Type checking and normalisation

James Maitland Chapman, BSc.

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

October 2008

Abstract

This thesis is about Martin-Löf's intuitionistic theory of types (type theory). Type theory is at the same time a formal system for mathematical proof and a dependently typed programming language. Dependent types are types which depend on data and therefore to type check dependently typed programming we need to perform computation (normalisation) in types.

Implementations of type theory (usually some kind of automatic theorem prover or interpreter) have at their heart a type checker. Implementations of type checkers for type theory have at their heart a normaliser. In this thesis I consider type checking as it might form the basis of an implementation of type theory in the functional language Haskell and then normalisation in the more rigorous setting of the dependently typed languages Epigram and Agda. I investigate a method of proving normalisation called Big-Step Normalisation (BSN). I apply BSN to a number of calculi of increasing sophistication and provide machine checked proofs of meta theoretic properties.

To Anne and Robin.

Acknowledgements

I would like to thank my supervisor Thorsten Altenkirch for his guidance in writing this thesis and his helpful feedback, and Conor McBride for his meticulous and highly constructive comments on a complete draft. I would also like to thank Wouter Swierstra, Peter Hancock, Nicolas Oury and Peter Morris for their comments on individual chapters. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through grant EP/C512022/1.

Contents

Chapter 1

Introduction

The aim of this work is to nail down the meta theory of Martin Löf's intuitionistic theory of types (type theory). Type theory is an intuitionistic formal system and a foundation for mathematics. In his own work on the meta theory of type theory Martin-Löf uses an informal intuitionistic meta language. Motivated by the fact that type theory is intended to be a full-scale system for intuitionistic mathematics I instead use type theory itself as the meta language and carry out my work formally. This thesis documents the progress I have made. In the rich language of type theory we are able to define the syntax and semantics (typed syntax) of languages in one go and deal only with expressions which are semantically valid (well typed). From a more logical perspective we are able to write down the judgments of the logic directly in type theory.

Whilst considerable extra effort is required to work in this more rigorous setting there are some payoffs. I am working towards writing a certified (correct by construction) type checker and this meta theoretic development forms the core of this. Also, type theory is a dependently typed programming language and our developments form substantial experiments in the relatively unexplored area of dependently typed programming. Previous work in this area have contributed to dependently typed programming. E.g. Induction-recursion [38] was previously an informal meta-theoretic technique. It has now crossed over into dependently typed programming.

In the next section I introduce intuitionistic mathematics, type theory, type checking and normalisation. After that I discuss related work. Finally I discuss the programming languages we will use.

1.1 Background

Intuitionism

At the beginning of the last century the foundations of mathematics were a very active subject and three schools were in fierce competition. Firstly, David Hilbert's programme attempted to justify mathematics by finitary means; by giving consistency proofs of mathematical theories in the theories themselves. Secondly Frege, and later Russell and Whitehead, attempted to derive all of mathematics from an axiomatic system of logic; essentially reducing mathematics to logic. This is known as logicism. Gödel's second incompleteness theorem showed that both these approaches were in pursuit of the impossible. Finally, there was Browuer's intuitionism. Brouwer took the radical approach to the foundational problem by attacking the central principles of classical mathematics and suggesting that mathematics should be reconstructed from the ground up by intuitionistic means. It is not that mathematics needs a foundation: it is that we need a new mathematics. This is what intuitionism intends to provide.

From an intuitionistic perspective the idea that mathematics should be concerned with statements that are either 'true' of 'false' in some universal sense is illusionary. Instead, mathematics is concerned with mental constructions. Fundamentally, we can only reason about mathematical structures we have constructed in our own minds. It is meaningless to assert that a statement is 'true' or 'false' without reference to how we might effect a construction (a proof) that the statement is true or that the statement is false. This subjective view of mathematics gives rise to a different interpretation of the logical operators ('and', 'or', 'not') to those found in classical logic. It is known (rightly or wrongly) as the Brouwer-Heyting-Kolmogorov (BHK) interpretation[1]. The most informative and easiest to grasp difference concerns the operator 'not'. In classical logic there is a rule which states for a given proposition A, 'A is equivalent to not not A'. Asserting A holds is equivalent to asserting that it is impossible that A does not hold. This is not the case intuitionistically. Only the weaker direction 'A implies not not A' is valid. To show A we must give a construction, we must provide some *evidence* and we do not consider that 'not not A' provides this.

[1]For an detailed survey of the different flavours of constructive mathematics including BHK see the first chapter of Troelstra and van Dalen [82]

Type theory

In the 1960s various formal systems were developed that tried to extend the BHK idea to a full-scale system. In particular they included the intuitionistic interpretation of universal and existential quantification. A proof of an existential statement is interpreted intuitionistically as a pair of a witness and a proof that the predicate holds. Howard's attempt was particularly influential. He privately circulated notes on the subject entitled "The Formulae-as-Types Notion of Construction" in 1969 which were finally published in 1980 [56]. Martin-Löf's system is the most enduring one but it owes a lot to those that came before. Expressions in Martin-Löf type theory are computer programs as well as constructive proofs. The types of the expressions are the specifications of these programs as well as the propositions being proved.

Type Checking

The correctness of expressions in type theory can be checked by a computer program. Indeed, the benefit of putting in the extra effort required to express constructions formally is that we reduce our steps to simple mechanical ones which can be checked by a machine.

Type theory is a formal logical system made up of a concise set of rules. A formal proof in type theory is, in effect, a derivation using the rules. Provided that the rules have been applied correctly and are in themselves valid then our derivation must be correct. It is unfeasible to check derivations by hand. When expressed formally, proofs of all but the most trivial propositions are extremely verbose. It quickly becomes impossible to just read a proof to decide whether you believe it.

Instead we can write a computer program (a *type checker*) which checks that the proof is a valid derivation (that it is a correct proof). It is possible (arguably crucial) to write such a program quite concisely. We sketch an implementation in the next chapter. If we can convince ourselves that the type checker is correct then we can indirectly believe that a proof is correct when the type checker says so. This point is discussed in detail in Pollack's paper "How to believe a machine-checked proof" [76].

A type checker checks whether a given term (a program or proof) is an element of a particular type (a specification or proposition). A type can also be thought of as a set. A type checker forms the core of any system to support

the construction of programs and proofs in type theory. Type checkers for type theory have at their core a normaliser. In this thesis we will consider first type checking as it might form the heart of an implementation of type theory in the functional language Haskell and then normalisation in the more rigorous setting of the dependently typed functional languages Epigram and Agda.

To type check programs in dependently typed (type theoretic) languages we must perform computation. This is because, unlike simply typed programming languages like Haskell, the types of expressions may contain term expressions which are syntactically distinct but nonetheless equal. We may need to perform computation to tell that they are equal. The concept of a term having multiple types (or more precisely multiple representatives of its type) is managed by the conversion rule:

$$\frac{\Gamma \vdash s \ : \ S \quad \Gamma \vdash S = T}{\Gamma \vdash s \ : \ T}\text{conv}$$

Here we see that how equality testing is connected to type checking.

As an example consider vectors (lists of a given length) where the type carries the length. The type checker must be able to tell that an element of the type 'Vector $(7 + 5)$' is also a element of the type 'Vector 12'. The easiest way to check this is to compute both expressions.

A central place where terms appear in types is the application rule for dependent functions:

$$\frac{\Gamma \vdash f \ : \ \Pi x{:}S.T \quad \Gamma \vdash s \ : \ S}{f\,s \ : \ T[x/s]}\text{app}$$

An example of a dependent function is vector concatenation:

$$\frac{vs \ : \ \text{Vector}\,m \quad ws \ : \ \text{Vector}\,n}{\text{conc}\,vs\,ws \ : \ \text{Vector}\,(m+n)}$$

$$\begin{array}{llcl} \text{conc} & \text{nil} & ws & = & ws \\ \text{conc} & (\text{cons}\,v\,vs) & ws & = & \text{cons}\,v\,(\text{conc}\,vs\,ws) \end{array}$$

On each line of the definition the type of the right hand side must be equal to the type of the left hand side. For the first line the type checker must check that 'Vector $(0 + n)$' is equal to 'Vector n'. On the second it must check that 'Vector $((1 + m) + n)$' is equal to 'Vector $(1 + (m + n))$'. This is indicative of the kind of computation that occurs in types. Note also that we must compute open terms.

Normalisation

A normaliser is a function which for any expression computes a value. This is a vital component of any type checker. A normalisation theorem usually states that every expression computes to a value. Consider, for example, a language that includes natural numbers and addition. If this language is normalising then any numeric expression (for example '7 + 5') is guaranteed to compute (normalise) to a numeral (in this case '12'). When considering a language that is also a logic, normalisation is a key property for consistency of the logic. Consistency states that it is impossible to derive something which is not the case using the rules of the logic. In a consistent logic we cannot prove nonsense such as '0 is equal to 1'.

The earliest known normalisation proof is Turing's proof [83] of normalisation for simply typed λ-calculus from 1942. It was published by Gandy in 1980 [42]. Normalisation proofs and cut-elimination proofs are in close correspondence[2] so the history could be traced back to Gentzen's 'Hauptsatz' [44] from 1935. However, the first published normalisation proof is that of Curry and Feys in their book Combinatory Logic [31] published in 1958.

1.2 Related work

First we discuss different approaches to formal meta theory and in particular normalisation. In the overview of the thesis in the next section we discuss work related specifically to each chapter.

Small-Step Operational Semantics

Traditionally small-step operational semantics has been used for normalisation proofs. Indeed Turing's [83], Tait's [79] and, Curry and Feys' [31] early proofs used this approach. The idea is to model the semantics on how one might perform a simple calculation by replacing expressions with equal ones until the desired result is achieved. To do this for a particular language one orients each equation in the equation theory (omitting reflexivity and transitivity) of the language as a rewrite (usually reduction) rule from left-to-right or right-to-left. The normalisation argument, in this setting, is usually either an arithmetic proof such as the earliest proof of normalisation by Turing or based on Tait's

[2] Howard [56] attributes this observation to Tait [79].

notion of strong computability in the case of Curry and Feys. A potential pitfall with this approach is that it does not model an efficient method of computation. Moreover, it is not always clear in which direction to orient the equations.

Typed Operational Semantics

H. Goguen introduced Typed Operational Semantics in his thesis [49]. Including typing information in the reduction relation allows him to prove meta theoretic properties more easily and more closely relate reduction (computation) and typing (type checking). He used this method to show strong normalisation of ECC [59]. Ghani used related techniques to show decidability of $\beta\eta$-equality for the difficult system of simply typed λ-calculus with coproducts [45].

Normalisation by Evaluation

Normalisation by Evaluation (NBE) uses computation at the meta level to implement computation at the object level. It represents object level functions as meta level functions. Berger and Schwichtenberg used Scheme functions to represent functions in simply typed λ-calculus [18]. Independently Martin-Löf [62] used a similar technique to NBE to represent functions in type theory by functions in a informal intuitionistic meta language. Since then NBE has been used in a variety of implementations and for a variety of systems of increasing complexity. The implementations of the theorem provers Epigram [68] and Agda [7] use NBE internally. Coquand and Dybjer considered NBE for combinatory logic and simply typed λ-calculus in [30]. C. Coquand considered NBE for simply typed λ-calculus with explicit substitutions [26]. Garillot and Werner formalise NBE for simply-typed λ-calculus in Coq[43]. Remarkably they are able to avoid the use of Kripke models. Danielsson extended this to Martin-Löf's Logical Framework [33]. Abel, Aehlig and Dybjer [3] and most recently Abel, Coquand and Dybjer [5] have used NBE for systems of dependent types.

Canonical forms and Hereditary Substitutions

This work takes the novel approach of considering first a system containing only canonical (normal) forms where the conversion relation is trivial and need

not be axiomatized. Whilst normal forms are not closed under ordinary substitutions, a version of substitution can be given called hereditary substitution which always returns normal forms and whose termination is justified by lexicographic recursion. A normaliser can then be easily defined using this notion of substitution. Abel has given such a normaliser for simply typed λ-calculus [2]. This approach simplifies the meta theory considerably but it is a long way from a practical implementation as it performs full normalisation. Also the elegant structural recursion argument does not easily extend to strong systems such as those with inductive types. The approach was pioneered in the Concurrent Logical Framework [87]. Harper and Licata have recently written a tutorial article [51] on formalising meta theory using Twelf [74]. Hereditary substitutions are central to their approach.

Based on very similar ideas is the work of Valentini [85] on normalisation for simply typed λ-calculus, later extended to system F by Capretta and Valentini [22]. The work of Matthes and van Raamsdonk in their respective PhD theses[66, 86] is also based on the same ideas.

Overview

Next we summarise each of the chapters of this thesis and work related specifically to them.

Chapter 2

In chapter 2 I present a type checker in Haskell. The idea here is to introduce a simple version of type theory, explain it by implementing it, discuss choices in implementation and introduce some components and techniques that we will use throughout the thesis.

Related work Type checkers are at the heart of the type theoretic theorem provers and programming systems AUTOMATH [34], the Constructive Engine [57], Coq [19], Lego [60], Alf [61], Agda [7], and Epigram [69]. Relatively little literature exists on the implementation of these systems but there are a number of papers about type checking algorithms. de Bruijn sketches an implementation of a type checker for a weaker theory than that of AUTOMATH in [35] but does not treat variables precisely. Coquand describes a simple type checker for a dependently typed language with β-conversion

in Haskell [29]. He considers $\beta\eta$-conversion for dependently typed language containing only Π-types in [27]. Abel and Coquand extend this to Σ-types in [4]. Epigram's type checker is closely related and a treatment of just the algorithm for $\beta\eta$-conversion for a language with Π,Σ,1 and 0-types is given in [25]. Most recently Abel, Coquand and Dybjer have given a type checker [6] for a dependently typed language whose syntax is close to the one considered in chapter 5 of this thesis.

Chapter 3

In chapter 3 we prove normalisation for a system based on combinators. I introduce the method for proving normalisation results (Big-Step Normalisation) and the Bove-Capretta technique [20] for dealing with nested recursive functions in type theory. I choose the simple system of combinators for pedagogical reasons. It should be noted that from this chapter onwards everything is formalised in Agda. I have carried out the full formalisation for this system and a number of extensions. The Agda formalisation of the basic system is presented appendix A and the extensions are available online [23].

Related work When extended with natural numbers the combinatory system corresponds to Gödel's system T [48]. This is the system for which Tait introduced his normalisation technique of strong computability [80]. We also use strong computability to show normalisation. The advantage of using combinators instead of λ-abstraction for function definitions is that it avoids intricate details of bound variables and substitution. In [30] Coquand and Dybjer also consider a combinatory calculus.

Chapter 4

I introduce λ-abstraction to the system in chapter 4 and prove normalisation for simply typed λ-calculus and a number of extensions. This necessitates a precise treatment of variables and substitutions which we were able to avoid in the previous chapter. The Agda formalisations are available online [23].

Related work We present a calculus with explicit substitutions which is similar to Abadi, Cardelli, Curien and Lèvy's λ^σ-calculus [1]. Our normalisation proof is similar to C. Coquand's [26] which considers λ-calculus with explicit substitutions but uses NBE instead of BSN.

Chapter 5

In chapter 5 we define a normaliser for a dependently typed language based on Martin-Löf's logical framework. We do not carry out the full formalisation but instead propose a implementation of a normaliser which is suitable for application of the BSN method.

