and that it is certainly not the focus of this project. However, to have the user describe the type system in our DSL rather than in Agda code, we will require a parser to make sense of this input.

We capitalise on the definitions in the Agda standard library and define a parser in terms of the State monad transformer. A parser is given a string as input, then it may fail or it may succeed and return something along with the rest of the string minus what was consumed

during parsing. For convenience, we also provide a similar type that describes a parser operating on lists of characters and a way to plumb this into a 'real' parser by making the appropriate conversions.

```

module Parser where
  Parser' : Set → Set
  Parser' = StateT (List Char) Maybe

  Parser : Set → Set
  Parser = StateT String Maybe

  →[_]→ : Parser' A → Parser A
  →[ P' ]→ str = do
    (a , rest) ← (P' ∘ toList) str
    just ((a , fromList rest))
  where open maybem Monad

```

We build a small library of useful parsers, allowing us to build parsers incrementally using these parser combinators. We provide parsers that parse conditionally, parse zero-or-more times. Parsers that fail if they do not consume all their input, parsers to parse literal characters and strings. The list is extensive so we will not detail these parsers here. A full list of the basic combinators used is available in appendix .3.

6.17 Parsing the DSL

Since we have modelled typing rules, β -rules and η -rules explicitly in our software, parsing the DSL becomes a reasonably straightforward process. We use our combinators to build parsers for patterns, expressions, premises, and premise chains before assembling these to parse type rules, elimination rules and other such rules we will require before finally combining these to parse the overall description of a type. Parsing a specification file is then a matter of parsing one or more types. The result of parsing a specification file successfully is a set of rules.

6.18 Parsing the language

We now want the user to be able to supply a candidate for type checking by writing the term in a file as opposed to supplying it as an Agda code representation in our internal syntax and so we build a second parser for this purpose.

We use the patterns in our type rules and \exists rules to determine valid constructions in the language and then use these patterns to guide our parsing, however, we must first overcome one major problem.

A user is perfectly able to have some *place* as the first part of a pattern. Attempting to parse this in a top-down manner may lead to infinite recursion in the same way that parsing a left-recursive grammar might. We can solve this problem in two ways, we could either implement a more sophisticated parser that is tolerant of such grammars or we could pre-process the patterns to create a right-recursive equivalent. We consider the first option to be outside the scope of this project and opt for the latter solution.

Since our objective is to synthesize some type, the top-level parsable element here is a single *computation*.

6.19 Putting it all together

We now have all the required components for our type-checker. A user runs the software with the command

```
TypeCheck <spec-file> <term-file>
```

We first read the whole of the spec-file as a string, and parse the rules from it with our DSL parser. Armed with the set of rules we can then parse a term in the language from the term-file, and then attempt to type the term with a simple call to 'infer'.

Chapter 7

Testing and verification

7.1 Type-led verification

Throughout the process of building the software, we have tried to ensure we leverage Agda's expressive type system to maintain invariants that allow us to forgo large swathes of testing that would otherwise be required. This approach carries benefits, however, there are limitations to this method and it is often not practical to rely on capturing such invariants in all cases.

Firstly we note that we are explicit about scope throughout the code. We formalise the notion of something being *Scoped* early on and make liberal use of this type throughout. Variables are defined to be well-scoped by construction allowing us to ensure that the scoped entities that use such variables (like terms and expressions) are guaranteed to be well-scoped and as such we manage to eliminate a huge testing space where various functions would have otherwise had to deal with potential "out of scope" errors. Similarly, we can use scope to mandate that we are given a context sufficient for use with a particular scoped term and, more generally, ensure that we are explicit about the scope that we expect, and the scope that we are returning.

Another area where we make careful use of types is that of patterns and environments. The way that an environment is constructed relative to some pattern that indexes it, ensures that it is well-formed according to the pattern in question. This also gives us the power to request such a well-formed environment, simultaneously excluding the need to test for behaviour on ill-formed environments while having the effect of guaranteeing a sensible output where we might otherwise have had to employ a *Maybe* result. Schematic variables get similar treatment, allowing us to provide a function to look up the term referred to by these variables in a manner that cannot fail.

The benefit of the careful use of types here has far-reaching consequences in our software. By way of example let us look at the type of our *toTerm* function which builds a term from an expression.

$$\text{toTerm} : (\gamma \otimes p) \rightarrow \text{Env} \rightarrow \text{Expr } p \text{ d } \gamma' \rightarrow \text{Term } d (\gamma + \gamma')$$

First, note how we can ensure that we get the correct kind of environment for the expression by sharing the same pattern *p* in the environment and expression types. We ensure we get the correct kind of term, either a construction or a computation, by sharing the same

directionality d between the expression type and term type. Finally, we manage the scope explicitly, ensuring that for an expression in some scope γ' and some pattern (in the empty scope although you cannot see that explicitly in the type here), which has been opened in γ results in a term scoped in $(\gamma + \gamma')$. Agda affords us the opportunity to conduct a lot of verification in this manner and we make sure to capitalise on this through the project.

Another example where could expose our type-led verification is in that of premise chains described in section 6.10. We give a type, *Placeless*, which is indexed by a pattern and can encode the information in the pattern if and only if the pattern contains no places. We then ensure that the only way to end a chain of premises is to give some proof of the placelessness of what is left to trust. This eliminates a vast amount of potential validation code and associated tests. It allows us to enforce this aspect of the well-formedness at the type level rather than having to test how the software might deal with various failures resulting from an insufficient premise chain.

```
data _Placeless {γ : Scope} : Pattern γ → Set where
  '      : (s : String) → 's Placeless
  _•_    : {l r : Pattern γ} → (l Placeless) → (r Placeless) → (l • r) Placeless
  bind   : {t : Pattern (suc γ)} → (t Placeless) → (bind t) Placeless
  ⊥      : ⊥ Placeless

data Prefs (p0 q0 : Pattern γ) : (p2 : Pattern γ) → Set where
  ε      : (q0 Placeless) → Prefs p0 q0 p0
  _⇒_    : Prem p0 q0 γ pg1' →
           Prefs (p0 • pg1) q1' p2' →
           Prefs p0 q0 p2'
```

We have given some examples of where we have used the type system to verify important properties of our code. This is a technique we used extensively when constructing the software and we consider it our primary method of ensuring a good level of correctness.

7.2 Where type-led verification is insufficient

We must provide tests for aspects of the system that depend on user-supplied instantiations of the concepts and rules we have defined. Especially where the results depend on several parts of the system working together. Using the type system to verify software is more suited to checking the software at a lower level by verifying smaller component parts. The invariants required when we start to combine these parts are often large, complex and overly difficult to capture in a type and so these aspects of the system are tested in a more traditional manner.