Related work Various attempts at a complete formal treatment of type theory in type theory have been carried out before. Pollack formalised the syntax of type theory in his thesis [75] and proved a number of properties, but not normalisation. Barras formalised the Calculus of Constructions in Coq [16], proved normalisation and extracted a type checker in his master's thesis. In his PhD thesis[17] Barras extended this to include inductive definitions but he assumes normalisation in the final proof and hard codes a weak-head normaliser into the extracted program.

Our syntax is in the style of Dyber's "categories with families" [37, 54] and Martin-Löf's substitution calculus [64]. We diverge from the usual presentation of internal categories with families [37] mainly in that we include equality of contexts and our type, term and substitution equalities are heterogeneous with respect to their indices. E.g., We equate types in potentially different contexts. This, in part, leads to the inclusion of context equality. We also postulate injectivity of Π-types in the syntax.

Recently, Danielsson [33] gave an implementation of a normaliser for Martin-Löf's logical framework in Agda-Light (a precursor to the version of Agda used here). His normaliser is based on NBE. It is a considerable and impressive development but it contains a number of loose ends. Firstly the soundness property of the normaliser is not shown. Secondly the development uses features of uncertain foundation. The inductive definition of semantic values is inherently negative and the there are uses of mutually defined inductive types and functions which do not follow the standard pattern of an inductive type given together with a function defined on its constructors and hence do not correspond to know forms of induction-recursion [39].

1.3 Programming in Haskell, Epigram and Agda

In this thesis we use three functional programming languages. In chapter 2 we sketch the implementation of a type checker in Haskell. In chapter 3 we use

both Haskell and Epigram to write a normaliser for a combinatory calculus. In chapters 4 and 5 we use Epigram. At the time of writing there is no version of Epigram with which we can verify substantial programs. For this reason we have checked our Epigram developments in the closely related system Agda. Although there are some differences in the design and implementation of Epigram and Agda, most programs written in one system could be represented in the other. All the Agda source code from the later chapters is available online [23].

I do not give an introduction to the three programming languages used here as other people have already done a good job. Haskell has extensive documentation on its website [81] and in Graham Hutton's textbook [58] amongst others. For an introduction to Epigram, see Conor McBride's tutorial [69]. This serves as a perfectly good introduction to Agda as well, modulo syntactic differences. Ana Bove and Peter Dybjer have written an introduction to Agda [21] and Ulf Norell has given a course [72]. Below we briefly summarise the three languages.

Haskell

Haskell is pure functional programming language. It has an industrial strength compiler in GHC [46] and extensive libraries [53]. It is a natural choice to write such a type checker as it is lightweight and the code can be easily read, understood and efficiently compiled. Indeed the type checkers (and the rest of the implementation for that matter) of the Epigram and Agda systems are written in Haskell. Haskell does not force us to write types but their use is encouraged. The system implements an extension of the Damas-Milner type inference algorithm [32]. We use it as a "poor man's type theory", losing the expressivity of dependent types and inductive families, and the guarantee of termination provided by a consistent type system.

Epigram

Epigram is a dependently typed functional programming language. It is based on Martin-Löf's type theory and it has dependent pattern matching, inductive families and structural recursion. Dependent pattern matching [28] first appeared as a primitive in the language Alf. McBride showed, in his thesis, how dependent pattern matching can be reduced to conventional elimination

principles in type theory extended with Streicher's axiom K [70]. The Epigram language was designed by McBride and McKinna [71]. Epigram reduces both pattern matching and structural recursion to eliminators in type theory. Work is currently underway to combine the benefits of intensional and extensional type theory by introducing Observational Equality [8, 13] into Epigram, a prototype is under development.

Agda

Agda is also a dependently typed functional programming language. It is a direct descendant of Alf [61] and Cayenne [15]. The latest version is very similar to Epigram in features. In addition to Epigram's features it also supports mutual definitions and induction-recursion. It differs in design to Epigram in that rather than internalising pattern matching and termination it utilises external checkers. It also has a universe stratification checker, a feature missing altogether from Epigram. The downside to this approach of externalising certain principles is that the system cannot currently spit out a proof object to be checked by a stand-alone type checker. The upside is that these checkers can be turned off and compliance assumed which allows the development of the system, as well as programs/proofs developed within it, to be completed in stages.

Chapter 2

Type theory and type checking

In this chapter we present the rules of type theory and then show how we can implement a type checker which checks that a program obeys these rules. Our implementation uses a type directed equivalence checker based on an algorithm by Coquand [27]. Our implementation is also influenced by the formal developments that follow and can be seen as an introduction to them. In particular we represent variables using de Bruijn indices [36], values using closures and evaluation using an abstract machine.

There are many versions of Martin-Löf's type theory. We present an early version here[1] for pedagogical reasons, as it is the simplest to define and easiest to understand. It contains the strongly impredicative rule "type is a type" and was shown to be inconsistent by Girard [47]. The inconsistency of this system does not render it useless as a programming language. Indeed, lots of programming languages have an inconsistent or undecidable type system and their type checkers can be made to loop in obscure cases. In this system we are able to write a paradox (Girard's paradox) which exploits the absence of a type hierarchy. It is only when deliberately invoking such a paradox that the system is rendered useless. In fact, an implementation will probably hang (Epigram does). It is unlikely, however, that we would use the paradox to accidentally deceive the system (into thinking that zero is equal to one for example). A combination of this system and an external universe checker

[1] Actually, the version presented here is slightly different to Martin–Löf's as we include judgemental equality as opposed to untyped conversion.

recovers consistency.

Relationship to published paper This chapter is loosely based on the paper "Epigram Reloaded" [25], jointly written with McBride and Altenkirch. In the paper the intention was to explain how to implement the central component of the Epigram system as it existed at the time. With this in mind we tried to include as much as possible of the actual system. Here the focus is different, the implementation is intended to be pedagogical and act as both an introduction to type theory and as an introduction to the rest of the thesis. Specifically the implementation here differs in using de Bruijn indices for free variables whereas the "Epigram Reloaded" uses names and also by using environment machine for evaluation and first order closures in values as opposed to NBE.

2.1 Formal system

We will present the type system, which serve as the definition of the syntax and the judgements or axiomatic semantics, and later give a type checker written in Haskell that implements the type theory. The formal type system itself is presented as a sequence of judgements and then a sequence of rules are given by which derivations of these judgements can be constructed inductively. There are three mutually defined judgement forms:

$\Gamma \vdash$	Γ is a well formed context
$\Gamma \vdash t : T$	t is a well typed term of type T in context Γ
$\Gamma \vdash t \equiv t' : T$	t and t' are equal well typed terms of type T in context Γ

We can be relaxed about variable side conditions such as name capture and freshness as our use of de Bruijn representation internally ensures such properties are invariants which do not need to be explicitly rechecked.

Below we explain how to construct contexts. Either the context is empty or it can be extended by a (fresh) variable of valid type ($\star$ is the type of types).

$$\frac{}{\epsilon \vdash} \qquad \frac{\Gamma \vdash S : \star}{\Gamma, x{:}S \vdash}$$

We assume that contexts are only extended by fresh variables and that substitution is always capture avoiding. Rest assured, we will be more precise about such details later!

Instances of the second judgement explain how to form terms. We start with three rules which make up the basic setup. Firstly, the constant $\star$ is its own type. Secondly, variables from the context can be terms. Thirdly, if we have an existing term and its type is equal to a second type then it can also be a term of the second type. It is this last rule (the conversion rule) which forces the equality judgement to be mutually defined with the typing judgement.

$$\frac{\Gamma \vdash}{\Gamma \vdash \star \,:\, \star} \qquad \frac{\Gamma \vdash \quad x{:}S \in \Gamma}{\Gamma \vdash x \,:\, S} \qquad \frac{\Gamma \vdash s \,:\, S \qquad \Gamma \vdash S \equiv T \,:\, \star}{\Gamma \vdash s \,:\, T}$$

Now we can add types, starting with dependent functions (Π-types) which are expressed by their formation, introduction and elimination rules:

$$\frac{\Gamma, x{:}S \vdash T \,:\, \star}{\Gamma \vdash \Pi x{:}S.\, T \,:\, \star} \qquad \frac{\Gamma, x{:}S \vdash t \,:\, T}{\Gamma \vdash \lambda x.t{:}\Pi x{:}S.\, T} \qquad \frac{\Gamma \vdash t \,:\, \Pi x{:}S.\, T \qquad \Gamma \vdash u \,:\, S}{\Gamma \vdash t\, u \,:\, T[x/u]}$$

The last judgement is the definitional equality relation. It is reflexive, symmetric and transitive:

$$\frac{\Gamma \vdash t \,:\, T}{\Gamma \vdash t \equiv t \,:\, T} refl \qquad \frac{\Gamma \vdash t \equiv t' \,:\, T}{\Gamma \vdash t' \equiv t \,:\, T} sym$$

$$\frac{\Gamma \vdash t \equiv t' \,:\, T \qquad \Gamma \vdash t' \equiv t'' \,:\, T}{\Gamma \vdash t \equiv t'' \,:\, T} trans$$

It is a congruence:

$$\frac{\Gamma \vdash S \equiv S' \,:\, \star \qquad \Gamma, x{:}S \vdash T \equiv T' \,:\, \star}{\Gamma \vdash \Pi x{:}S.\, T \equiv \Pi x{:}S'.\, T' \,:\, \star} \Pi\text{cong}$$

$$\frac{\Gamma, x{:}S \vdash t \equiv t' \,:\, T}{\Gamma \vdash \lambda x.\, t \equiv \lambda x.\, t' \,:\, \Pi x{:}S.T} \xi$$

$$\frac{\Gamma \vdash t \equiv t' \,:\, \Pi x{:}S.\, T \qquad \Gamma \vdash u \equiv u' \,:\, S}{\Gamma \vdash t\, u \equiv t'\, u' \,:\, T[x/u]} \text{appcong}$$

It has β and η rules for functions:

$$\frac{\Gamma, x{:}S \vdash t \,:\, T \qquad \Gamma \vdash u \,:\, S}{\Gamma \vdash (\lambda x.\, t)\, u \equiv t[x/u] \,:\, T[x/u]} \beta \qquad \frac{\Gamma \vdash f \,:\, \Pi x{:}S.\, T}{\Gamma \vdash \lambda x.\, f\, x \equiv f \,:\, \Pi x{:}S.\, T} \eta$$

As part of the implementation that follows we define a type directed equality checker. A minor variation on the above rules fits more closely with this approach and we can easily show they are equivalent. We can replace the η and ξ rules with a single 'observational' rule:

$$\frac{\Gamma, x{:}S \vdash f\, x \equiv g\, x \,:\, T}{\Gamma \vdash f \equiv g{:}\Pi x{:}S.\, T} \text{obs}$$

This more accurately reflects how the type checker will compare functions. We create a fresh variable of appropriate type and compare the results of applying the functions to the fresh variable. Note that this rule does not test for extensional equality as we are not testing functions at every possible argument, we are just testing at one abstract argument.

The rules ξ and η are equivalent to obs:

Theorem 1.

$$\eta \wedge \xi \iff obs$$

Proof. Both directions follow by simple equation reasoning. □

2.2 Implementation

Having completed the formal definition of the type system we will describe the implementation of a type checker for this system in Haskell. The definition of the type checker starts with the definition of syntactic terms. This syntax is not typed in the sense that we do not give terms together with their types as in the formal judgements. We do however give variables with their types to avoid having to carry around a context.

When representing abstract syntax with binding (here Π and λ) we have two basic choices: higher order or first order. Higher order abstract syntax [50] (HOAS) uses the binding constructs of the meta language to explain binding in the object language. This makes it hard to inspect the body of a binder without first applying the meta level function to a first order representation of a variable. Another downside is that it does not translate naturally into type theory as the definitions are inherently negative. Here is a typical definition of untyped λ-terms in Haskell:

```
data Term = App Term Term | Lam (Term -> Term)
```

This makes sense because, whilst we allow any functions in `(Term -> Term)` as arguments to the `Lam` constructor, in practice we only use functions that encode binding. In type theory we would have to be more precise in the definition and restrict the function space in some way to avoid the negative occurrence of `Term`. The benefit of HOAS is that binding and variables have to be dealt with only once (in the meta language) but it doesn't solve the problem completely. How do we treat binding in the meta language?

For first order abstract syntax the central question is whether to give a named or nameless presentation of variables. Proponents of the named representations are usually interested in sticking closely to the usual paper based practices so as to easily formalise existing literature. For example Cheney and Urban's recent verification [84] of Harper and Pfenning's equality checker for LF [52]. This is certainly a worthy task. We, on the other hand, do not want to be tied to existing on paper mathematical conventions. We choose a nameless and first order approach as it fits more closely with the categorical semantics in the dependently typed case and is more convenient to implement.

Nameless syntax presents us with two sensible options: de Bruijn indices or de Bruijn levels. The latter would make our algorithm easier to implement as we can avoid having to implement weakening. However we stoically use de Bruijn indices here as we use them in with typed syntax later where there is no choice but to implement weakening (we must introduce new assumptions into the context somehow). Nameless approaches are often criticised for involving intricate error prone arithmetic and bookkeeping and often leading to irritating off-by-one errors. In this chapter we try to give a simple presentation and must be careful not to introduce things like off-by-one errors. Later, when we work with typed syntax, we can guarantee not to encounter these problems and manipulate de Bruijn indices in a safe way.

Terms, values and evaluation

We introduce the syntax of terms below. After this we introduce an internal (to the type checker) representation of values which is more convenient to work with. Our representation of values is for terms that have been computed to weak-head normal form. This is not as computationally expensive as performing full normalisation and is a convenient internal representation as it is just enough to be able to tell what something is by looking at the outer constructor. This allows us to easily answer question like 'is this $\star$?' or 'is this a Π-type?' which show up regularly when type checking. Now let us look at the definition of terms:

```
data Term = V (Int,Val)      -- type carrying de Bruijn indices
          | Pi Term Term     -- pi types
          | Lam Term         -- lambda terms
          | Term :$ Term     -- application
```

```
  | Coe Term Term    -- term given with its type (first)
  | Star             -- the type of types
```

Variables are represented as de Bruijn indices and carry (the internal value of) their type to avoid needing an explicit context. Π-types carry their domain and range (remember that types and terms are syntactically identified). Quantifiers (`Pi` and `Lam`) do not carry a name for the variable. λ-abstractions do not carry their domain type. Application is represented using an infix operator. The constructor `Coe` means that an explicit type can be given to a term. In the case of applying a λ-abstraction to an argument this is necessary as by omitting the domain from the λ-abstraction we cannot easily infer its type so it is given explicitly.

Dependently typed terms can be quite large and contain a lot of duplication so it is not sensible to do more computation than is strictly necessary when type checking hence we use the internal representation of weak-head normal forms for internal values. We could just use the representation of terms for values but we choose to use a new datatype which allows us to guarantee that the values are in weak-head normal form in the type itself and confirm the exclusion of insufficiently computed terms. Coquand's algorithm uses closures to represent λ-abstractions and we will do the same here. In "Epigram Reloaded" [25] we used higher order closures but here we use first order closures which are pairs of syntactic λ-abstraction bodies and environments (sequences of values) that give values to all the variables in the context except the variable bound by the λ. This lends itself easily to defining an evaluator from terms to values expressed as an environment machine.