When writing this software, we include a suite of tests that cover areas such as β -reduction, normalisation, η -expansion, type checking and various aspects of parsing (an area where we focused less on type-led verification). A larger example of the testing we conducted is available in appendix .4.

Finally, we tested our software using the three specifications in appendix .2 that we constructed following the design of our DSL in chapter 5 to ensure that the software met the minimum acceptance criteria outlined in section 4.1.6.

Chapter 8

Evaluation

8.1 DSL evaluation

In section 5.8 we identify three common problems with the way type systems are currently described in the literature.

1. Descriptions are often jargon-heavy.
2. Descriptions are often lengthy with repeated information.
3. The separated nature of the rules make dependencies non-explicit.

We aimed to address these issues when designing our DSL while also prioritising readability, in particular aiming for type definitions to be readable in a linear fashion.

We believe that in section 5.8 we demonstrated the concrete ways in which we addressed the identified problems, although there were some compromises to be made.

To aid readability, we chose keywords carefully in our DSL, favouring hyphenated representations of phrases such as "eliminated-by" and "resulting-in-type". This decision has the dual purpose of allowing the descriptions to be read in a manner somewhat like reading a description in English while also reducing some jargon, however, this can sometimes be at the cost of addressing the second identified problem: brevity. We feel that the gains in readability outway the cost in brevity here, and that the brevity is far better addressed by the use of hierarchical structure than by choosing shorter keywords.

We were careful with our design to ensure flexibility with regards to what type systems were describable. In appendix .2.1 we show a complete example of a variation of simply typed lambda calculus with two base types and product types. In appendix .2.2 we demonstrate how we can encode arbitrary polymorphic terms by wrapping them in a value that is first eliminated by a type. Finally, in .2.3 we demonstrate that the DSL is readily used for dependent types by describing singleton types and dependent functions. The inclusion of a dependently typed example allows us to discharge the acceptance criteria where we specified that our solution must be capable of representing both non-dependent and dependent types.

Some aspects of the DSL are worth further thought as there are improvements to be had.

Firstly, the dependency on consistent ordering between elimination type descriptions and corresponding reductions in each value is not ideal. We identified non-explicit dependencies

as an issue we wished to address and this is a prime example of such a dependency.

Another area worth our consideration is the notable absence of any way to describe user-defined datatypes. This would certainly need to be addressed in some form before the DSL was useful in any real-world application.

A final limitation of our DSL is that when we introduce a new variable binding, the scope of that new binding begins immediately after the variable is introduced. There is no way for us to specify an alternative location to begin the newly weakened scope. Consequently, we are unable to represent let-expressions in the customary manner, as the variable name appears before the value we then bind to it meaning that we provide the new value in the already weakened scope. I.e. for

```
let x = <some-term> in <some expression involving x>
```

then `x` is in-scope when defining the term we plan to associate to it. This problem appears more generally any time we wish to provide a syntax for some kind of assignment in this manner. We can work around this issue by simply providing the term before the binding name, however, this results in unintuitive syntax, opposing decades of convention.

```
let <some-term> = x in <some expression involving x>
```

8.2 Software evaluation

We open our evaluation of the software by first revisiting the acceptance criteria provided in section 4.1.6. We have already shown that the software can be run in the manner described by the first criterion.

We overachieve on the second criterion by including somewhat more helpful output that we originally considered in-scope in section 4.1.4. The program output details the number of each kind of rule it was able to parse from the specification file rather than just a simple success or failure message. Similarly, the program output does not simply acknowledge whether it managed to parse the source file but shows the user a representation of what it managed to parse along with any remaining input it was unable to parse. Although we only committed to a success or failure message resulting from the type-checking process, we instead supply the user with the type of the term provided in the source file. By providing this functionality over and above what we committed to, we greatly improve the experience in using the software, especially where users can see various rules accumulate as they write their type-system description.

We have described in chapter 7 the test cases that evidence our efforts in meeting the third acceptance criterion.

Lastly, appendix .2 shows the specifications that we used to test our system supports both non-dependant and dependent types discharging our final acceptance criterion.

As well as achieving our acceptance criteria, another area where we believe this project succeeds is in the translation of various areas of meta-theory to Agda code. We manage to capture many useful invariants in our careful use of types as discussed in the previous chapter. This helped to ensure consistent code quality throughout the lifetime of the project.

We have, so far, detailed some of our success in this project, but there are several areas where we now identify shortcomings and suggest potential improvements.

8.2.1 Forced explicit type information

Although there are a variety of languages we can type-check using this software, there are also certain notable limitations. We are limited by what we can describe using the DSL as is discussed in the previous section. We also require users' source code to be extremely explicit about type information. There are no appeals to techniques such as unification to shorten the syntax of the languages we describe. For example, when we eliminate some term in our end syntax, that term cannot be a construction, it must be a computation with a synthesizable type. This is true even if the elimination itself is type-annotated and so we might be able to make appeals to 'magic' to provide in this missing type information.

8.2.2 Limited DSL validation

There are areas where we conduct sensible checks when parsing the DSL to protect the user from their own errors. We make sure that when a user supplies some pattern that we only use it in particular rules if the premises can establish complete trust. However, there are other areas where we do not conduct checks and this leaves room for user errors in our DSL.

Most notably, users can construct types that lead to circular checks and therefore lead to a type-checking process that we cannot guarantee to terminate. The following description is perfectly legal in our DSL and our software will parse and attempt to use such information.

```
type: A
  if:
    type A
```

Further testing with this example specification exposes another limitation of our system which we will now explore.

8.2.3 Rule matching depends on rule declaration order

When we collect together rules of a certain type, we make no checks to ensure that some syntax can only match zero or one of the rules. We will discuss this further when we explore missed opportunities for mathematical structure, however, what we expose here is that where multiple rules might match, the rule we use is the one that is declared first. As a result, having some erroneous, but very generic, entity declared early in the file might "catch" matches before they reach their intended rules and such a problem could be difficult to debug.

A fundamental decision that led to this problem was that of storing the collections of rules as a list. The arbitrary ordering of rules in a list does not admit any structure as to make an alternative method of processing obvious.

In hindsight, we propose a tree-like structure where rules are stored at leaves and nodes store the most specific pattern that matches all the rules stored in its subtrees. A node might either yield children for all sub-patterns giving rise to a collection of more specific patterns, or a *single* leaf containing a rule that matches on the pattern. In this way, we might eliminate the problem that some syntax might match multiple patterns (since a node cannot contain both a leaf and a set of child nodes) while also eliminating the arbitrary dependency of matching on the order that certain elements are declared.