Below we define values, environments and neutral terms. Values are terms that have been computed to the point that we know what their outer constructor is. λ-abstractions and Π-types contain closures represented as pairs of a syntactic term and an environment giving values to all but the bound variable in the term. Intuitively a closure contains everything necessary to evaluate a term that is under a binder except for a value for the bound variable, when this value turns up the remaining evaluation can happen. Values can also be neutral terms or star. A neutral term is something inert that is irreducibly in eliminator form, due to the presence of a variable impeding computation. Star is, as before, the type of types.

```
data Val = VLam (Term,Env)
```

```
        | VPi Val (Term,Env)
        | VStar
        | Ne Ne
```

Environments are left-to-right sequences of values.

```
data Env = E0 | Env :< Val
```

We define a simple lookup operation on environments:

```
lookup :: Int -> Env -> Val
lookup 0 (_  :< v) = v
lookup i (vs :< _) = lookup (i - 1) vs
```

Neutral terms are either variables represented as a pair of a de Bruijn index (`Int`) and the type (`Val`), or 'stuck' applications of a neutral term (`Ne`) to a value (`Val`).

```
data Ne = NV (Int,Val) |  Ne :$$ Val
```

Next we define a simple evaluator that takes a term in an appropriate environment (assigning values to variables) and produces a value. The function `eval` is defined mutually with a value application operation `$$`. This style of evaluator is a recurring theme in this thesis and will be used later as a component in normalisation proofs.

```
eval :: Term -> Env -> Val
eval (V (i,_)) vs = lookup i vs
eval (Pi d r)  vs = VPi (eval d vs) (r,vs)
eval (Lam t)   vs = VLam (t,vs)
eval (t :$ u)  vs = eval t vs $$ eval u vs
eval (Coe _ t) vs = eval t vs
eval Star      _  = VStar

($$) :: Val -> Val -> Val
VLam (r,vs) $$ a = eval r (vs :< a)
Ne n        $$ a = Ne (n :$$ a)
```

Implementing judgments

In the remainder of this chapter we introduce some operations which match closely to the judgments in the formal presentation. We give their signatures now:

```
infer :: Env -> Term -> Maybe Val
check :: Env -> (Term,Val) -> Maybe Val
```

These two operations correspond to the typing judgment $\Gamma \vdash t : T$. The `infer` operation takes an environment (mapping variables to values) and a (potentially open) syntactic term, and infers its type or fails. The `check` operation takes an environment, a syntactic term and the value of it type. It infers the type of the term and compares it to the supplied type. It returns the value of the term or fails.

```
eq :: (Val,(Val,Val)) -> Maybe () -- equate values
neq :: (Ne,Ne) -> Maybe Val       -- equate neutral terms
```

Next is the equality checker which corresponds to the equality judgment $\Gamma \vdash t \equiv t' : T$. The first operation `eq` takes first a value type and then a pair of values and checks that they are equal or fails. The second operation `neq` compares neutral terms but instead of the type being pushed in the type is returned this time or it fails.

When testing equality we need to create fresh variables. We can see this in the 'obs' rule particularly:

$$\frac{\Gamma, x{:}S \vdash f\,x \equiv g\,x \;:\; T}{\Gamma \vdash f \equiv g \;:\; \Pi x{:}S.\,T}\text{obs}$$

Looking at this from a logical perspective for a moment we can see that we introduce a new assumption into the context on the top line, this weakens our statement and for this reason the operation that we require to introduce a new variable is called *weakening*. As we are using de Bruijn indices (effectively offsets) for variables, we need a weakening operation to introduce a fresh variable on the right of the context. This effectively shifts all the other offsets along by one. We will define weakening mutually for values, environments and neutral terms:

```
wk :: Val -> Val
```

```
wk (VLam (r,vs))  = VLam (r, ewk vs)
wk (VPi d (r,vs)) = VPi  (wk d) (r, ewk vs)
wk (Ne n)         = Ne   (nwk n)
wk VStar          = VStar
```

An advantage of our representation of values is that we do not have to deal with the problematic operation of pushing a weakening under a binder. Our binders (`VLam` and `VPi`) always contain closures so we just weaken the environment instead of pushing the weakening through the body of the binder.

```
ewk :: Env -> Env
ewk E0        = E0
ewk (vs :< v) = ewk vs :< wk v


nwk :: Ne -> Ne
nwk (NV (i,t)) = NV (i + 1,wk t)
nwk (n :$$ v)  = nwk n :$$ wk v
```

When we finally reach a variable we increment its value by one and weaken its type.

Checking equality

We now have enough machinery to actually implement the components of the type checker which correspond to the judgements. The first component we describe is a type directed equality checker which corresponds to the equality judgement. It takes the value of a type and pair of values of that type and checks if they are equal.

```
eq :: (Val,(Val,Val)) -> Maybe ()
```

We will use the Maybe monad to propagate failure, rather than explicitly producing and detecting values representing error conditions. This fits more easily with the other components that use `Maybe`. We deal with canonical types first.

```
eq (VStar,(VStar,VStar)) = return ()
```

Canonical elements of `VStar` are either also `VStar` in which case they are equal by reflexivity (as above) or they are both `VPi`. When comparing `VPis`

we compare the domains and then create a fresh variable of domain type and compare the result of evaluating the respective ranges with the fresh variable. This corresponds closely to the Πcong rule.

```
eq (VStar,(VPi d1 (r1,vs1),VPi d2 (r2,vs2))) = do
  eq (VStar,(d1,d2))
  eq (VStar,(eval r1 (ewk vs1 :< Ne (NV (0,d1))),
             eval r2 (ewk vs2 :< Ne (NV (0,d2)))))
```

Next are elements of Π-types (functions). As described in the obs-rule we create a fresh variable and compare the results of applying the functions to that variable. We also compute the resultant type of instantiating the range of the Π-type with the fresh variable as that is the type at which to compare the result of applying the functions. As we are using de Bruijn indices for all variables we must weaken the functions before applying them to the fresh variables.

```
eq (VPi d (r,vs),(f,g)) =
  eq (eval r (ewk vs :< Ne (NV (0,d))),
      (wk f $$ Ne (NV (0,d)),wk g $$ Ne (NV (0,d))))
```

If the elements are not of canonical type, they must be neutral. In which case we defer to the function `neq` to compare them structurally.

```
eq (_,(Ne n1,Ne n2)) = neq (n1,n2) >> return ()
```

If the elements are not neutral and not in a canonical type they cannot be equal.

```
eq (_,(_,_)) = Nothing
```

The function `neq` compares neutral terms and returns either failure or the type of the neutral terms being compared.

```
neq :: (Ne,Ne) -> Maybe Val
```

Given two neutral terms – iterated applications – what we are doing is peeling the arguments away until we get to the variables at the bottom (which must be the same) then we pass the types back so we can compare the arguments in the appropriately instantiated types. First we consider the case for variables:

```
neq (NV (i,t),NV (j,_)) = if i == j then return t else Nothing
```

If the variables are equal we return the type otherwise we fail. We do not check that the types of the variables are equal. This is an invariant of the representation. This syntax already has variables decorated with the *value* of their types, which would be produced by an earlier parse. We assume that this operation (from a contexts and terms to decorated terms) is correct.

```
neq (n1 :$$ v1,n2 :$$ v2) = do
  VPi d (r,vs) <- neq (n1,n2) -- compare functions
  eq (d,(v1,v2))              -- compare arguments at dom type
  return (eval r (vs :< v1))  -- return type of application
```

In the case of neutral applications we compare the neutral functions and retrieve the function type. Then we compare the arguments in the domain type and return the range instantiated with one of the arguments (either will do as they have been shown to be equal).

```
neq (_,_) = Nothing
```

A variable cannot be equal to an application and vice versa. That completes the definition of the type directed equality checker.

Checking types

Next we define two functions `infer` and `check`. They each take as a first argument an appropriate environment which gives values to variables. `infer` takes a syntactic term and returns the value of its type or failure. `check` takes a syntactic term and the value of its type and returns the computed value of the term.

```
infer :: Env -> Term -> Maybe Val
```

If the term is `Star` we can directly infer its type, `VStar`.

```
infer vs Star       = return VStar
```

Next are variables:

```
infer vs (V (i,v))  = return v
```

In the case of variables we simply return the type which they carry.

```
infer vs (Pi d r)   = do
  vd <- check vs (d, VStar)
  check (ewk vs :< Ne (NV (0,vd))) (r,VStar)
  return VStar
```

Before returning that the type of a `Pi` is `VStar` we must check that the domain is a type and that the range is a type in the presence of a fresh variable of domain type.

```
infer vs (t :$ u)   = do
  VPi d (r,vs') <- infer vs t
  vu <- check vs (u,d)
  return (eval r (vs' :< vu))
```

For applications we first infer the type of the function. Then we check that the argument has the type of the domain type of the function. Finally we return the type of the range instantiated with the argument. Next we consider `Coe` where terms are given together with a type. In particular λ-terms in applications must be given with a type as we have not included the domain as an argument to `Lam` so the type cannot be inferred by the algorithm. This presentation of λ-abstraction is so-called Curry style, where we can only check the type. If we presented the system in Church style, where the λ-abstractions are annotated with their domain type then we would be able to infer the type but we would need to extend the algorithm with some extra components to support this. It is quite possible to support both checkable and inferable λ-abstractions in one system and this is quite desirable in a full-scale system as it is more flexible for the user.

For `Coe` we first check that the type is in fact a type and then we check the term in that type.

```
infer vs (Coe ty t) = do
  vty <- check vs (ty,VStar)
  check vs (t,vty)
  return vty
```

The next function is `check`. There are two cases. Firstly we consider λ-terms whose type cannot be inferred. We check the that the body of the λ-term has the type of the range of its Π-type when they are both instantiated with

a fresh variable. If this succeeds we return the closure of the syntactic body and the environment we were given.

```
check :: Env -> (Term,Val) -> Maybe Val
check vs (Lam t,VPi d (r,vs')) = do
 let rty = eval (ewk vs' :< Ne (NV (0,d))) r
 check (ewk vs :< Ne (NV (0,d))) (t,rty)
 return (VLam (t,vs))
```

The other case for check corresponds to the conversion rule. We infer the type of the term and then compare the supplied type with the inferred one. If this succeeds we return the value of the term.

```
check vs (t,ty) = do
  vty <- infer vs t
  eq (VStar,(vty,ty))
  return (eval t vs)
```

Now that we have all the necessary machinery, we can define a checking function for closed terms as an interface to our type checker. It takes a syntactic term in a syntactic type. First we check that the syntactic type is actually a type and compute its value. Then we check the term in the value of the type. We return success or failure.

```
checker :: (Term,Term) -> Maybe ()
checker (t,ty) = do
  vty <- check E0 (ty,VStar)
  check E0 (t,vty)
  return ()
```

2.3 Extensions

The unit type and the empty type

We extend the type system with the unit type and the empty type. Logically they correspond to true and false respectively. The unit type has an introduction rule and no elimination rule and the empty type has no introduction rule and an elimination rule. The unit type has a trivial canonical inhabitant (called 'void') and the empty type as an eliminator (called 'naughtE') which

states that if you have an inhabitant of the empty type then you can have an inhabitant of any type.

$$\frac{}{\Gamma \vdash 1 \ : \ \star}\text{1-form} \quad \frac{}{\Gamma \vdash \langle\rangle \ : \ 1}\text{1-intro}$$

$$\frac{}{\Gamma \vdash 0 \ : \ \star}\text{0-form} \quad \frac{\Gamma \vdash z \ : \ 0}{\Gamma \vdash NE\, z \ : \ \Pi T{:}\star.\, T}\text{0-elim}$$

We have an observational rule for the unit type which states that we do not even have to look at inhabitants of the unit type to know that they are equal. We have a similar rule for the empty type as well. In the case of the empty type this does not give us full η-equality as, for example, we don't identify all functions from the empty type to a particular type. Lastly we have a congruence rule for applications of the 'naughtE' eliminator. Notice there is no computation rule (β-rule) for either type.

$$\frac{}{\Gamma \vdash u \equiv u' \ : \ 1}\text{1-obs} \quad \frac{}{\Gamma \vdash z \equiv z' \ : \ 0}\text{0-obs}$$

$$\frac{\Gamma \vdash z \equiv z' \ : \ 0}{\Gamma \vdash NE\, z \equiv NE\, z' \ : \ \Pi T{:}\star.\, T}\text{NE-cong}$$

We extend the `Term`, `Val` and `Ne` datatypes to accommodate the new types:

```
data Term = ...
          | One
          | Void
          | Zero
          | OE Term
          | ...
```

For values we add constructors for the type constructors and for the introduction (canonical) forms. Only the unit type has a constructor form, void.

```
data Val = ...
         | VOne
         | VVoid
         | VZero
         | ...
```

For elimination forms we add a constructor for neutral instances and function to deal with non-neutral instances:

```
data Ne = ...
        | NOE Ne
        | ...

vOE :: Val -> Val
vOE (Ne n) = Ne (NOE n)
vOE _      = error "eek"
```

Here the semantic eliminator has no computational behaviour as the only possible inhabitant of the zero types is a neutral term, if we have something else then something must have gone wrong and we produce an error.

We extend the evaluator:

```
eval One          _  = VOne
eval Void         _  = VVoid
eval Zero         _  = VZero
eval (OE t)       vs = vOE (eval t vs)
```

We extend weakening:

```
wk VOne           = VOne
wk VVoid          = VVoid
wk VZero          = VZero

nwk (NOE n)   = NOE (nwk n)
```

Next we extend the equality checker. The unit and empty types are equal to themselves. We did not add explicit rules for this in the formal presentation. The unit and empty types are simple constants and these rules are just instances of reflexivity.

```
eq (VStar,(VOne,VOne))   = return ()
eq (VStar,(VZero,VZero)) = return ()
```

All instances of the unit and empty type are equal:

```
eq (VOne,_)  = return ()
eq (VZero,_) = return ()
```

We add an instance to the equality checker for neutral terms to account for 0-elimination. Here we do not even check that the instances of the empty type are equal and just return the appropriate type:

```
neq (NOE z1,NOE z2) =
  return (VPi VStar (V (0,Star), E0))
```

These three additions to the equality checker correspond exactly to the three equality rules above except for the fact we ignore the premise in the last one.

We extend the `infer` function with four cases which correspond exactly to the four typing rules above.

```
infer _  One        = return VStar
infer _  Void       = return VOne
infer _  Zero       = return VStar
infer vs (OE t)      = do
  check vs (t,VZero)
  return (VPi VStar (V (0,Star), E0))
```

Σ-types

We add dependent pairs (Σ-types) with $\beta\eta$-equality to the system. They are a generalisation of cartesian products in the same sense as dependent functions (Π-types) are a generalisation of simple functions. Here the type of the second component depends on the value of the first component of the pair.

$$\frac{\Gamma,\, x \,:\, S \vdash T \,:\, \star}{\Gamma \vdash \Sigma x{:}S.\, T}\Sigma\text{-form} \quad \frac{\Gamma \vdash s \,:\, S \quad \Gamma \vdash t \,:\, T[x/s]}{\Gamma \vdash \langle s,\, t\rangle \,:\, \Sigma x{:}S.\, T}\Sigma\text{-intro}$$

$$\frac{\Gamma \vdash p \,:\, \Sigma x{:}S.\, T}{\Gamma \vdash \pi_1\, p \,:\, S}\Sigma\text{-elim-1} \quad \frac{\Gamma \vdash p \,:\, \Sigma x{:}S.\, T}{\Gamma \vdash \pi_2\, p \,:\, T[x/\pi_1\, p]}\Sigma\text{-elim-2}$$

We have formation (identical to Π), introduction (pairing) and elimination rules (projections). In the type of the introduction rule $\langle \_, \_ \rangle$ we see that the type of the second component t is instantiated with the value of the first, s. In the second rule for projection (π_2) we see that the type is instantiated with

the first projection.

$$\frac{\Gamma \vdash S \equiv S' \ : \ \star \qquad \Gamma, x{:}S \vdash T \equiv T' \ : \ \star}{\Gamma \vdash \Sigma x{:}S.\, T \equiv \Sigma x{:}S'.\, T' \ : \ \star}\Sigma\text{-cong}$$

$$\frac{\Gamma \vdash \pi_1\, p \ \equiv \ \pi_1\, p' \ : \ S \qquad \Gamma \vdash \pi_2\, p \ \equiv \ \pi_2\, p' \ : \ T[x/\pi_1\, p]}{\Gamma \vdash p \ \equiv \ p' \ : \ \Sigma x{:}S.\, T}\Sigma\text{-obs}$$

$$\frac{\Gamma \vdash \langle s,\, t\rangle \ : \ \Sigma x{:}S.\, T}{\Gamma \vdash \pi_1\, \langle s,\, t\rangle \ \equiv \ s \ : \ S}\Sigma\text{-}\beta\text{-}1 \qquad \frac{\Gamma \vdash \langle s,\, t\rangle \ : \ \Sigma x{:}S.\, T}{\Gamma \vdash \pi_2\, \langle s,\, t\rangle \ \equiv \ t \ : \ T[x/s]}\Sigma\text{-}\beta\text{-}2$$

We have a structural rule for equating Σ-types analogous to the one for Π-types. We have an observational rule for pairs which says that pairs are equal if their projections are equal. We also have computation (β) rules for pairs which explain how to project from canonical pairs.

In the implementation we extend the term syntax with constructors for the formation, introduction and elimination forms:

```
data Term = ...
          | Si Term Term
          | Pair Term Term
          | Fst Term
          | Snd Term
          | ...
```

We extend values with the formation and introduction forms. Like Π-types we use a closure for the body of Σ-types.

```
data Val = ...
         | VSi Val (Term,Env)
         | VPair Val Val
         | ...
```

For the elimination forms (the projections) we add neutral instances (for stuck projections) and functions to compute projections from pairs in introduction form.

```
data Ne = ...
        | NFst Ne
        | NSnd Ne
        | ...
```

The first line of each function to compute projections (in the case of a non-neutral pair) corresponds to the computation rules we added to equality.

```
vfst :: Val -> Val
vfst (VPair v _) = v
vfst (Ne n)      = Ne (NFst n)
vfst _           = error ``eek''

vsnd :: Val -> Val
vsnd (VPair _ v) = v
vsnd (Ne n)      = Ne (NSnd n)
vsnd _           = error ``eek''
```

We extend the evaluation and weakening for Σ-types:

```
eval (Si d r)     vs = VSi (eval d vs) (r,vs)
eval (Pair t1 t2) vs = VPair (eval t1 vs) (eval t2 vs)
eval (Fst t)      vs = vfst (eval t vs)
eval (Snd t)      vs = vsnd (eval t vs)

wk (VSi d (r,vs)) = VSi (wk d) (r, ewk vs)
wk (VPair v1 v2)  = VPair (wk v1) (wk v2)

nwk (NFst n)  = NFst (nwk n)
nwk (NSnd n)  = NSnd (nwk n)
```

Next we extend the equality checker. The first rule equates Σ-types as we equated Π-types and second equates inhabitants of Σ-types by taking their projections and observing the results.

```
eq (VStar,(VSi d1 (r1,vs1),VPi d2 (r2,vs2))) = do
  eq (VStar,(d1,d2))
  eq (VStar,(eval r1 (ewk vs1 :< Ne (NV (0,d1))),
             eval r2 (ewk vs2 :< Ne (NV (0,d2)))))
eq (VSi d (r,vs),(p1,p2)) = do
  eq (d,(vfst p1,vfst p2))
  eq (eval r (vs :< vfst p1),(vsnd p1,vsnd p2))
```

We also add instances for neutral projections to the neutral equality checker. Here we just compare the underlying pairs. Doing so passes back their type and we use this to return the appropriate type of the projection.

```
neq (NFst n1,NFst n2) = do
  VSi d (r,vs) <- neq (n1,n2)
  return d
neq (NSnd n1,NSnd n2) = do
  VSi d (r,vs) <- neq (n1,n2)
  check (vs :< Ne (NFst n1)) (r,VStar)
```

Non-neutral projections have been computed out in values so we do not check the β-rules explicitly in the equality checker. Instead we choose a representative of the equivalence class: the actual component of the pair being projected.

We now extend the `infer` and `check` functions to accommodate Σ-types. First we infer the type of Σ-types as we did for Π. This corresponds to to the formation rule.

```
infer vs (Si d r)   = do
  vd <- check vs (d,VStar)
  check (ewk vs :< Ne (NV (0,vd))) (r,VStar)
  return VStar
```

We cannot infer the type of a pair as we would need to have supplied the range of the Σ-type explicitly. We can only check the types of pairs. This means we must provide non-neutral pairs with an explicit type using `Coe`. This is analogous to Π: here we cannot infer the range, there we cannot infer the domain.

```
infer vs (Pair s t) = Nothing


check vs (Pair t1 t2,VSi d (r,vs')) = do
  v1  <- check vs (t1,d)
  rty <- check (vs' :< v1) (r,VStar)
  v2  <- check vs (t2,rty)
  return (VPair v1 v2)
```

Lastly we add rules to `infer` to cope with projections, these correspond to the elimination rules for pairs.

```
infer vs (Fst t) = do
  VSi d (r,vs') <- infer vs t
  return d
infer vs (Snd t) = do
  VSi d (r,vs') <- infer vs t
  v1 <- check vs (Fst t,d)
  check (vs' :< v1) (r,VStar)
```

2.4 Chapter summary

In this chapter I have given the rules of type theory with the rule type is a type and implemented a type checker for it in Haskell. The rules and the typechecker were then extended to include the empty type, the unit type and Σ-types.

I do not claim that the algorithm presented here corresponds as closely as possible to the formal rules. Nor do I intend to prove the algorithm correct. Its presence in this thesis is purely pedagogical. Its purpose is to serve as an introduction to type theory by means of a relatively short implementation whose design is similar to the normalisers in later chapters.

Chapter 3

Simply typed combinatory calculi

In this chapter we consider a system based on typed combinatory logic. This system is very similar to simply typed λ-calculus. A translation from one to the other is easy to give. The main difference is the replacement of the use of variables and binding with low level primitives (combinators) for manipulating environments. The K combinator discards the environment and the S combinator distributes it to its arguments. Combinatory logic was first suggested by Schönfinkel [77]. We choose to present it here so as to separate the explanation of the approach to guaranteeing normalisation from the technicalities of variables and binding.

We will introduce the meta language of Epigram in which we will work and the technique for showing normalisation and termination for a simple object language. Then we extend the core of combinatory logic to include finite products, finite coproducts and natural numbers. In appendix A the full code of a normaliser for combinatory logic is given in Agda syntax. The Agda code for the extensions are online [23].

Combinatory Logic with natural numbers corresponds to so-called system T. Gödel introduced system T in 1941, finally publishing it in 1958 [48]. In 1967 Tait [80] presented a version with functions defined by basic combinators. In the same paper he introduced strong computability (which he called reducibility) to show normalisation of the calculus. It is Tait's formulation that we consider here.

The aim of this chapter is to write a normaliser that is guaranteed to be

correct and to terminate. Firstly we consider a Haskell implementation that preserves types using Generalized Algebraic Data Types [73] (GADTs). Then we move to Epigram to show that this normaliser is terminating, sound and complete. We carry out the construction for combinatory logic in full before considering extensions.

The approach to showing normalisation is as follows:

1. Define the typed syntax of the object language.
2. Define the equational theory.
3. Define a first order normaliser as a simple recursive function.
4. Prove termination of the normaliser.
5. Use the Bove-Capretta technique [20] to yield a structurally recursive normaliser using our original definition and the termination proof.

In Haskell we can only do steps 1 and 3. We do this anyway first to ease the transition from (the likely familiar) Haskell to (the likely unfamiliar) Epigram. After this we carry out all the steps in Epigram.

Relationship to published paper This chapter is based on the paper "Tait in one big step" [9], written jointly with Thorsten Altenkirch. The proof in the paper has been streamlined and formalised, and the language considered in the paper (combinatory system T) extended with finite products and coproducts. The paper did not include a Haskell implementation.

3.1 Combinatory logic in Haskell

Combinatory logic can be represented by three combinators `K`, `S` and (left associative) application `:@`. We index by types to guarantee that application is always well typed. This rules out nonsense terms like `S :@ S :@ S`. We can define the well typed combinatory terms in Haskell as a GADT:

```
data Tm σ where
  K    :: Tm (σ → τ → σ)
  S    :: Tm ((σ → τ → ρ) → (σ → τ) → σ → ρ)
  (:@) :: Tm (σ → τ) → Tm σ → Tm τ
```

We cannot define the the equational theory as a GADT in Haskell as this would require indexing by data instead of Haskell types. Instead we fall back to the formal notation from the previous chapter. We define β-equality on combinatory terms as the least congruence generated by the syntax with the following computation rules for S and K respectively.

$$\frac{\vdash x : \sigma \quad \vdash y : \tau}{\vdash K\,x\,y = x} \qquad \frac{\vdash x : \sigma \to \tau \to \rho \quad \vdash y : \sigma \to \tau \quad \vdash z : \sigma}{\vdash S\,x\,y\,z = x\,z\,(y\,z)}$$

It helps to think of the last argument to S and K as an environment being passed around: in the case of K it is discarded; in the case of S it is distributed through application.

Next we define normal forms. These are terms that have been computed as much as possible. They are applications of the combinators `S` and `K` that have not received enough arguments to compute (less than two in the case of `K` and less than three in the case of `S`).

```
data Nf σ where
  K0 :: Nf (σ → τ → σ)
  K1 :: Nf σ → Nf (τ → σ)
  S0 :: Nf ((σ → τ → ρ) → (σ → τ) → σ → ρ)
  S1 :: Nf (σ → τ → ρ) → Nf ((σ → τ) → σ → ρ)
  S2 :: Nf (σ → τ → ρ) → Nf (σ → τ) → Nf (σ → ρ)
```

To define the normaliser that follows we are not forced to define a specific type of normal forms as we can just perform computation on the syntactic terms. However, it is often useful to do this. For example we can exploit the fact that certain types are uninhabited.

First we consider the normaliser function:

```
nf :: Tm σ → Nf σ
nf K         = K0
nf S         = S0
nf (t :@ u) = nf t @@ nf u
```

The terms `K` and `S` are just replaced by their normal forms `K0` and `S0` and for application we normalise the function and its argument and then call the application operation for normal forms which we define next.

We define an application operation for (well typed) normal forms. It performs the appropriate computation when the combinators receive enough arguments. Notice that this is the source of nested recursion.

```
(@@) :: Nf (σ → τ) → Nf σ → Nf τ
K0     @@ x = K1 x
K1 x   @@ y = x
S0     @@ x = S1 x
S1 x   @@ y = S2 x y
S2 x y @@ z = (x @@ z) @@ (y @@ z)
```

In the cases for `K0`, `S0` and `S1` we just accumulate arguments. In the case for `K1 x @@ y` the application computes to `x` as `K` returns its first argument discarding the second (the environment). In the case for `S2 x y @@ z` the first argument and second arguments are applied to the third argument (the environment) and then the results are applied to each other.

This type checks using the Haskell type checker but we want to go further. Firstly, Haskell has no termination checker so we cannot verify if a function is terminating, we can only test. Secondly, we want to show that the normaliser actually produces the normal form of its input. To do this formally, we would need to represent the equational theory in Haskell but we cannot do this easily because we cannot inductively define the equivalence relation as a GADT as we cannot index by data (combinatory terms in this case).

3.2 Syntax in Epigram

To show this algorithm is terminating and decides the equational theory we move to Epigram. Apart from the difference in syntax the biggest change is that before we were indexing by Haskell types; now we inductively define object level types and index by them. We can index by actual data. In Epigram inductive types are defined as 'data *type constructor* where *data constructors*' and constructors are usually written fully applied. In Epigram we present inductive definitions as natural deduction style rules. We use blue for type constructors, red for data constructors, purple for variables, and green for functions. The types we require are either base type ($\bullet$) or function type ($\sigma \to \tau$).

$$\underline{\text{data}} \quad \frac{}{\mathsf{Ty} : \star} \quad \underline{\text{where}} \quad \frac{}{\bullet : \mathsf{Ty}} \quad \frac{\sigma : \mathsf{Ty} \quad \tau : \mathsf{Ty}}{\sigma \to \tau : \mathsf{Ty}}$$

Terms are indexed by object level types. In the definitions of the data constructors for terms we do not need to explicitly declare the object level types (σ, τ, etc.) as their type (Ty) can be inferred. We use juxtaposition for application to reduce syntactic clutter.

$$\underline{\text{data}} \quad \frac{\sigma \;:\; \mathsf{Ty}}{\mathsf{Tm}\,\sigma \;:\; \star} \quad \underline{\text{where}} \quad \frac{}{\mathsf{K} \;:\; \mathsf{Tm}\,(\sigma{\to}\tau{\to}\sigma)}$$

$$\frac{}{\mathsf{S} \;:\; \mathsf{Tm}\,((\sigma{\to}\tau{\to}\rho){\to}(\sigma{\to}\tau){\to}\sigma{\to}\rho)} \qquad \frac{t \;:\; \mathsf{Tm}\,(\sigma{\to}\tau) \quad u \;:\; \mathsf{Tm}\,\sigma}{t\,u \;:\; \mathsf{Tm}\,\tau}$$

Equational theory

We present the equational theory as an inductively defined relation. As we can index by actual data we can index by terms. The type constructor takes two arguments; one for each side of the equation. The data constructors are really the names of their rules (usually written at the side) and their types are the rules themselves. The first two rules are the computation rules for K and S. The remaining rules express closure under the conditions necessary for our relation to be considered an equivalence relation, reflexivity, symmetry, transitivity and congruence of application. Later, as a consequence of normalisation, we will show that this equivalence relation is decidable.

$$\underline{\text{data}} \quad \frac{t,\,u \;:\; \mathsf{Tm}\,\sigma}{t \simeq u \;:\; \mathsf{Prop}} \quad \underline{\text{where}} \quad \frac{}{\mathsf{cK} \;:\; \mathsf{K}\,x\,y \simeq x} \quad \frac{}{\mathsf{cS} \;:\; \mathsf{S}\,x\,y\,z \simeq x\,z\,(y\,z)}$$

$$\frac{}{\mathsf{crefl} \;:\; t \simeq t} \qquad \frac{p \;:\; t \simeq u}{\mathsf{csym}\,p \;:\; u \simeq t} \qquad \frac{p \;:\; t \simeq u \quad q \;:\; u \simeq v}{\mathsf{ctrans}\,p\,q \;:\; t \simeq v}$$

$$\frac{p \;:\; t \simeq v \quad q \;:\; u \simeq w}{\mathsf{ccong}\,p\,q \;:\; t\,u \simeq v\,w}$$

Instead of defining an ordinary data type for the equational theory that inhabits $\star$ this type inhabits a universe Prop of propositions. We adopt this convention to indicate that it has no computationally interesting content and we would like it to be proof irrelevant. The current versions of Epigram and Agda do not have appropriate support for this so we are just adopting a voluntary notational discipline here. Indeed we are just using Prop as a synonym for $\star$. For the equational theory we use this notation to indicate that we are not

interested in which inhabitant of a particular equation we have, just whether or not there is one. Later the use of this discipline will be more important.