8.2.4 Missed opportunities for mathematical structure

We missed several opportunities to capitalise on mathematical structure when writing our code. We noted previously that we might have altered our definition of *Pattern* to expose some applicative functor structure which would have been a welcome addition to later definitions. We are also aware that patterns as they are defined presently form a lattice. A partial order on the set of patterns may be defined.

let $\mu(p)$ = the set of all terms that match the pattern p

$$p \leq q \iff \mu(p) \subseteq \mu(q)$$

We give algorithms for calculating the meet and join of a set of patterns in appendix .5 completing the definition of a lattice. We might have then used this structure to validate user-supplied typing rules by ensuring that a piece of syntax describing a type cannot match more than one rule. For each new user-supplied type pattern t and the set of all existing type patterns T this would be equivalent to ensuring

$$\forall P \in \{t\} \times T, \mu(\text{meet } P) = \emptyset$$

which is easily calculable by checking whether $(\text{meet } P)$ contains any \perp s.

There are also areas where we might have used this structure to help us in parsing the source code as we rely heavily on parsing according to patterns.

8.2.5 Single universe limitation

To simplify the late-stage implementation of the software, we decided that all languages described would consist of a single universe of types. We call this universe "set" in our syntax and we may use the universe in any place where a type might be expected.

While there exist some parts of the implementation that we wrote while intending for a user to describe the universes in our DSL too, this was not a feature we ended up carrying through to the final system due to time constraints.

We might have thought to include the ability for the user to choose the name of the universe. This might have been easy to implement and could have been a welcome addition to the functionality we have provided.

8.2.6 Code quality

There are areas of the project where we might find ways to improve the quality of the code. Particularly in sections 6.14 and 6.15 we notice that many functions take long lists of arguments. This is a known anti-pattern, indicating that our code is suboptimal. There are improvements to be had by revisiting some sections of the code and collecting commonly associated information under some sensible types, and perhaps more generally reconsidering where some of these definitions are defined to remove the need for so many arguments. For example, the function for running a particular rule would be better defined inside the record type of the rule itself. Another concrete example to be had is that Context and Ruleset are frequently supplied together in function definitions, it may have been easier to collect this information under some type.

8.2.7 Documentation

We addressed our approach to documentation early in the planning stage of the project, conveying our use of literate Agda files to produce documented source code files as we work on our implementation. While we still believe this was a sensible choice for this project there are downsides to having the documentation structured in this way.

Firstly, if particular information is required from the documentation this is not readily available since the documentation is not available pre-compiled and indexed in a particular format. As a result, finding information would be a slow process unless the person was familiar with the codebase. Secondly, because of the amount of documentation interspersed with code in the same file, the maintenance of these files could become cumbersome. When making any alterations to the code we must also make sure that we update the documentation otherwise we might end up with a file that contains information that is out of date even with regards to itself. This is as opposed to the scenario where a separate document might end up slightly behind the most up-to-date versions of the source code, here the problem is lessened since we might be able to determine its worthiness by cross-checking the date of the latest revision of each document.

The benefit of choosing this style of documentation despite these problems is that we have a more 'lightweight' development process that is better suited to smaller projects and shorter timeframes, especially when we are not concerned that someone else may have to maintain this code in future.

8.3 Personal performance evaluation

In general, the project unfolded as detailed in the project plan. We deliver the artefacts outlined in section 4.1.3 and we hit the milestones outlined in section 4.1.5. We devoted much of the first semester to reading and research and started to write the software during the Christmas break, completing the software in early March and leaving some time for testing and verification.

During the implementation phase, we did not hit the target dates chosen for the completion of the software and the completion of the testing. We found the project to be substantially more difficult than we anticipated and found ourselves lacking key skills that would have helped us to meet the deadlines we set originally. This has resulted in a slight decrease in code quality that is apparent in some of the later areas of the codebase.

Firstly, we over-estimated our dependently-typed programming skills and, being somewhat new to this typing paradigm, progress was naturally slower as thought was given as to how best to use the features it affords. Secondly, we found ourselves lacking experience in the design of functional programs. We certainly had experience in providing individual functions, however, we had very little experience of functional design at a higher level. We often found ourselves having to rewrite sections of code that seemed sufficient at the time but that made subsequent definitions prohibitively difficult.

Two possible solutions could have been employed here to make better use of the time that was available. Firstly we might have been more careful about reducing scope earlier in the project. While we did not over commit in defining exactly what we would deliver, we initially included some optional areas that we might explore if time allowed. This led us to conduct a broader range of background research that was required when it might have been more

useful to regain some of that time to use in the implementation and testing of the software. Secondly, we might have considered writing the software in a language in which we were already familiar.

Although we do detail the above points as potential ways in which we could have improved certain aspects of the project, we believe that we were able to learn new skills by writing the project in a dependently typed function language despite being less familiar with this paradigm. These skills would have been more difficult to learn in other paradigms and we do not regret our decision.

We have discussed many issues in our evaluation, however, these issues did not necessitate any compromise on the features we committed to delivering and our final artefacts conform fully with our acceptance criteria.

Chapter 9

Conclusion

9.1 Summary

This project was concerned with type checking code according to descriptions of type systems. We began by conducting a range of background research in the field of type theory. We then expanded our research to include various topics that might have helped us with the construction of a type-checker. We designed a DSL that was capable of describing type systems. We then wrote a type checker that was able to read such a description from a file, generate the type checker, then type-check code in the language it described.

We believe that we can represent a broad range of type-systems using our DSL and type-check all of these with the software we provide. We also explore areas where both our DSL and our software fall short. We finish by giving a qualitative evaluation of the DSL, the software and our personal performance during the development process.

9.2 Future work

There are many areas that could provide an excellent opportunity to expand on this work.

One area of particular interest would be to explore the introduction of Hindley-Milner style type inference. There are areas where such inference is impossible in a dependently typed setting but is it such a black and white distinction? What exactly would a user have to forgo to enjoy the convenience of this inference, and could they choose to allow it in only certain places, yet still use dependent types in others? This would be an excellent area for exploration and it is not clear to us how it could be used effectively for user-specified type systems.

Another area of interest would be to consider an extension of the DSL syntax to allow subtyping. Is there a sensible level of abstraction that could be introduced here to allow the user to specify the mechanics of subtyping in a language while being prohibitive enough to exclude some amount of nonsense that might be provided? This is certainly worth further thought.

Finally, we might explore how such a system as this might be combined with a compiler-generator so that we might give specifications of entire languages and get a fully-fledged compiler with static analysis.

9.3 Conclusion

We are pleased with the outcome of this project.

We were able to accumulate a diverse range of knowledge and gained a valuable insight into the foundations of type-theory while fostering the opportunity to further develop our functional programming skills.

This project helped to familiarise us with the reading of academic literature and taught us the value of being precise in our definitions. We learned that being realistic is easier but being ambitious is more enjoyable. This project certainly proved to be a major challenge, one which we are glad that we undertook.