3.3 Normal forms

We define normal forms analogously to the Haskell definition. Note that the superscripts in the names are just part of names.

$$\underline{\text{data}}\ \frac{\sigma\ :\ \mathsf{Ty}}{\mathsf{Nf}\,\sigma\ :\ \star}\ \underline{\text{where}}\ \frac{}{\mathsf{nK}\ :\ \mathsf{Nf}\,(\sigma{\to}\tau{\to}\sigma)}\quad \frac{u\ :\ \mathsf{Nf}\,\sigma}{\mathsf{nK}^1\,u\ :\ \mathsf{Nf}\,(\tau{\to}\sigma)}$$

$$\frac{}{\mathsf{nS}\ :\ \mathsf{Nf}\,((\sigma{\to}\tau{\to}\rho){\to}(\sigma{\to}\tau){\to}\sigma{\to}\rho)}\quad \frac{u\ :\ \mathsf{Nf}\,(\sigma{\to}\tau{\to}\rho)}{\mathsf{nS}^1\,u\ :\ \mathsf{Nf}\,((\sigma{\to}\tau){\to}\sigma{\to}\rho)}$$

$$\frac{u\ :\ \mathsf{Nf}\,(\sigma{\to}\tau{\to}\rho)\quad u'\ :\ \mathsf{Nf}\,\sigma{\to}\tau}{\mathsf{nS}^2\,u\,u'\ :\ \mathsf{Nf}\,\sigma{\to}\rho}$$

We define an embedding operation $\ulcorner - \urcorner$ from normal forms to terms.

$$\frac{}{\ulcorner - \urcorner\ :\ \mathsf{Nf}\,\sigma \to \mathsf{Tm}\,\sigma}$$

$$\begin{aligned}
\ulcorner \mathsf{nK} \urcorner &= \mathsf{K}\\
\ulcorner \mathsf{nK}^1\,u \urcorner &= \mathsf{K}\ulcorner u \urcorner\\
\ulcorner \mathsf{nS} \urcorner &= \mathsf{S}\\
\ulcorner \mathsf{nS}^1\,u \urcorner &= \mathsf{S}\ulcorner u \urcorner\\
\ulcorner \mathsf{nS}^2\,u\,u' \urcorner &= \mathsf{S}\ulcorner u \urcorner\ulcorner u' \urcorner
\end{aligned}$$

3.4 Recursive normalisation

Our goal is to define a normalisation function

$$\frac{}{\mathbf{nf}\ :\ \mathsf{Tm}\,\sigma \to \mathsf{Nf}\,\sigma}$$

which should have the following properties:

soundness Normalisation takes convertible terms to identical normal forms

$$\frac{a \simeq a'}{\mathbf{nf}\ a = \mathbf{nf}\ a'}$$

completeness Terms are convertible to their normal forms

$$a \simeq \ulcorner \mathbf{nf}\ a \urcorner$$

As a consequence we obtain that convertibility corresponds to having the same normal form:

$$t \simeq u \iff \mathbf{nf}\, t = \mathbf{nf}\, u$$

As a first approximation we write **nf** and its helper function @ to apply normal functions to normal arguments as general recursive functions analogously to the Haskell definitions.

$$\frac{t \;:\; \mathsf{Tm}\,\sigma}{\mathbf{nf}\; t \;:\; \mathsf{Nf}\,\sigma}$$

$$\begin{array}{lcl} \mathbf{nf}\;\; \mathsf{K} & = & \mathsf{nK} \\ \mathbf{nf}\;\; \mathsf{S} & = & \mathsf{nS} \\ \mathbf{nf}\;\; (t\; u) & = & (\mathbf{nf}\; t) \mathbin{@} (\mathbf{nf}\; u) \end{array}$$

$$\frac{f \;:\; \mathsf{Nf}\,(\sigma \to \tau) \qquad a \;:\; \mathsf{Nf}\,\sigma}{f \mathbin{@} a \;:\; \mathsf{Nf}\,\tau}$$

$$\begin{array}{lcl} \mathsf{nK} \mathbin{@} x & = & \mathsf{nK}^1\, x \\ (\mathsf{nK}^1\, x) \mathbin{@} y & = & x \\ \mathsf{nS} \mathbin{@} x & = & \mathsf{nS}^1\, x \\ (\mathsf{nS}^1\, x) \mathbin{@} y & = & \mathsf{nS}^2\, x\, y \\ (\mathsf{nS}^2\, x\, y) \mathbin{@} z & = & (x \mathbin{@} z) \mathbin{@} (y \mathbin{@} z) \end{array}$$

This is a definition in partial type theory. In Epigram we cannot write it as a function, instead we would write it as a functional relation. In Agda we can write it as a function and just assume it terminates. It does terminate; we prove this in the next section.

3.5 Big-Step semantics

Next we want to show termination of our normaliser. To do this we translate our function definitions to inductive relations between the function's inputs and outputs. This is a so-called big-step semantics. We then show that termination in terms of the big-step semantics. When we have done this we can use our termination proof to define a structurally recursive version of our normaliser using the Bove-Capretta technique [20]. Before proceeding we carry out the whole process for a simpler example.

The Bove-Capretta technique

This technique exploits the property that many functions are structurally recursive over their own call graph. As a simple example, instead of a normaliser we consider a function on natural numbers that always returns zero. Natural numbers are defined as follows:

$$\underline{\text{data}}\ \ \frac{}{\mathsf{Nat} : \star}\ \ \underline{\text{where}}\ \ \frac{}{\mathsf{zero} : \mathsf{Nat}}\ \ \frac{n : \mathsf{Nat}}{\mathsf{suc}\, n : \mathsf{Nat}}$$

We define the function using nested recursion so we can demonstrate the technique:

$$\frac{n : \mathsf{Nat}}{\mathbf{f}\, n : \mathsf{Nat}}$$

$$\begin{aligned} \mathbf{f}\ \ \mathsf{zero} &= \mathsf{zero} \\ \mathbf{f}\ \ (\mathsf{suc}\, n) &= \mathbf{f}\, (\mathbf{f}\, n) \end{aligned}$$

While it is obvious to us that $\mathbf{f}$ is total, it is not obviously structurally recursive due to the nested recursive call. However, we can inductively define the graph of the function as a relation—its big-step semantics:

$$\underline{\text{data}}\ \ \frac{n, n' : \mathsf{Nat}}{\mathsf{f}\, n \Downarrow n' : \mathsf{Prop}}\ \ \underline{\text{where}}$$

$$\frac{}{\mathsf{fz} : \mathsf{f}\, \mathsf{zero} \Downarrow \mathsf{zero}}\qquad \frac{p : \mathsf{f}\, n \Downarrow n' \quad p' : \mathsf{f}\, n' \Downarrow n''}{\mathsf{fs}\, p\, p' : \mathsf{f}\, (\mathsf{suc}\, n) \Downarrow n''}$$

We adopt the convention that the relation corresponding to the recursive definition of $\mathbf{f} : \mathsf{Nat} \to \mathsf{Nat}$ is written as $\mathsf{f} - \Downarrow - : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Prop}$. We can now define a structurally recursive version of $\mathbf{f}$ called $\mathbf{f}^{\mathbf{str}}$ which takes an extra argument (the derivation of the graph) and returns a (dependent) pair of the intended output of the function and a proof that it agrees with the graph. We note that the argument p in the function declaration is propositional and its indices are not available for projection. In particular we cannot just project out m.

$$\frac{p : \mathsf{f}\, n \Downarrow m}{\mathbf{f}^{\mathbf{str}}\, n\, p : \Sigma\, n' : \mathsf{Nat}\, .\, n' = m}$$

$$\begin{aligned} \mathbf{f}^{\mathbf{str}}\ \ \mathsf{zero}\ \ \ \mathsf{fz}\ \ \ \ \ \ \ \ &= (\mathsf{zero}, \mathsf{refl}) \\ \mathbf{f}^{\mathbf{str}}\ \ (\mathsf{suc}\, n)\ \ (\mathsf{fs}\, p\, p')\ \ \underline{\text{with}}\ \ &\mathbf{f}^{\mathbf{str}}\, n\, p \\ \qquad\qquad\qquad\qquad\qquad\qquad\ \ \mid\ \ &(n', \mathsf{refl}) \quad = \quad \mathbf{f}^{\mathbf{str}}\, n'\, p' \end{aligned}$$

The $\underline{\text{with}}$ allows us to pattern match on an intermediate result much like `case` does in Haskell. An important technical detail is that in the $\underline{\text{with}}$ we pattern match on refl. This unifies n' with the 'output' of p and the 'input' of p'.

These arguments appear in the types of p and p' as we can see by looking at the fs rule above. If we did not do this (instead just pattern matching on, for example, (n', p'') in the with) the application $\mathbf{f}^{\mathbf{str}}\ n'\ p'$ would not be well typed.

We establish that $\mathsf{f} - \Downarrow -$ is total by proving:

Theorem 2.

$$\frac{n\ :\ \mathsf{Nat}}{\mathsf{f}\ n\ \Downarrow\ \mathsf{zero}}$$

Proof. By induction on n. □

If we had not defined the graph to be propositional and this theorem had no computational content we could just execute the theorem to produce our desired output (although in this case the output is just zero). The idea is that we will use such theorems to justify the termination of function definitions but use the computational behaviour of the original function. We keep computation and termination separate. Now we can redefine **f** as a structurally recursive function: [1]

$$\frac{n\ :\ \mathsf{Nat}}{\mathbf{f}\ n\ :\ \mathsf{Nat}} \qquad \mathbf{f}\quad n\quad =\quad \pi_0\,(\mathbf{f}^{\mathbf{str}}\ n\,(\mathbf{theorem2}\ n))$$

Big-step semantics of nf

We now return to our normaliser. First we write out the graph of the normaliser. There is no reason why this could not be automated but for now we must do it by hand. It should also be noted that is easy to do this as the function definition is first order.

$$\underline{\text{data}}\quad \frac{t\ :\ \mathsf{Tm}\,\sigma \quad n\ :\ \mathsf{Nf}\,\sigma}{\mathsf{nf}\ t \Downarrow\ n\ :\ \mathsf{Prop}}\quad \underline{\text{where}}$$

$$\frac{}{\mathsf{rK}\ :\ \mathsf{nf}\,\mathsf{K} \Downarrow\ \mathsf{nK}} \qquad \frac{}{\mathsf{rS}\ :\ \mathsf{nf}\,\mathsf{S} \Downarrow\ \mathsf{nS}}$$

$$\frac{p\ :\ \mathsf{nf}\ t \Downarrow\ t' \quad q\ :\ \mathsf{nf}\ u \Downarrow\ u' \quad r\ :\ t' \mathbin{@} u' \Downarrow v}{\mathsf{app}\ p\ q\ r\ :\ \mathsf{nf}\ t\ u \Downarrow\ v}$$

[1] When using theorems as proof terms in programs we write **theoremi** where **i** is the number of the theorem.

Secondly we write out the graph of @.

$$\underline{\text{data}} \quad \frac{m \;:\; \mathsf{Nf}\,(\sigma{\to}\tau) \quad n \;:\; \mathsf{Nf}\,\sigma \quad o \;:\; \mathsf{Nf}\,\tau}{m \mathbin{@} n \Downarrow o \;:\; \mathsf{Prop}} \quad \underline{\text{where}}$$

$$\frac{}{\mathsf{rK}^1 \;:\; \mathsf{K} \mathbin{@} x \Downarrow \mathsf{nK}^1\, x} \qquad \frac{}{\mathsf{rK}^2 \;:\; \mathsf{nK}^1\, x \mathbin{@} y \Downarrow x}$$

$$\frac{}{\mathsf{rS}^1 \;:\; \mathsf{nS} \mathbin{@} x \Downarrow \mathsf{nS}^1\, x} \qquad \frac{}{\mathsf{rS}^2 \;:\; \mathsf{nS}^1\, x \mathbin{@} y \Downarrow \mathsf{nS}^2\, x\, y}$$

$$\frac{p \;:\; x \mathbin{@} z \Downarrow u \quad q \;:\; y \mathbin{@} z \Downarrow v \quad r \;:\; u \mathbin{@} v \Downarrow w}{\mathsf{rS}^3\, p\, q\, r \;:\; \mathsf{nS}^2\, x\, y \mathbin{@} z \Downarrow w}$$

Having defined the big-step semantics we want to prove the following theorem which expresses that the normaliser terminates. Notice that instead of Σ for dependent pairs here we use $\exists$. This is a propositional dependent pair, we are not allowed to just project out the witness. We want to use this theorem to justify the termination of our original normaliser function, not to use it as a normaliser directly. The reason is that we want to use the computational behaviour of our original näive definition, instead *running* the proof.

Theorem 3. *All terms are normalising.*

$$\frac{t \;:\; \mathsf{Tm}\,\sigma}{\mathbf{norm}\; t \;:\; \exists n{:}\mathsf{Nf}\,\sigma.\, \mathsf{nf}\; t \Downarrow\; n}$$

Attempts to prove it by induction on the structure of terms or types fail. First, it is clear that we would need the following lemma which is the thing we cannot prove by induction on normal forms:

Lemma 1. *All applications are normalising.*

$$\frac{f \;:\; \mathsf{Nf}\,(\sigma \to \tau) \quad a \;:\; \mathsf{Nf}\,\sigma}{\exists n{:}\mathsf{Nf}\,\tau.\, f \mathbin{@} a \Downarrow n}$$

Proof attempt. By induction on f. The problematic case is $\mathsf{nS}^2\, x\, y$. Given an argument $z : \mathsf{Nf}\,\sigma$ we have by inductive hypotheses: $\exists f : \mathsf{Nf}\,(\tau \to \rho).\, x \mathbin{@} z \Downarrow f$ and $\exists a : \mathsf{Nf}\,\tau.\, y \mathbin{@} z \Downarrow a$. But we cannot show that $\exists n : \mathsf{Nf}\,\rho.\, f \mathbin{@} a \Downarrow n$. Looking back at the function definition we see that this is the source of the nested application $(x \mathbin{@} z) \mathbin{@} (y \mathbin{@} z)$. What we need is that we can keep applying safely until we reach base type. This is where strong computability comes in.

3.6 Strong computability

Proving normalisation by induction on the structure of terms fails. To overcome this we follow Tait and use *strong computability*[2]. The central idea is to interpret object level function space as meta level function space.