9.4 Acknowledgements

I would like to take this opportunity to thank Dr Conor McBride for his patient, precise explanations, his reassurance and guidance and for being so generous with his time.

I would also like to thank Dr Clemens Kupke for his feedback following the progress presentation.

Appendices

.1 Typing rules

.1.1 System F

$$\begin{array}{c}
 \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \text{ var} \qquad \frac{}{\Gamma \vdash c : T} \text{ const} \\
 \\
 \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x_\sigma . e) : \sigma \rightarrow \tau} \text{ abs} \qquad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1 e_2) : \tau} \text{ app}
 \end{array}$$

Thus far the typing rules are the same as in the simply typed lambda calculus, to complete his system Reynolds extends it with two more rules to introduce parametric polymorphism:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash (\Lambda \alpha . M) : \Delta \alpha . \sigma} \Delta\text{-abs} \qquad \frac{\Gamma \vdash M : \Delta \alpha . \sigma}{\Gamma \vdash (M \tau) : \sigma[\tau/\alpha]} \Delta\text{-app}$$

Note that in these rules, α is a type variable.

.1.2 Hindley-Milner

In Hindley-Milner, we have very similar rules for typing lambda abstraction, application, free type variables and type constants:

$$\begin{array}{c}
 \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \text{ var} \qquad \frac{}{\Gamma \vdash c : T} \text{ const} \\
 \\
 \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x . e) : \sigma \rightarrow \tau} \text{ abs} \qquad \frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1 e_2) : \tau} \text{ app}
 \end{array}$$

These rules are then extended to accomodate the 'let' language construct:

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash (\text{let } x = e_1 \text{ in } e_2) : \tau} \text{ let}$$

Before we detail the last two rules detailing instantiation and generification we first outline the meaning of a judgement $\sigma \sqsubseteq \sigma'$. Intuitively this means that σ is some subtype of σ' - we can create it by some substitution of the quantified variables in σ' . More precisely it is defined as follows:

$$\frac{\tau' = \{\alpha_i \mapsto \tau_i\} \tau \quad \beta_i \notin \text{free}(\forall \alpha_1 \dots \forall \alpha_n \cdot \tau)}{\forall \alpha_1 \dots \forall \alpha_n \cdot \tau \sqsubseteq \forall \beta_1 \dots \forall \beta_m \cdot \tau'} \text{ spec}$$

Lastly we have rules to instantiate a type scheme, or generify a type (i.e. to make a type more specific and narrow, or less specific and general):

$$\frac{\Gamma \vdash e : \sigma \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma'} \text{ inst} \quad \frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall \alpha \sigma} \text{ gen}$$

The latter three rules are the ones that capture the idea of Hindley-Milner polymorphism.

.2 Example specifications

This appendix shows example specification files for various languages.

.2.1 Simply typed lambda calculus with product types

```
type:  $\alpha$ 
value: a
```

```
type:  $\beta$ 
value: b
```

```
type: A  $\rightarrow$  B
if:
  type A
  type B
  eliminated-by: E
  if:
    (A)  $\leftarrow$  E
    resulting-in-type: B
value: \ X.  $\rightarrow$  M
if:
  X : (A)  $\mid$ - (B)  $\leftarrow$  M
  reduces-to: M/[ , E:A]
expanded-by: \ Y.  $\rightarrow$  Y
```

```
type: A x B
if:
  type A
  type B
  eliminated-by: fst
  resulting-in-type: A
  eliminated-by: snd
  resulting-in-type: B
value: L and R
if:
  (A)  $\leftarrow$  L
  (B)  $\leftarrow$  R
  reduces-to: L
  reduces-to: R
expanded-by: fst , snd
```

.2.2 System-F-like language

Our variation is slightly more general than the original encoding as opposed to only polymorphic functions we give the means of representing arbitrary polymorphic types.

```
type:  $\alpha$ 
  value: a
```

```
type:  $\beta$ 
  value: b
```

```
type: A -> B
  if:
    type A
    type B
  eliminated-by: E
    if:
      (A) <- E
    resulting-in-type: B
  value: \ X. -> M
    if:
      X : (A) |- (B) <- M
    reduces-to: M/[ , E:A]
  expanded-by: \ Y. -> Y
```

```
type:  $\forall$  T. => M
  if:
    T : (set) |- type M
  eliminated-by: TY
    if:
      type TY
    resulting-in-type: M/[ , TY:set]
  value:  $\delta$  T. PTY
    if:
      T : (set) |- (M/[ , .T]) <- PTY
    reduces-to: PTY/[ , TY:set]
```

.2.3 Lambda calculus variation with dependent types

```
type:  $\alpha$ 
  value: a1
  value: a2

type: isa TY TM
  if:
    type TY
    (TY) <- TM
  value: is V
  if:
    (TY) <- V
    (TM) = V

type: A X. -> B
  if:
    type A
    X : (A) |- type B
  eliminated-by: E
  if:
    (A) <- E
    resulting-in-type: B/[ , E:A]
  value: \ X. -> M
  if:
    X : (A) |- (B/[ , .X]) <- M
    reduces-to: M/[ , E:A]
  expanded-by: \ Y. -> Y
```


.3 Parser combinators

```
fail : Parser A
fail str = nothing

-- guarantees a parser, if it succeeds, consumes some input
safe : Parser A → Parser A
safe p s = do
    (a , leftover) ← p s
    if length leftover <b length s then just (a , leftover) else nothing
    where open maybemonad

peak : Parser Char
peak str with toList str
... | []      = nothing
... | c :: rest = just ((c , fromList (c :: rest)))

all : Parser String
all = λ s → just (s , "")

try : Parser A → A → Parser A
try p a str = p str <> just (a , str)

consumes : Parser A → Parser (ℕ × A)
consumes p = do
    bfr ← (λ s → just (length s , s))
    a ← p
    afr ← (λ s → just (length s , s))
    return ( bfr - afr , a)
    where open parsermonad

biggest-of_and_ : Parser A → Parser A → Parser A
biggest-of_and_ p1 p2 str with p1 str | p2 str
... | nothing | nothing = nothing
... | nothing | just p  = just p
... | just p   | nothing = just p
... | just (a1 , rst1) | just (a2 , rst2) = just (if length rst1 <b length rst2 then (a1

_or_ : Parser A → Parser B → Parser (A ⊕ B)
(pa or pb) str = (inj1 <$> pa) str <> (inj2 <$> pb) str
    where
        open parsermonad

either_or_ : Parser A → Parser A → Parser A
(either pa or pb) str = maybe' just (pb str) (pa str)

ifp_then_else_ : Parser A → Parser B → Parser B → Parser B
(ifp p then pthen else pelse) str with p str
... | nothing = pelse str
```

```

... | just x = pthen str

nout : Parser T
nout = return tt
  where open parsermonad

optional : Parser A → Parser (A ⊔ T)
optional = _or nout

complete : Parser A → Parser A
complete p = do
  a ← p
  rest ← all
  if rest == "" then return a else fail
  where open parsermonad

takeIfc : (Char → Bool) → Parserp Char
takeIfc p [] = nothing
takeIfc p (c :: chars) = if p c then just (c , chars) else nothing

takeIf : (Char → Bool) → Parser Char
takeIf p = →[ takeIfc p ]→

-- This terminates as we ensure to make p "safe" before we
-- use it, forcing the parser to fail if it does not consume
-- any input
{-# TERMINATING #-}
_+[_,_] : Parser A → (A → B → B) → B → Parser B
p *[ f , b ] = do
  inj1 a ← optional (safe p)
  where inj2 _ → return b
  b ← p *[ f , b ]
  return (f a b)
  where open parsermonad

_+[_,_] : Parser A → (A → B → B) → B → Parser B
p ^+[ f , b ] = pure f p (p *[ f , b ])
  where open parsermonad

anyof : List (Parser A) → Parser A
anyof [] = fail
anyof (p :: ps) = either p or (anyof ps)

biggest-consumer : List (Parser A) → Parser A
biggest-consumer [] = fail
biggest-consumer (p :: ps) = biggest-of p and biggest-consumer ps

all-of : List (Parser A) → Parser (List A)

```

```

all-of [] str = just ([], str)
all-of ps str = just (foldr ( $\lambda$  p las  $\rightarrow$  maybe' ( $\lambda$  (a , _)  $\rightarrow$  a :: las) las (p str) ) [] ps

{-# TERMINATING #-}
how-many? : Parser A  $\rightarrow$  Parser ( $\Sigma$ [ n  $\in$   $\mathbb{N}$  ] Vec A n)
how-many? p = ifp p then (do
    a  $\leftarrow$  safe p
    (n , as)  $\leftarrow$  how-many? p
    return (N.suc n , a :: as))
  else return (0 , [])
  where open parsermonad

max_how-many? :  $\mathbb{N}$   $\rightarrow$  Parser A  $\rightarrow$  Parser ( $\Sigma$ [ n  $\in$   $\mathbb{N}$  ] Vec A n)
max zero how-many? _ = return (0 , [])
  where open parsermonad
max (suc n) how-many? p = ifp p then (do
    a  $\leftarrow$  safe p
    (n , as)  $\leftarrow$  max n how-many? p
    return (N.suc n , a :: as))
  else return (0 , [])
  where open parsermonad

exactly : (n :  $\mathbb{N}$ )  $\rightarrow$  Parser A  $\rightarrow$  Parser (Vec A n)
exactly zero _ = return []
  where open parsermonad
exactly (suc n) p = do
    a  $\leftarrow$  p
    as  $\leftarrow$  exactly n p
    return (a :: as)

-- text/number parsing

whitespace : Parser  $\top$ 
whitespace str = just (tt , trim $\leftarrow$  str)

ws+nl : Parser  $\top$ 
ws+nl str = just (tt , trim $\leftarrow$ p str)

ws+nl! : Parser  $\top$ 
ws+nl! = do
    c  $\leftarrow$  takeIf Data.Char.isSpace
    ws+nl
  where open parsermonad

ws-tolerant : Parser A  $\rightarrow$  Parser A
ws-tolerant p = do
    whitespace
    r  $\leftarrow$  p
    whitespace
    return r

```

```

where open parsermonad

wsnl-tolerant : Parser A → Parser A
wsnl-tolerant p = do
    ws+nl
    r ← p
    ws+nl
    return r
where open parsermonad

wsnl-tolerant! : Parser A → Parser A
wsnl-tolerant! p = do
    ws+nl!
    r ← p
    ws+nl!
    return r
where open parsermonad

literalc : Char → Parserp Char
literalc c [] = nothing
literalc c (x :: rest)
    = if c == x then just (c , rest)
      else nothing

literal : Char → Parser Char
literal c = →[ literalc c ]→

newline = literal "\n"

nextCharc : Parserp Char
nextCharc [] = nothing
nextCharc (c :: chars) = just (c , chars)

nextChar = →[ nextCharc ]→

literalAsString = (fromChar <$>_) ∘' literal
where open parsermonad

string : String → Parser String
string str = foldr
    (λ c p → (pure _++_) (literalAsString c) p)
    (return "") (toList str)
where open parsermonad

stringof : (Char → Bool) → Parser String
stringof p = takeIf p *[_++_ ∘ fromChar , "" ]

until : (Char → Bool) → Parser String

```

```

until p = stringof (not ∘ p)

nonempty : Parser String → Parser String
nonempty p = do
    r ← p
    if r == "" then (λ _ → nothing) else return r
    where open parsermonad

nat : Parser ℕ
nat = do
    d ← nonempty (stringof isDigit)
    return (toNat d)
    where open parsermonad

bracketed : Parser A → Parser A
bracketed p = do
    literal "("
    a ← wsnl-tolerant p
    literal ")"
    return a
    where open parsermonad

potentially-bracketed : Parser A → Parser A
potentially-bracketed p = either bracketed p or p

curlybracketed : Parser A → Parser A
curlybracketed p = do
    literal "{"
    a ← wsnl-tolerant p
    literal "}"
    return a
    where open parsermonad

squarebracketed : Parser A → Parser A
squarebracketed p = do
    literal "["
    a ← wsnl-tolerant p
    literal "]"
    return a
    where open parsermonad
open Parsers

```

.4 A selection of testing code

.4.1 Description of STLC

```
open import CoreLanguage
```

```

open import Data.Nat using (suc)
open import Pattern using (Pattern; ';; place; bind; _•_; ★; _•; •_; svar)
open import Expression using (_/_; ';; _•_; _::_)
open import Rules using (ElimRule; TypeRule; UnivRule; ∃rule; ε; _placeless; type;
    _⇒_; _⊢_; _∃' _[]; ')
open import Thinning using (Ø; _O; ι)
open import BwdVec using (ε)
open import Data.Product using (_,_)
open import TypeChecker using (RuleSet; rs)
open import Semantics renaming (β-rule to β-Rule)
open import EtaRule using (η-Rule)
open import BwdVec
open β-Rule
open ElimRule
open TypeRule
open UnivRule
open ∃rule
open η-Rule

```

We begin by introducing some combinators to construct our language terms as creating these terms directly in our internal language can be tedious.

```

module combinators where
  α : ∀{γ} → Term const γ
  α = ' "α"

  a : ∀{γ} → Term const γ
  a = ' "a"

  β : ∀{γ} → Term const γ
  β = ' "β"

  b : ∀{γ} → Term const γ
  b = ' "b"

  ___ : ∀{γ} → Const γ → Const γ → Term const γ
  x y = x • ((' "→") • y)
  infixr 20 ___

  lam : ∀ {γ} → Term const (suc γ) → Term const γ
  lam t = ' "λ" • bind t

  ~ : ∀ {γ} → Var γ → Term const γ
  ~ vr = thunk (var vr)

  app : ∀ {γ} → Compu γ → Const γ → Term compu γ
  app e s = elim e s

```

- Can we model STLC?