We define a strong computability predicate on normal forms $\mathbf{SCN}$ by induction over types.

$$\frac{t \;:\; \mathsf{Nf}\,\sigma}{\mathbf{SCN}_\sigma\; t \;:\; \mathsf{Prop}}$$

$$\begin{aligned}\mathbf{SCN}_\bullet\; t &= \mathbf{True}\\ \mathbf{SCN}_{\sigma\to\tau}\; t &= \forall u{:}\mathsf{Nf}\,\sigma.\mathbf{SCN}_\sigma\; u \to\\ &\quad \exists n{:}\mathsf{Nf}\,\tau.t \mathbin{@} u \Downarrow n \wedge \mathbf{SCN}_\tau n \wedge \ulcorner t \urcorner \ulcorner u \urcorner \simeq \ulcorner n \urcorner\end{aligned}$$

The arrow case basically says that a function is strongly computable if when give a strongly computable input you get a strongly computable output. We extend this to include that the function, its input and output are related by the big-step semantics and the equational theory. The former to use to for our termination proof and the latter so we don't have to prove it separately. A strongly computable term is a term which reduces to a strongly computable normal form — this is represented in Epigram by defining the predicate $\mathbf{SC}$:

$$\frac{t \;:\; \mathsf{Tm}\,\sigma}{\mathbf{SC}_\sigma\; t \;:\; \mathsf{Prop}}$$

$$\mathbf{SC}_\sigma\; t \;=\; \exists n{:}\mathsf{Nf}\,\sigma\,.\,\mathsf{nf}\; t \Downarrow\; n \wedge \mathbf{SCN}_\sigma\; n \wedge t \simeq \ulcorner n \urcorner$$

Note that, again, we are not just showing strong computability on its own. We are also showing that the normal form produced respects the big-step semantics and completeness of the normaliser. to prove all three properties at the same time.

Our main technical lemma is that all normal forms are strongly computable:

Proposition 1.

$$\forall n{:}\mathsf{Nf}\,\sigma\,.\,\mathbf{SCN}_\sigma\; n$$

Proof. By induction on n. We show the previously problematic case for $\mathsf{nS}^2\; x\; y$ ommitting the tertiary (completeness) conjunct: We assume a strongly computable argument $\mathbf{SCN}_\sigma\; z$. By induction hypothesis we have that $\mathbf{SCN}_{(\sigma\to\tau\to\rho)}$

[2]It is strong in the sense that it strengthens the inductive hypothesis not in the sense of strong normalisation.

and $\mathbf{SCN}_{(\tau\to\rho)}$. By definition of **SCN** this implies that there exists an $n : \mathsf{Nf}\,(\tau\to\rho)$ such that $x \mathbin{@} z \Downarrow n$ (1) and $\mathbf{SCN}_{(\tau\to\rho)}\, n$ (2) and there exists an $m : \mathsf{Nf}\,\tau$ such that $y \mathbin{@} z \Downarrow m$ (3) and $\mathbf{SCN}_\tau\, m$ (4). By (2, 4) and the definition of **SCN** we get that there exists an $h : \mathsf{Nf}\,\rho$ such that $n \mathbin{@} m \Downarrow h$ (5) and $\mathbf{SCN}_\rho\, h$ (6). Then $\mathsf{nS}^2\, x\, y \mathbin{@} z \Downarrow h$ by rS^3 with (1, 3, 5), and $\mathbf{SCN}_\rho\, h$ (6). □

The main theorem is now an easy consequence:

Proposition 2. *All terms are Strongly Computable.*

$$\forall t : \mathsf{Tm}\,\sigma\,.\,\mathbf{SC}_\sigma\, t$$

Proof. By induction on t. □

Normalisation is now an obvious corollary:

Corollary 1. *All terms are normalising.*

$$\overline{\mathbf{norm}\ :\ \forall t : \mathsf{Tm}\,\sigma\,.\exists n : \mathsf{Nf}\,\sigma\,.\,\mathsf{nf}\ t \Downarrow\ n}$$

3.7 Structurally recursive normalisation

We have a non-computational proof of normalisation. We want a bona fide function that computes normal forms. To actually get a type theoretic function we need to use the Bove-Capretta technique.

We first define structurally recursive versions of the functions that take an extra argument (the proof) and return a pair of the desired output and a proof that it agrees with the proof, as we did for the function that always returned zero.

$$\frac{t\ :\ \mathsf{Tm}\,\sigma \quad p\ :\ \mathsf{nf}\ t \Downarrow\ n}{\mathbf{nf}^{\mathbf{str}}\ t\ p\ :\ \Sigma n' : \mathsf{Nf}\,\sigma\,.\,n' = n}$$

$$\begin{array}{llll}
\mathbf{nf}^{\mathbf{str}} & \mathsf{K} & \mathsf{rK} & = \quad (\mathsf{nK};\ \mathsf{refl}) \\
\mathbf{nf}^{\mathbf{str}} & \mathsf{S} & \mathsf{rS} & = \quad (\mathsf{nS};\ \mathsf{refl}) \\
\mathbf{nf}^{\mathbf{str}} & (t\ u) & (\mathsf{app}\ p\ p'\ p'') & \underline{\text{with}}\quad \mathbf{nf}^{\mathbf{str}}\ t\ p \mid \mathbf{nf}^{\mathbf{str}}\ u\ p' \\
 & \multicolumn{3}{l}{\mid (f;\ \mathsf{refl}) \mid (a;\ \mathsf{refl}) \ =\ \mathbf{napp}^{\mathbf{str}}\ f\ a\ p''}
\end{array}$$

$$\frac{f \,:\, \mathsf{Nf}\,(\sigma{\to}\tau) \quad a \,:\, \mathsf{Nf}\,\sigma \quad p \,:\, f \mathbin{@} a \Downarrow n}{\mathbf{napp^{str}}\, f\; a\; p \;:\; \Sigma n' {:} \mathsf{Nf}\,\tau\,.\, n' \;=\; n}$$

$$\begin{array}{llllll}
\mathbf{napp^{str}} & (\mathsf{nK}) & x & \mathsf{rK}^1 & = & (\mathsf{nK}^1\, x;\, \mathsf{refl}) \\
\mathbf{napp^{str}} & (\mathsf{nK}^1\, x) & y & \mathsf{rK}^2 & = & (x;\, \mathsf{refl}) \\
\mathbf{napp^{str}} & (\mathsf{nS}) & x & \mathsf{rS}^1 & = & (\mathsf{nS}^1\, x;\, \mathsf{refl}) \\
\mathbf{napp^{str}} & (\mathsf{nS}^1\, x) & y & \mathsf{rS}^2 & = & (\mathsf{nS}^2\, x\; y;\, \mathsf{refl}) \\
\mathbf{napp^{str}} & (\mathsf{nS}^2\, x\; y) & z & (\mathsf{rS}^2\, p\; p'\; p'') & \underline{\text{with}} & \mathbf{napp^{str}}\, x\; z\; p \,|\, \mathbf{napp^{str}}\, y\; z\; p' \\
 & \multicolumn{5}{l}{\quad |\,(f;\, \mathsf{refl})\,|\,(a;\, \mathsf{refl}) \;=\; \mathbf{napp^{str}}\, f\; a\; p''}
\end{array}$$

Using the normalisation theorem (our termination proof) we can produce instances of the big-step semantics (witnesses of termination) and then implement a terminating version of our normalisation function with the same type as the original one:

$$\begin{aligned}
\mathbf{nf} \;&:\; \mathsf{Tm}_{\,\sigma} \to \mathsf{Nf}_{\,\sigma} \\
\mathbf{nf} \quad t \;&=\; \pi_0\,(\mathbf{nf^{str}}\; t\;(\pi_0\,(\mathbf{norm}\; t)))
\end{aligned}$$

Proposition 3. $\mathbf{nf}$ *is a normalisation function, i.e.*

$$\frac{a \simeq a'}{\mathbf{nf}\; a = \mathbf{nf}\; a'}(1) \qquad \frac{}{a \simeq \ulcorner \mathbf{nf}\; a \urcorner}(2)$$

Proof. (1) by induction on the derivation $a \simeq a'$. (2) follows from proposition 2 and the definition of $\mathbf{nf^{str}}$. □

3.8 Extensions

In this section we extend our system of combinatory logic with finite products, finite coproducts and finally natural numbers.

Finite Products

We add the unit type and binary products to our system. First we unit and binary product types to the definition of types:

$$\frac{}{1 \;:\; \mathsf{Ty}} \qquad \frac{\sigma \;:\; \mathsf{Ty} \qquad \tau \;:\; \mathsf{Ty}}{\sigma \times \tau}$$

Next we extend the syntax with the constant void (the only element of the unit type) and combinators for pairing and projection:

$$\frac{}{[] \;:\; \mathsf{Tm}\,1} \qquad \frac{}{\mathsf{pr} \;:\; \mathsf{Tm}\,(\sigma \to \tau \to (\sigma \times \tau))}$$

$$\frac{}{\mathsf{fst} \;:\; \mathsf{Tm}\,((\sigma \times \tau) \to \sigma)} \qquad \frac{}{\mathsf{snd} \;:\; \mathsf{Tm}\,((\sigma \times \tau) \to \tau)}$$

We add two computation rules to the equational theory:

$$\frac{}{\mathsf{cfst} \;:\; \mathsf{fst}\,(\mathsf{pr}\;a\;b) \simeq a} \qquad \frac{}{\mathsf{csnd} \;:\; \mathsf{snd}\,(\mathsf{pr}\;a\;b) \simeq b}$$

Normal forms are extended accordingly, including pairing applied to zero, one and two arguments. We omit the trivial extension of the embedding.

$$\frac{}{[]^{\mathsf{n}} \;:\; \mathsf{Nf}\,1} \qquad \frac{}{\mathsf{npr} \;:\; \mathsf{Nf}\,(\sigma \to \tau \to (\sigma \times \tau))}$$

$$\frac{a \;:\; \mathsf{Nf}\,\sigma}{\mathsf{npr}^1\;a \;:\; \mathsf{Nf}\,(\tau \to (\sigma \times \tau))} \qquad \frac{a \;:\; \mathsf{Nf}\,\sigma \qquad b \;:\; \mathsf{Nf}\,\tau}{\mathsf{npr}^2\;a\;b \;:\; \mathsf{Nf}\,(\sigma \times \tau)}$$

$$\frac{}{\mathsf{nfst} \;:\; \mathsf{Nf}\,((\sigma \times \tau) \to \sigma)} \qquad \frac{}{\mathsf{nsnd} \;:\; \mathsf{Nf}\,((\sigma \times \tau) \to \tau)}$$

The normaliser is extended as follows:

$$\frac{t \;:\; \mathsf{Tm}\,\sigma}{\mathbf{nf}\;t \;:\; \mathsf{Nf}\,\sigma}$$

$$\begin{array}{lcl}
\ldots & & \\
\mathbf{nf} \;\; [] & = & []^{\mathsf{n}} \\
\mathbf{nf} \;\; \mathsf{pr} & = & \mathsf{npr} \\
\mathbf{nf} \;\; \mathsf{fst} & = & \mathsf{nfst} \\
\mathbf{nf} \;\; \mathsf{snd} & = & \mathsf{nsnd}
\end{array}$$

Notice that we have no neutral terms so the projections from pairs below always compute.

$$\frac{f \;:\; \mathsf{Nf}\,(\sigma \to \tau) \qquad a \;:\; \mathsf{Nf}\,\sigma}{f \mathbin{@} a \;:\; \mathsf{Nf}\,\tau}$$

$$\begin{array}{lcll}
\ldots & & & \\
\mathsf{npr} & @ & x & = \;\; \mathsf{npr}^1\;x \\
(\mathsf{npr}^1\;x) & @ & y & = \;\; \mathsf{npr}^2\;x\;y \\
\mathsf{nfst} & @ & (\mathsf{npr}^2\;x\;y) & = \;\; x \\
\mathsf{nsnd} & @ & (\mathsf{npr}^2\;x\;y) & = \;\; y
\end{array}$$

For strong computability of normal forms (which is defined by recursion on the type) we add clause for unit and binary products:

$$\frac{t \;:\; \mathsf{Nf}\,\sigma}{\mathbf{SCN}_\sigma\, t \;:\; \mathsf{Prop}}$$

$$\begin{aligned}
\mathbf{SCN}_1\, t &= \mathbf{True}\\
&\dots\\
\mathbf{SCN}_{\sigma\times\tau}\, p &= (\exists n{:}\mathsf{Nf}\,\sigma.\mathsf{nfst}\mathbin{@} p \Downarrow n \wedge \mathbf{SCN}_\sigma n \wedge \mathsf{fst}\ulcorner p\urcorner \simeq \ulcorner n\urcorner) \wedge\\
&\quad\ (\exists n{:}\mathsf{Nf}\,\tau.\mathsf{nsnd}\mathbin{@} p \Downarrow n \wedge \mathbf{SCN}_\tau n \wedge \mathsf{snd}\ulcorner p\urcorner \simeq \ulcorner n\urcorner)
\end{aligned}$$

The proofs are easily extended to accommodate the additions and so is the structurally recursive normaliser.

Finite Coproducts

We add the empty type and binary coproducts to our system. First we extend the definition of types with empty and binary coproducts:

$$\frac{}{0 \;:\; \mathsf{Ty}} \qquad \frac{\sigma \;:\; \mathsf{Ty} \qquad \tau \;:\; \mathsf{Ty}}{\sigma+\tau}$$

Next we extend the syntax with the eliminator for the empty type and combinators for injection and case analysis:

$$\frac{}{\mathsf{OE} \;:\; \mathsf{Tm}\,(0\to\sigma)} \qquad \frac{}{\mathsf{inl} \;:\; \mathsf{Tm}\,(\sigma\to(\sigma+\tau))} \qquad \frac{}{\mathsf{inr} \;:\; \mathsf{Tm}\,(\tau\to(\sigma+\tau))}$$

$$\frac{}{\mathsf{case} \;:\; \mathsf{Tm}\,((\sigma\to\rho)\to(\tau\to\rho)\to(\sigma+\tau)\to\rho)}$$

We add two computation rules for case to the equational theory:

$$\frac{}{\mathsf{ccasel} \;:\; \mathsf{case}\,l\,r\,(\mathsf{inl}\,s) \simeq l\,s} \qquad \frac{}{\mathsf{ccaser} \;:\; \mathsf{case}\,l\,r\,(\mathsf{inr}\,t) \simeq r\,t}$$

Normal forms are extended accordingly including partial applications of injections and case:

$$\frac{}{\mathsf{nOE} \;:\; \mathsf{Nf}\,(0\to\sigma)} \qquad \frac{}{\mathsf{ninl} \;:\; \mathsf{Nf}\,(\sigma\to(\sigma+\tau))} \qquad \frac{}{\mathsf{ninr} \;:\; \mathsf{Nf}\,(\tau\to(\sigma+\tau))}$$

$$\frac{n \;:\; \mathsf{Nf}\,\sigma}{\mathsf{ninl}^1\, n \;:\; \mathsf{Nf}\,(\sigma+\tau)} \qquad \frac{n \;:\; \mathsf{Nf}\,\tau}{\mathsf{ninr}^1\, n \;:\; \mathsf{Nf}\,(\sigma+\tau)}$$

$$\frac{}{\mathsf{ncase} \;:\; \mathsf{Nf}\,((\sigma\to\rho)\to(\tau\to\rho)\to(\sigma+\tau)\to\rho)}$$

$$\frac{l \;:\; \mathsf{Nf}\,(\sigma\to\rho)}{\mathsf{ncase}^1\, l \;:\; \mathsf{Nf}\,((\tau\to\rho)\to(\sigma+\tau)\to\rho)} \qquad \frac{l \;:\; \mathsf{Nf}\,(\sigma\to\rho) \qquad r \;:\; \mathsf{Nf}\,(\tau\to\rho)}{\mathsf{ncase}^2\, l\, r \;:\; \mathsf{Nf}\,((\sigma+\tau)\to\rho)}$$

The normaliser is extended as follows:

$$\frac{t \ : \ \mathsf{Tm}\,\sigma}{\mathbf{nf}\ t \ : \ \mathsf{Nf}\,\sigma}$$

$$\begin{array}{lcl}
\ldots & & \\
\mathbf{nf} \quad \mathsf{OE} & = & \mathsf{nOE} \\
\mathbf{nf} \quad \mathsf{inl} & = & \mathsf{ninl} \\
\mathbf{nf} \quad \mathsf{inr} & = & \mathsf{ninr} \\
\mathbf{nf} \quad \mathsf{case} & = & \mathsf{ncase}
\end{array}$$

Notice there is no right hand side in the definition of @ for the eliminator for the empty type (nOE) being applied to normal form of empty type, as there are no normal forms of empty type. The type checker is able to check that this case is impossible.

$$\frac{f \ : \ \mathsf{Nf}\,(\sigma \to \tau) \quad a \ : \ \mathsf{Nf}\,\sigma}{f \mathbin{@} a \ : \ \mathsf{Nf}\,\tau}$$

$$\begin{array}{lcll}
\ldots & & & \\
\mathsf{nOE} & @ & () & \\
\mathsf{ninl} & @ & x & = \quad \mathsf{ninl}^1\ x \\
\mathsf{ninr} & @ & x & = \quad \mathsf{ninr}^1\ x \\
\mathsf{ncase} & @ & l & = \quad \mathsf{ncase}^1\ l \\
(\mathsf{ncase}^1\ l) & @ & r & = \quad \mathsf{ncase}^2\ l\ r \\
(\mathsf{ncase}^2\ l\ r) & @ & (\mathsf{ninl}^1\ n) & = \quad l\ n \\
(\mathsf{ncase}^2\ l\ r) & @ & (\mathsf{ninr}^1\ n) & = \quad r\ n
\end{array}$$

For strong computability of normal forms (which is defined by recursion on the type) we add clauses for the empty type and binary coproducts:

$$\frac{t \ : \ \mathsf{Nf}\,\sigma}{\mathbf{SCN}_\sigma\ t \ : \ \mathsf{Prop}}$$

$$\begin{array}{lcl}
\mathbf{SCN}_0\ t & = & \mathbf{False} \\
\ldots & & \\
\mathbf{SCN}_{\sigma+\tau}\ (\mathsf{ninl}^1\ x) & = & \mathbf{SCN}_\sigma\ x \\
\mathbf{SCN}_{\sigma+\tau}\ (\mathsf{ninr}^1\ x) & = & \mathbf{SCN}_\tau\ x
\end{array}$$

We need an extra lemma. It is essentially a version of case for strongly computable normal forms:

Lemma 2.

$$\frac{\begin{array}{ll} l \;:\; \mathsf{Nf}\,(\sigma{\to}\rho) & p \;:\; \mathbf{SCN}\,l \\ r \;:\; \mathsf{Nf}\,(\tau{\to}\rho) & p' \;:\; \mathbf{SCN}\,r \\ c \;:\; \mathsf{Nf}\,(\sigma{+}\tau) & p'' \;:\; \mathbf{SCN}\,c \end{array}}{\mathbf{scase}\;p\;p'\;p'' \;:\; \exists\, n{:}\mathsf{Nf}\;\rho.\mathsf{ncase}^2\;l\;r\;@\;c \Downarrow n \wedge \mathbf{SCN}_\rho n \wedge \mathsf{case}\ulcorner l\urcorner\ulcorner r\urcorner\ulcorner c\urcorner \simeq \ulcorner n\urcorner}$$

Proof. By case on c. □

The rest of the construction is easily extended to accommodate the additions of the empty type and binary coproducts.

Natural numbers

We add the base type N.

$$\frac{}{\mathsf{N} \;:\; \mathsf{Ty}}$$

Next we extend the syntax with zero and combinators for successor and primitive recursion:

$$\frac{}{\mathsf{zero} \;:\; \mathsf{Tm}\,\mathsf{N}} \qquad \frac{}{\mathsf{suc} \;:\; \mathsf{Tm}\,(\mathsf{N}{\to}\mathsf{N})} \qquad \frac{}{\mathsf{prec} \;:\; \mathsf{Tm}\,(\sigma{\to}(\mathsf{N}{\to}\sigma{\to}\sigma){\to}\mathsf{N}{\to}\sigma)}$$

We add two computation rules for primitive recursion to the equational theory:

$$\frac{}{\mathsf{cprecz} \;:\; \mathsf{prec}\,z\,f\,\mathsf{zero} \simeq z} \qquad \frac{}{\mathsf{cprecf} \;:\; \mathsf{prec}\,z\,f\,(\mathsf{suc}\,n) \simeq f\,n\,(\mathsf{prec}\,z\,f\,n)}$$

Normal forms are extended accordingly:

$$\frac{}{\mathsf{nzero} \;:\; \mathsf{Nf}\,\mathsf{N}} \qquad \frac{}{\mathsf{nsuc} \;:\; \mathsf{Nf}\,(\mathsf{N}{\to}\mathsf{N})} \qquad \frac{x \;:\; \mathsf{Nf}\,\mathsf{N}}{\mathsf{nsuc}^1 x \;:\; \mathsf{Nf}\,\mathsf{N}}$$

$$\frac{}{\mathsf{nprec} \;:\; \mathsf{Nf}\,(\sigma{\to}(\mathsf{N}{\to}\sigma{\to}\sigma){\to}\mathsf{N}{\to}\sigma)} \qquad \frac{z \;:\; \mathsf{Nf}\,\sigma}{\mathsf{nprec}^1\,z \;:\; \mathsf{Nf}, ((\mathsf{N}{\to}\sigma{\to}\sigma){\to}\mathsf{N}{\to}\sigma)}$$

$$\frac{z \;:\; \mathsf{Nf}\,\sigma \qquad f \;:\; \mathsf{Nf}\,(\mathsf{N}{\to}\sigma{\to}\sigma)}{\mathsf{nprec}^2 z\,f \;:\; \mathsf{Nf}\,(\mathsf{N}{\to}\sigma)}$$

The normaliser is extended as follows:

$$\frac{t \;:\; \mathsf{Tm}\,\sigma}{\mathbf{nf}\;t \;:\; \mathsf{Nf}\,\sigma}$$

$$\begin{aligned}
&\ldots\\
&\mathbf{nf}\quad \mathsf{zero} \;=\; \mathsf{nzero}\\
&\mathbf{nf}\quad \mathsf{suc} \;=\; \mathsf{nsuc}\\
&\mathbf{nf}\quad \mathsf{prec} \;=\; \mathsf{nprec}
\end{aligned}$$

$$\frac{f \;:\; \mathsf{Nf}\,(\sigma{\to}\tau)\quad a \;:\; \mathsf{Nf}\,\sigma}{f \mathbin{@} a \;:\; \mathsf{Nf}\,\tau}$$

$$\begin{aligned}
&\ldots\\
&\mathsf{nsuc} \mathbin{@} x \;=\; \mathsf{nsuc}^1\,x\\
&\mathsf{nprec} \mathbin{@} z \;=\; \mathsf{nprec}^1\,z\\
&(\mathsf{nprec}^1\,z) \mathbin{@} f \;=\; \mathsf{nprec}^2\,z\,f\\
&(\mathsf{nprec}^2\,z\,f) \mathbin{@} \mathsf{nzero} \;=\; z\\
&(\mathsf{nprec}^2\,z\,f) \mathbin{@} (\mathsf{nsuc}^1\,n) \;=\; (f \mathbin{@} n) \mathbin{@} (\mathsf{nprec}^2\,z\,f \mathbin{@} n)
\end{aligned}$$

For strong computability of normal forms we add a clause for base type with one for natural numbers

$$\frac{t \;:\; \mathsf{Nf}\,\sigma}{\mathbf{SCN}_\sigma\,t \;:\; \mathsf{Prop}}$$

$$\begin{aligned}
&\mathbf{SCN}_{\mathsf{N}}\,t \;=\; \mathbf{True}\\
&\ldots
\end{aligned}$$

We need an extra lemma. It is essentially a version of primitive recursion for strongly computable normal forms. Notice that we don't need that n is strongly computable as this contains no useful information.

Lemma 3.

$$\frac{z \;:\; \mathsf{Nf}\,\sigma\quad p \;:\; \mathbf{SCN}\,z\quad f \;:\; \mathsf{Nf}\,(\mathsf{N}{\to}\sigma{\to}\sigma)\qquad p' \;:\; \mathbf{SCN}\,f\quad n \;:\; \mathsf{Nf}\,\mathsf{N}}{\mathbf{sprec}\;p\;p' \;:\; \exists\, x{:}\mathsf{Nf}\,\rho.\mathsf{nprec}^2\,z\,f \mathbin{@} n \Downarrow x \wedge \mathbf{SCN}_\sigma x \wedge \mathsf{prec}\ulcorner z\urcorner\ulcorner f\urcorner\ulcorner n\urcorner \simeq \ulcorner x\urcorner}$$

Proof. By induction on n. □

The rest of the construction is easily extended. The three extensions we have show can be combined to produce a combinatory calculus with finite products and coproducts, and natural numbers. The formalisation of the basic systems, the extensions and their combination is online.

3.9 Chapter summary

We have defined a terminating, sound, and complete normaliser for combinatory logic extended with finite products, coproducts and finally extended to system T. In the next chapter we define a normaliser for a system based on simply typed λ-calculus.

Chapter 4

Simply typed λ-calculi

We present simply typed λ-calculus with explicit substitutions, much like Abadi, Cardelli, Curien and Lèvy's λ^σ-calculus [1]. Substitution becomes an explicit constructor in the syntax and the laws that govern its behaviour become part of the equational theory. This approach avoids the special status of substitution which is often defined by recursion over the syntax. In the next chapter this will allow us to give a first order well typed syntax for a dependently typed language. In this chapter we will follow the same course set out in the previous chapter, this time, for a more sophisticated language. We consider first a core system (simply typed λ-calculus), carry out our construction in full, and then consider extensions to the system.

We are now obliged to give a precise treatment of variables. We choose to use de Bruijn indices and explicit substitutions. One reason to do this is that it scales to dependent types and corresponds closely to models of type theory based on Dybjer's "categories with families" [37] which are based on an underlying notion of a category of substitutions. Other than this, using a nameless, as opposed to named, syntax is mainly a matter of taste. The advantage of avoiding weakening associated with de Bruijn levels is also lost here as we carry around contexts and they must be weakened.

Relationship to published paper This chapter is based on the paper "Big-Step Normalisation" [10], written jointly with Thorsten Altenkirch. Since writing the paper I have simplified and improved the formalisation by using 'order preserving embeddings' to implement weakening and to also to define strong computability without referring to context extension. I have also ex-

tended the system to include finite products. The paper included a second notion of equality: weak β-equality. I have omitted this as it is not required for the construction and was presented only as an observation.

4.1 Syntax

As in the previous chapter we diverge from the conventional strategy of defining pre-terms first and then introducing a typing judgment. Instead, we directly present the well typed terms. We are, after all, only interested in the well typed terms.

The inductive definition of the set of types $\mathsf{Ty} : \star$ with one base type is unchanged from the previous chapter and has the following constructor forms:

$$\frac{}{\bullet \ : \ \mathsf{Ty}} \qquad \frac{\sigma \ : \ \mathsf{Ty} \quad \tau \ : \ \mathsf{Ty}}{(\sigma \to \tau) \ : \ \mathsf{Ty}}$$

Contexts $\mathsf{Con} : \star$ (which are nameless) are represented as left-to-right sequences of types. Contexts are either empty or extended on the right:

$$\frac{}{\varepsilon \ : \ \mathsf{Con}} \qquad \frac{\Gamma \ : \ \mathsf{Con} \quad \sigma \ : \ \mathsf{Ty}}{(\Gamma ; \sigma) \ : \ \mathsf{Con}}$$

Next we define inductive families of well typed terms and substitutions mutually. We write the contexts and types in the same order as they would appear in judgments: $\Gamma \vdash t \ : \ \sigma$ and $\Gamma \vdash \vec{t} \ : \ \Delta$. For substitutions Γ is the context which the constituent terms are in and Δ is the sequence of their types. We give their type constructors first and then explain how to construct elements of each type below.

$$\frac{\Gamma \ : \ \mathsf{Con} \quad \sigma \ : \ \mathsf{Ty}}{\mathsf{Tm}\, \Gamma\, \sigma \ : \ \star} \qquad \frac{\Gamma, \ \Delta \ : \ \mathsf{Con}}{\mathsf{Sub}\, \Gamma\, \Delta \ : \ \star}$$

The syntax of terms uses categorical combinators which subsume variables. There is a term $\varnothing$ which refers to the rightmost variable in the context and $t[\vec{t}]$ is the application of an explicit substitution to a term. Other variables are constructed by combining $\varnothing$ with weakening substitutions $\uparrow^{\sigma}$. E.g. $\varnothing[\uparrow^{\sigma}]$ is the second from the right in the context.

$$\frac{}{\varnothing \ : \ \mathsf{Tm}\, (\Gamma ; \sigma)\, \sigma} \qquad \frac{t \ : \ \mathsf{Tm}\, \Delta\, \sigma \quad \vec{t} \ : \ \mathsf{Sub}\, \Gamma\, \Delta}{t[\vec{t}] \ : \ \mathsf{Tm}\, \Gamma\, \sigma}$$

$$\frac{t \ : \ \mathsf{Tm}\, (\Gamma ; \sigma)\, \tau}{\lambda^{\sigma} t \ : \ \mathsf{Tm}\, \Gamma\, (\sigma \to \tau)} \qquad \frac{t \ : \ \mathsf{Tm}\, \Gamma\, (\sigma \to \tau) \quad u \ : \ \mathsf{Tm}\, \Gamma\, \sigma}{t\, u \ : \ \mathsf{Tm}\, \Gamma\, \tau}$$

Our syntax for substitutions uses the standard categorical combinators: id_Γ the identity substitution, $(\vec{t} \circ \vec{u})$ composition where $t[\vec{t} \circ \vec{u}] \simeq t[\vec{t}][\vec{u}]$, $(\vec{t}; t)$ extension and $\uparrow^\sigma$ weakening. We choose this representation of substitutions over simple sequences of terms as it fits more closely with the categories with families inspired syntax we use for dependent types in the next chapter and it is technically cleaner.

$$\frac{}{\mathsf{id}_\Gamma \ : \ \mathsf{Sub}\,\Gamma\,\Gamma} \qquad \frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta \quad \vec{u} \ : \ \mathsf{Sub}\,B\,\Gamma}{\vec{t} \circ \vec{u} \ : \ \mathsf{Sub}\,B\,\Delta}$$

$$\frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta \quad t \ : \ \mathsf{Tm}\,\Gamma\,\sigma}{(\vec{t}; t) \ : \ \mathsf{Sub}\,\Gamma\,(\Delta; \sigma)} \qquad \frac{}{\uparrow^\sigma \ : \ \mathsf{Sub}\,(\Gamma; \sigma)\,\Gamma}$$

As a special case, we can derive substitution of the last variable by a term: given $t \ : \ \mathsf{Tm}\,(\Gamma; \sigma)\,\tau$ and $u \ : \ \mathsf{Tm}\,\Gamma\,\sigma$, we obtain t with $\varnothing$ substituted by u as $t[u] = t[\mathsf{id}_\Gamma; u] \ : \ \mathsf{Tm}\,\Gamma\,\tau$.