- we have a universe

$U : \text{Pattern } 0$

$U = \text{' "U" '}$

$U\text{-type} : \text{TypeRule}$

subject $U\text{-type} = U$

premises $U\text{-type} = \text{' "T" '}, (\varepsilon (U \text{ placeless}))$

$U\text{-univ} : \text{UnivRule}$

input $U\text{-univ} = U$

premises $U\text{-univ} = \text{input } U\text{-univ}, (\varepsilon (\text{' "T" '}))$

- a base type α in the universe

$\alpha : \text{Pattern } 0$

$\alpha = \text{' "\alpha" '}$

$\alpha\text{-rule} : \text{TypeRule}$

subject $\alpha\text{-rule} = \alpha$

premises $\alpha\text{-rule} = \text{' "T" '}, (\varepsilon (\alpha \text{ placeless}))$

$\alpha\text{-inuniv} : \exists\text{rule}$

subject $\alpha\text{-inuniv} = \alpha$

input $\alpha\text{-inuniv} = U$

premises $\alpha\text{-inuniv} = \text{' "U" '}, (\varepsilon (\alpha \text{ placeless}))$

- which has a value "a"

$a : \text{Pattern } 0$

$a = \text{' "a" '}$

$a\text{-rule} : \exists\text{rule}$

subject $a\text{-rule} = a$

input $a\text{-rule} = \alpha$

premises $a\text{-rule} = \text{' "\alpha" '}, (\varepsilon (a \text{ placeless}))$

$\beta : \text{Pattern } 0$

$\beta = \text{' "\beta" '}$

$\beta\text{-rule} : \text{TypeRule}$

subject $\beta\text{-rule} = \beta$

premises $\beta\text{-rule} = \text{' "T" '}, (\varepsilon (\beta \text{ placeless}))$

$\beta\text{-inuniv} : \exists\text{rule}$

subject $\beta\text{-inuniv} = \beta$

input $\beta\text{-inuniv} = U$

premises $\beta\text{-inuniv} = \text{' "U" '}, (\varepsilon (\beta \text{ placeless}))$

- and a value "b"

$b : \text{Pattern } 0$

b = ' "b"

b-rule : \exists rule

subject b-rule = b

input b-rule = β

premises b-rule = (' " β " ' , (ε (b placeless)))

- REMEMBER TO ADD RULE TO BOTTOM!!!

- and a function type \Rightarrow in the universe

\Rightarrow : Pattern 0

\Rightarrow = place ι • ' " \rightarrow " • place ι

\Rightarrow -rule : TypeRule

subject \Rightarrow -rule = \Rightarrow

premises \Rightarrow -rule = ((' " \top " • place ι) • place ι) , ((type (\star •) $\iota \Rightarrow$
type ($\bullet \bullet \star$) $\iota \Rightarrow$
 ε (\Rightarrow placeless)))

\Rightarrow -inuniv : \exists rule

subject \Rightarrow -inuniv = \Rightarrow

input \Rightarrow -inuniv = U

premises \Rightarrow -inuniv = (((U • place ι) • place ι) , ((type (\star •) $\iota \Rightarrow$
type ($\bullet \bullet \star$) $\iota \Rightarrow$
 ε (\Rightarrow placeless)))

- which has lambda terms as it's values

lam : Pattern 0

lam = ' " λ " • bind (place ι)

- we check the type of abstractions

lam-rule : \exists rule

subject lam-rule = lam

input lam-rule = \Rightarrow

premises lam-rule = input lam-rule • bind (place ι) , (((\star •) / ε) \vdash ' ((($\bullet \bullet \star$) / ε) \exists ' • bind \star [ι]))
 $\Rightarrow \varepsilon$ ((' " λ " • bind (' " \top ")) placeless)

- and we can type lam elimination

app-rule : ElimRule

targetPat app-rule = \Rightarrow

eliminator app-rule = place ι

premises app-rule = targetPat app-rule • place ι ,
(((\star •) / ε) \exists ' \star [ι]) \Rightarrow
 ε ((' " \top ") placeless)

output app-rule = ((($\bullet \bullet \star$) •) / ε)

- β rules

app- β rule : β -Rule

target app- β rule = ' " λ " • bind (place ι)

erule app- β rule = app-rule


```

redTerm app-βrule = ((• bind ★) •) / (ε -, (((• (• ★)) / ε) :: ((• ((★ •) •)) / ε)))

- η rules

lam-ηrule : η-Rule
checkRule lam-ηrule = lam-rule
eliminators lam-ηrule = ' • (bind (Pattern.thing (thunk (var ze))))

- first lets get all our rules together:

open import Data.List using (List; []; _::_)

typerules : List TypeRule
typerules = U-type :: α-rule :: ⇒-rule :: β-rule :: []

univrules : List UnivRule
univrules = U-univ :: []

⇒rules : List ⇒rule
⇒rules = lam-rule :: α-inuniv :: a-rule :: ⇒-inuniv :: b-rule :: []

elimrules : List ElimRule
elimrules = app-rule :: []

betarules : List β-Rule
betarules = app-βrule :: []

etarules : List η-Rule
etarules = lam-ηrule :: []

rules : RuleSet
rules = rs typerules univrules ⇒rules elimrules betarules etarules

```

.4.2 Beta reduction and normalisation

```

-----
- β-reduction tests
-----

module βredtests where

open import TypeChecker using (check-premise-chain)
open import BwdVec using (ε)
PC = check-premise-chain rules ε

test1 : Failable (Compu 0)
test1 = reduce betarules PC (lam (~ ze)) (α α) a

_ : test1 ≡ succeed (a :: α)

```

```

__ = refl

- function as input
test2 : Failable (Compu 0)
test2 = reduce betarules PC (lam (~ ze)) (( $\alpha$   $\alpha$ ) ( $\alpha$   $\alpha$ )) (lam (~ ze))

__ : test2  $\equiv$  succeed (lam (~ ze) :: ( $\alpha$   $\alpha$ ))
__ = refl

- take in a function, an argument and apply them
func : Const  $\gamma$ 
func = lam (lam (thunk (app (var (su ze)) (~ ze))))

ftype : Const  $\gamma$ 
ftype = ( $\alpha$   $\alpha$ )  $\alpha$   $\alpha$ 

arg1 : Const  $\gamma$ 
arg1 = lam (~ ze)

reducible-term : Compu 1
reducible-term = elim (elim (func :: ftype) arg1) (thunk (var ze))

test3 : Failable (Compu 0)
test3 = reduce betarules PC func ftype arg1

__ : test3  $\equiv$  succeed (lam (thunk (app (arg1 :: ( $\alpha$   $\alpha$ )) (~ ze))) :: ( $\alpha$   $\alpha$ ))
__ = refl

-----
- Normalization by evaluation tests
-----

module normbyeval where

open import TypeChecker using (infer; check-premise-chain)
open import Data.Product using (_,_)
open import BwdVec
PC = check-premise-chain rules

- take in a function, an argument and apply them
func : Const  $\gamma$ 
func = lam (lam (thunk (app (var (su ze)) (~ ze))))

ftype : Const  $\gamma$ 
ftype = ( $\alpha$   $\alpha$ )  $\alpha$   $\alpha$ 

arg1 : Const  $\gamma$ 
arg1 = lam (~ ze)

reducible-term : Compu 1
reducible-term = elim (elim (func :: ftype) arg1) (thunk (var ze))

```

- should perform a single reduction

test1 : Const 0

test1 = normalize etarules betarules ((infer rules) , PC) ε α (thunk (app (lam (~ ze) :: (α α)) a))

__ : test1 \equiv a

__ = refl

- should reduce multiple nested eliminations

test2 : Const 1

test2 = normalize etarules betarules
((infer rules) , PC) (ε -, α) α reducable-term

__ : test2 \equiv thunk (var ze)

__ = refl

- should eta-long variable

test3 : Const 1

test3 = normalize etarules betarules
((infer rules) , PC) (ε -, (α α)) (α α) (var ze)

__ : test3 \equiv lam (thunk (app (var (su ze)) (~ ze)))

__ = refl

- should eta-long stuck eliminations with function type

test4 : Const 1

test4 = normalize etarules betarules ((infer rules) , PC)
(ε -, (α (α α))) (α α) (app (var ze) a)

__ : test4 \equiv lam (thunk (app (app (var (su ze)) a) (~ ze)))

__ = refl

- should normalize under a binder

test5 : Const 0

test5 = normalize etarules betarules
((infer rules) , PC) ε α (lam (thunk (app (lam (~ ze) :: (α α)) a)))

__ : test5 \equiv lam a

__ = refl

- should normalize the eliminator, even if the target is neutral

test6 : Const 1

test6 = normalize etarules betarules
((infer rules) , PC) (ε -, (α α)) α (app (var ze)
(thunk (app (lam (~ ze) :: (α α)) a)))

__ : test6 \equiv thunk (app (var ze) a)

```

__ = refl

- should normalize the target even if it results in a neutral term

test7 : Const 1
test7 = normalize etarules betarules
      ((infer rules) , PC) (ε -, α) α
      (app (app (lam (~ (su ze))) :: (α α)) a) a)

__ : test7 ≡ thunk (app (var ze) a)
__ = refl

- should normalize even if the elimination target body was initially stuck
test8 : Const 0
test8 = normalize etarules betarules
      ((infer rules) , PC) ε α
      (app (lam (thunk (app (var ze) a)) :: ((α α) α))
      (lam (~ ze)))

__ : test8 ≡ a
__ = refl

- should eta-expand multiple times
test9 : Const 1
test9 = normalize etarules betarules
      ((infer rules) , PC) (ε -, (α α α)) (α α α) (var ze)

__ : test9 ≡ lam (lam (thunk (app (app (var (su (su ze))) (~ (su ze))) (~ ze))))
__ = refl

```

.4.3 Eta expansion

```

-----
- Test 1:
- η expanding λx.a = λy.(λx.a y)
-----

test1 : Const 0
test1 = lam a

test1type : Const 0
test1type = α α

__ : η-match lam-ηrule test1type
    ≡
    succeed ((thing α) • (' • (thing α)))
__ = refl

```

```

__ :  $\eta$ -expand lam- $\eta$ rule test1type test1
       $\equiv$ 
      lam (thunk (elim ((test1 ^term) :: (test1type ^term)) (~ ze)))
__ = refl

-----
- Test 2:
-  $\eta$  expanding  $((\lambda x. \lambda y. x) b) = \lambda y. (((\lambda x. \lambda y. x) b) y)$ 
-----

test2 : Const 0
test2 = thunk (elim (lam (lam (~ (su ze))) :: ( $\beta$   $\alpha$   $\beta$ )) b)

test2type : Const 0
test2type =  $\alpha$   $\beta$ 

__ :  $\eta$ -match lam- $\eta$ rule test2type
       $\equiv$ 
      succeed ((thing  $\alpha$ ) • (' • (thing  $\beta$ )))
__ = refl

__ :  $\eta$ -expand lam- $\eta$ rule test2type test2
       $\equiv$ 
      lam (thunk (elim ( $\leftarrow\leftarrow$  (test2 ^term) (test2type ^term)) (~ ze)))
__ = refl

```

.4.4 Type checking STLC

```

- should check annotated terms are typed:

__ : infer rules  $\varepsilon$  (lam (~ ze) :: ( $\alpha$   $\alpha$ ))
       $\equiv$ 
      succeed ( $\alpha$   $\alpha$ )
__ = refl

__ : infer rules  $\varepsilon$  ( $\alpha$  :: ( $\alpha$   $\alpha$ ))
       $\equiv$ 
      fail "failed  $\ni$ -check:  $\alpha \rightarrow \alpha$  is not the type of  $\alpha$ "
__ = refl

- should check applications are typed:

__ : infer rules  $\varepsilon$  (app (lam (~ ze) :: ( $\alpha$   $\alpha$ )) a)
       $\equiv$ 
      succeed  $\alpha$ 
__ = refl

```

```

_ : infer rules  $\varepsilon$  (app (lam (~ ze) :: (( $\alpha$   $\alpha$ ) ( $\alpha$   $\alpha$ ))) (lam (~ ze)))
  ≡
  succeed ( $\alpha$   $\alpha$ )
_ = refl

_ : infer rules  $\varepsilon$  (app ((lam b) :: ( $\alpha$   $\beta$ )) a)
  ≡
  succeed  $\beta$ 
_ = refl

_ : infer rules  $\varepsilon$  (app
  ((lam (thunk (elim (var ze) a))) :: (( $\alpha$   $\alpha$ )  $\alpha$ ))
  (thunk (elim (lam (~ ze) :: (( $\alpha$   $\alpha$ ) ( $\alpha$   $\alpha$ ))) (lam (~ ze)))))
  ≡
  succeed  $\alpha$ 
_ = refl

_ : infer rules ( $\varepsilon$  -, ( $\alpha$   $\alpha$ ) -,  $\beta$ ) ((lam (thunk (elim (var (su (su ze))) a))) :: ( $\beta$   $\alpha$ ))
  ≡
  succeed ( $\beta$   $\alpha$ )
_ = refl

- should check the target of elimination first:

_ : infer rules  $\varepsilon$  (app ((lam a) :: ( $\alpha$   $\beta$ )) b)
  ≡
  fail "failed  $\ni$ -check:  $\beta$  is not the type of a"
_ = refl

- if target of elimination passes typchecking, should check the eliminator:

_ : infer rules  $\varepsilon$  (app (lam (thunk (var ze)) :: ( $\alpha$   $\alpha$ )) b)
  ≡
  fail "failed  $\ni$ -check:  $\alpha$  is not the type of b"
_ = refl

- should correctly type nested eliminations

_ : infer rules  $\varepsilon$ 
  (elim
    (lam (thunk
      (elim
        (var ze)
        a))
      :: (( $\alpha$   $\alpha$ )  $\alpha$ ))
    (thunk
      (elim
        ((lam (~ ze)) :: (( $\alpha$   $\alpha$ ) ( $\alpha$   $\alpha$ )))
        ((lam (~ ze)))))
  )
  ≡
  succeed  $\alpha$ 
_ = refl

```

.5 Lattice meet/join algorithms

We simplify these algorithms by ignoring scope, in practice we would need to consider scope if we were to use these algorithms in our code.

The following algorithm may be used to calculate the join of a set of patterns.

1. Consider all the topmost data constructors in the patterns (disregarding places).
 - (a) if they are not all the same then the join is a *place*
 - (b) if they are all *atoms*
 - i. if they are all the same atom, then the join is that atom
 - ii. if they are all different atoms, then the join is a *place*
 - (c) if they were all *places* then the join is a *place*
 - (d) if they are all \perp s then the join is \perp
 - (e) if they are all pairs
 - i. calculate the join of all the left elements J_l
 - ii. calculate the join of all the right elements J_r
 - iii. the join is the pair $(J_l \bullet J_r)$
 - (f) if they are all binds
 - i. calculate the join of the terms under all the binders J_b
 - ii. the join is (bind J_b)

The following algorithm may be used to calculate the meet of a set of patterns.

1. Consider all the topmost data constructors in the patterns (disregarding places).
 - (a) if they are not all the same then the meet is \perp
 - (b) if they are all *atoms*
 - i. if they are all the same atom, then the meet is that atom.
 - ii. if they are all different atoms, then the meet is \perp
 - (c) if they were all *places* then the meet is a *place*
 - (d) if they are all \perp s then the meet is \perp
 - (e) if they are all pairs
 - i. calculate the meet of all the left elements M_l
 - ii. calculate the meet of all the right elements M_r
 - iii. the meet is the pair $(M_l \bullet M_r)$
 - (f) if they are all binds
 - i. calculate the meet of the terms under all the binders M_b
 - ii. the meet is (bind M_b)

Bibliography

- [1] Andreas Abel. “Termination checking with types”. In: *RAIRO - Theoretical Informatics and Applications* 38.4 (Oct. 2004), pp. 277–319. doi: 10.1051/ita:2004015. URL: <https://doi.org/10.1051/ita:2004015>.
- [2] Guillaume Allais et al. “A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs”. In: *CoRR* abs/2001.11001 (2020). arXiv: 2001.11001. URL: <https://arxiv.org/abs/2001.11001>.
- [3] David Bell. “Gottlob Frege: Philosophical and Mathematical Correspondence.” In: *Philosophical Books* 22.2 (1981), pp. 117–121. doi: 10.1111/j.1468-0149.1981.tb00986.x. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1468-0149.1981.tb00986.x>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1468-0149.1981.tb00986.x>.
- [4] James Chapman et al. “The gentle art of levitation”. In: *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. Ed. by Paul Hudak and Stephanie Weirich. ACM, 2010, pp. 3–14. doi: 10.1145/1863543.1863547. URL: <https://doi.org/10.1145/1863543.1863547>.
- [5] Alonzo Church. “A Formulation of the Simple Theory of Types”. In: *The Journal of Symbolic Logic* 5.2 (June 1940), pp. 56–68.
- [6] Alonzo Church. “An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58.2 (Apr. 1936), pp. 345–363.
- [7] Luis Damas and Robin Milner. “Principal type-schemes for functional programs”. In: *POPL ’82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1982, pp. 207–212.
- [8] Jean-Yves Girard. “Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur”. PhD thesis. Université Paris Diderot, 1972.
- [9] Adam Gundry, Conor McBride, and James McKinna. “Type Inference in Context”. In: *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*. MSFP ’10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pp. 43–54. ISBN: 9781450302555. doi: 10.1145/1863597.1863608. URL: <https://doi.org/10.1145/1863597.1863608>.
- [10] Roger J. Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (Dec. 1969).
- [11] Andres Löb, Conor McBride, and Wouter Swierstra. “A Tutorial Implementation of a Dependently Typed Lambda Calculus”. In: *fundamenta-informaticae* (Jan. 2010).

- [12] Zhaohui Luo. “A Unifying Theory of Dependent Types: The Schematic Approach”. In: *Logical Foundations of Computer Science - Tver '92, Second International Symposium, Tver, Russia, July 20-24, 1992, Proceedings*. Ed. by Anil Nerode and Michael A. Tait-slin. Vol. 620. Lecture Notes in Computer Science. Springer, 1992, pp. 293–304. doi: 10.1007/BFb0023883. URL: <https://doi.org/10.1007/BFb0023883>.
- [13] Per Martin-Löf. *Intuitionistic Type Theory*. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980. Bibliopolis, 1984.
- [14] Conor McBride. “The types who say ‘ni’”. <https://github.com/pigworker/TypesWhoSayNi>. Feb. 2019.
- [15] Conor McBride and James McKinna. “Functional pearl: i am not a number-i am a free variable”. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*. Ed. by Henrik Nilsson. ACM, 2004, pp. 1–9. doi: 10.1145/1017472.1017477. URL: <https://doi.org/10.1145/1017472.1017477>.
- [16] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.
- [17] Ulf Norell and James Chapman. “Dependently Typed Programming in Agda”.
- [18] Benjamin C. Pierce and David N. Turner. “Local type inference”. In: *ACM Trans. Program. Lang. Syst.* 22.1 (2000), pp. 1–44. doi: 10.1145/345099.345100. URL: <https://doi.org/10.1145/345099.345100>.
- [19] John C. Reynolds. “Towards a Theory of Type Structure”. In: *Colloquium on Programming* (1974).
- [20] Philip Wadler. “Propositions as Types”. In: *Communications of the ACM* 58.12 (Dec. 2015).