As a more substantial example of a term we represent the λ-term implementing the S combinator. Given $\sigma, \tau, \rho \ : \ \mathsf{Ty}$

$$\vdash \lambda f.\lambda g.\lambda x.f\ x\ (g\ x) \ : \ (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \tau$$

is represented as

$$\lambda(\lambda(\lambda((\varnothing[\uparrow^{\sigma \to \tau}][\uparrow^\sigma])\ \varnothing\ ((\varnothing[\uparrow^\sigma])\ \varnothing)))) \ : \ \mathsf{Tm}\,\varepsilon\,((\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \tau)$$

Equational Theory

We define $\beta\eta$-equality (or conversion) $\simeq$ mutually for terms and substitutions:

$$\frac{t, u \ : \ \mathsf{Tm}\,\Gamma\,\sigma}{t \simeq u \ : \ \mathsf{Prop}} \qquad \frac{\vec{t}, \vec{u} \ : \ \mathsf{Sub}\,\Gamma\,\Sigma}{\vec{t} \simeq \vec{u} \ : \ \mathsf{Prop}}$$

Conversion for terms

First we show the rules for how terms interact with substitutions.

$$\begin{array}{lcll}
\varnothing[\vec{t}; t] & \simeq & t & \mathsf{proj} \\
t[\mathsf{id}_\Gamma] & \simeq & t & \mathsf{id} \\
t[\vec{t} \circ \vec{u}] & \simeq & t[\vec{t}][\vec{u}] & \mathsf{comp} \\
(\lambda^\sigma t)[\vec{t}] & \simeq & \lambda^\sigma(t[\vec{t} \circ \uparrow^\sigma; \varnothing]) & \mathsf{lam} \\
(t\ u)[\vec{t}] & \simeq & t[\vec{t}]\,(u[\vec{t}]) & \mathsf{capp}
\end{array}$$

Next we have β and η rules for functions:

$$\begin{array}{lcll} (\lambda^\sigma t)\, u & \simeq & t[u] & \beta \\ f & \simeq & \lambda^\sigma((f[\uparrow^\sigma])\, \varnothing) & \eta \end{array}$$

In addition we have refl, sym and trans and all congruence rules and ξ, the congruence rule for λ-terms.

$$\frac{t, u \;:\; \mathsf{Tm}\,(\Gamma;\sigma)\,\tau \qquad t \simeq u}{\lambda^\sigma t \simeq \lambda^\sigma u}\;\xi$$

Conversion for substitutions

The conversion for substitutions is given by the usual laws defining a category

$$\begin{array}{lcll} (\vec{t} \circ \vec{u}) \circ \vec{v} & \simeq & \vec{t} \circ (\vec{u} \circ \vec{v}) & \mathsf{assoc} \\ \mathsf{id}_\Gamma \circ \vec{u} & \simeq & \vec{u} & \mathsf{idl} \\ \vec{u} \circ \mathsf{id}_\Gamma & \simeq & \vec{u} & \mathsf{idr} \end{array}$$

and the following laws which formalise the existence of finite products:

$$\begin{array}{lcll} \uparrow^\sigma \circ (\vec{u}; u) & \simeq & \vec{u} & \mathsf{wk} \\ (\vec{t}; t) \circ \vec{u} & \simeq & (\vec{t} \circ \vec{u}); t[\vec{u}] & \mathsf{cons} \\ \mathsf{id}_{\Gamma;\sigma} & \simeq & (\mathsf{id}_\Gamma \circ \uparrow^\sigma); \varnothing & \mathsf{sid} \end{array}$$

The choice of laws is motivated by the need to show soundness and completeness of our normalisation algorithm.

4.2 Normal forms

Having defined the syntax of terms we move onto the syntax of $\beta\eta$-normal forms. $\beta\eta$-normal forms are like ordinary terms but we enforce the invariants that there are no β-redexes present and all possible η-expansions have been performed. This representation of $\beta\eta$-normal form is called β-normal η-long form. Performing the η-expansions puts terms that are not of base type in constructor form. In the present version with only function types this means all terms of function type are λ-abstractions.

Normal forms, like terms are indexed by context and type. At non-base type (only function types currently) normal forms must be in constructor form. We only allow neutral terms (which we define next) at base type.

Alternatively, β normal forms can be defined by allowing neutral terms of any type.

$$\underline{\text{data}}\quad \frac{\Gamma\ :\ \mathsf{Con}\quad \sigma\ :\ \mathsf{Ty}}{\mathsf{Nf}\ \Gamma\ \sigma\ :\ \star}\quad \underline{\text{where}}\quad \frac{n\ :\ \mathsf{Nf}\ (\Gamma;\sigma)\ \tau}{\lambda^{\sigma}\, n\ :\ \mathsf{Nf}\ \Gamma\ (\sigma{\to}\tau)}\qquad \frac{n\ :\ \mathsf{Ne}^{\mathsf{Nf}}\ \Gamma\ \bullet}{\underline{n}\ :\ \mathsf{Nf}\ \Gamma\ \bullet}$$

Neutral terms are terms whose computation is stuck due to the presence of variables in key positions. Since we need neutral terms again later we abstract over the type in the argument position (T) of an application. Neutral terms are either a variable (which we define next) or a neutral term applied to an argument (a stuck application).

$$\underline{\text{data}}\quad \frac{T\ :\ \mathsf{Con}\to\mathsf{Ty}\to\star\quad \Gamma\ :\ \mathsf{Con}\quad \sigma\ :\ \mathsf{Ty}}{\mathsf{Ne}^{T}\ \Gamma\ \sigma\ :\ \star}\quad \underline{\text{where}}$$

$$\frac{x\ :\ \mathsf{Var}\Gamma\sigma}{\hat{x}\ :\ \mathsf{Ne}^{T}\ \Gamma\ \sigma}\qquad \frac{f\ :\ \mathsf{Ne}^{T}\ \Gamma\ (\sigma{\to}\tau)\quad a\ :\ T\ \Gamma\ \sigma}{f\ a\ :\ \mathsf{Ne}^{T}\ \Gamma\ \tau}$$

Lastly we define variables (de Bruijn indices) in the style of Altenkirch and Reus [14]. The variable $\varnothing$ refers to the variable at the (right-hand) end of the context. $\varnothing^{+}$ is the next one in etc.

$$\frac{\Gamma\ :\ \mathsf{Con}\quad \sigma\ :\ \mathsf{Ty}}{\mathsf{Var}\Gamma\sigma\ :\ \star}\quad \underline{\text{where}}\quad \frac{}{\varnothing\ :\ \mathsf{Var}(\Gamma;\sigma)\sigma}\qquad \frac{x\ :\ \mathsf{Var}\Gamma\sigma}{x^{+\tau}\ :\ \mathsf{Var}(\Gamma;\tau)\sigma}$$

We define embeddings back into syntax for normal forms, neutral terms and variables. The embedding operations are not needed for the algorithm itself but are needed for the normalisation proof. We define them now as it helps to show how normal forms and terms are related.

$$\frac{x\ :\ \mathsf{Var}\ \Gamma\ \sigma}{\ulcorner x\urcorner\ :\ \mathsf{Tm}\ \Gamma\ \sigma}\qquad \begin{aligned}\ulcorner\varnothing\urcorner &= \varnothing\\ \ulcorner x^{+\sigma}\urcorner &= \ulcorner x\urcorner[\uparrow^{\sigma}]\end{aligned}$$

$$\frac{n\ :\ \mathsf{Nf}\ \Gamma\ \sigma}{\ulcorner n\urcorner\ :\ \mathsf{Tm}\ \Gamma\ \sigma}\qquad \begin{aligned}\ulcorner\lambda^{\sigma}\, n\urcorner &= \lambda^{\sigma}\ulcorner n\urcorner\\ \ulcorner\underline{n}\urcorner &= \ulcorner n\urcorner\end{aligned}$$

$$\frac{n\ :\ \mathsf{Ne}^{\mathsf{Nf}}\ \Gamma\ \sigma}{\ulcorner n\urcorner\ :\ \mathsf{Tm}\ \Gamma\ \sigma}\qquad \begin{aligned}\ulcorner\hat{x}\urcorner &= \ulcorner x\urcorner\\ \ulcorner n\ n'\urcorner &= \ulcorner n\urcorner\ulcorner n'\urcorner\end{aligned}$$

4.3 Recursive normalisation

Our approach is the same as the previous chapter. We will define a recursive normaliser and later prove it correct.

Specification

Our goal is, again, to define a normalisation function which takes a term and produces its normal form

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathbf{nf}\;t \;:\; \mathsf{Nf}\,\Gamma\,\sigma}$$

such that it satisfies the following properties:

soundness Normalisation takes convertible terms to identical normal forms

$$\frac{t \simeq t'}{\mathbf{nf}\;t = \mathbf{nf}\;t'}$$

completeness Terms are convertible to their normal forms

$$t \simeq \ulcorner \mathbf{nf}\;t \urcorner$$

As a consequence we obtain that convertibility corresponds to having the same normal form:

$$t \simeq u \iff \mathbf{nf}\;t = \mathbf{nf}\;u$$

We start with a recursive implementation of normalisation and will later verify that it is terminating, sound and complete.

Top level structure

We define the normalisation function (**nf**) as follows:

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathbf{nf}\;t \;:\; \mathsf{Nf}\,\Gamma\,\sigma} \quad \mathbf{nf}\;t \;=\; \mathbf{quote}\,(\mathbf{eval}\;t\,\mathbf{id}_\Gamma)$$

We first sketch the top level structure of the algorithm before going into the details of the implementation. Normalisation proceeds in two steps: we define a simple evaluator, basically an environment machine, which produces intermediate values (weak head normal forms):

$$\frac{t \;:\; \mathsf{Tm}\,\Delta\,\sigma \quad \vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\mathbf{eval}\;t\;\vec{v} \;:\; \mathsf{Val}\,\Gamma\,\sigma}$$

The evaluator takes an environment (the identity environment $\mathbf{id}_\Gamma$ when invoked directly by **nf**) which assigns to every free variable a value of the appropriate type and returns a value. To complete the normalisation function

we define a type directed quotation operation which returns a normal form by recursively evaluating the value:

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathbf{quote}\,v \;:\; \mathsf{Nf}\,\Gamma\,\sigma}$$

Now, a value is either a λ-closure or a neutral value (embedded at any type so we allow neutral functions). We also define environments mutually:

$$\frac{\Gamma \;:\; \mathsf{Con} \quad \sigma \;:\; \mathsf{Ty}}{\mathsf{Val}\,\Gamma\,\sigma \;:\; \star} \qquad \frac{\Gamma, \Delta \;:\; \mathsf{Con}}{\mathsf{Env}\,\Gamma\,\Delta \;:\; \star} \quad \underline{\text{where}}$$

$$\frac{t \;:\; \mathsf{Tm}\,(\Delta;\sigma)\,\tau \quad \vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\lambda^\sigma t[\vec{v},-] \;:\; \mathsf{Val}\,\Gamma\,(\sigma\to\tau)} \qquad \frac{n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma}{\underline{n} \;:\; \mathsf{Val}\,\Gamma\,\sigma}$$

$$\frac{}{\varepsilon \;:\; \mathsf{Env}\,\Gamma\,\varepsilon} \qquad \frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma \quad \vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{(\vec{v};v) \;:\; \mathsf{Env}\,\Gamma\,(\Delta;\sigma)}$$

Having defined values and environments mutually we define their embeddings back into syntax:

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\ulcorner v \urcorner \;:\; \mathsf{Tm}\,\Gamma\,\sigma} \qquad \begin{aligned} \ulcorner \lambda^\sigma t[\vec{v},-] \urcorner &= (\lambda^\sigma t)[\ulcorner \vec{v} \urcorner] \\ \ulcorner \underline{n} \urcorner &= \ulcorner n \urcorner \end{aligned}$$

$$\frac{n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma}{\ulcorner n \urcorner \;:\; \mathsf{Tm}\,\Gamma\,\sigma} \qquad \begin{aligned} \ulcorner \hat{x} \urcorner &= \ulcorner x \urcorner \\ \ulcorner n\; n' \urcorner &= \ulcorner n \urcorner \ulcorner n' \urcorner \end{aligned}$$

$$\frac{\vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\ulcorner \vec{v} \urcorner \;:\; \mathsf{Sub}\,\Gamma\,\Delta} \qquad \begin{aligned} \ulcorner \vec{v}; v \urcorner &= \ulcorner \vec{v} \urcorner ; \ulcorner v \urcorner \\ \ulcorner \varepsilon \urcorner^{\varepsilon} &= \mathsf{id} \\ \ulcorner \varepsilon \urcorner^{(\Gamma;\sigma)} &= \ulcorner \varepsilon \urcorner^{\Gamma} \circ \uparrow^{\sigma} \end{aligned}$$

Evaluation

We are ready to define evaluation mutually with evaluation of substitutions and applications of values. Note that these are not yet structurally recursive functions expressed in type theory. This is the näive implementation that we will show to be terminating in the sections that follow.

$$\frac{t \;:\; \mathsf{Tm}\Delta\sigma \quad \vec{v} \;:\; \mathsf{Env}\Gamma\Delta}{\mathbf{eval}\,t\,\vec{v} \;:\; \mathsf{Val}\Gamma\sigma} \qquad \frac{\vec{t} \;:\; \mathsf{Sub}\,\Delta\,\Sigma \quad \vec{v} \;:\; \mathsf{Env}\Gamma\Delta}{\mathbf{eval}\,\vec{t}\,\vec{v} \;:\; \mathsf{Env}\Gamma\Sigma}$$

$$\frac{f \;:\; \mathsf{Val}\Gamma(\sigma\to\tau) \quad a \;:\; \mathsf{Val}\Gamma\sigma}{f \mathbin{@} a \;:\; \mathsf{Val}\Gamma\tau}$$

We define **eval**, an environment based evaluator for terms

$$\begin{array}{llll} \mathbf{eval} & \varnothing & (\vec{v}; v) & = & v \\ \mathbf{eval} & t[\vec{t}] & \vec{v} & = & \mathbf{eval}\, t\, (\vec{\mathbf{eval}}\, \vec{t}\, \vec{v}) \\ \mathbf{eval} & \lambda t & \vec{v} & = & \lambda t[\vec{v}, -] \\ \mathbf{eval} & t\, u & \vec{v} & = & (\mathbf{eval}\, t\, \vec{v}) \mathbin{@} (\mathbf{eval}\, u\, \vec{v}) \end{array}$$

and substitutions

$$\begin{array}{llll} \vec{\mathbf{eval}} & \mathsf{id} & \vec{v} & = & \vec{v} \\ \vec{\mathbf{eval}} & \vec{t} \circ \vec{u} & \vec{v} & = & \vec{\mathbf{eval}}\, \vec{t}\, (\vec{\mathbf{eval}}\, \vec{u}\, \vec{v}) \\ \vec{\mathbf{eval}} & (\vec{t}; t) & \vec{v} & = & (\vec{\mathbf{eval}}\, \vec{t}\, \vec{v}); (\mathbf{eval}\, t\, \vec{v}) \\ \vec{\mathbf{eval}} & \uparrow^{\sigma} & (\vec{v}; v) & = & \vec{v} \end{array}$$

Application for λ-values recursively calls **eval** on the term with the environment extended by the argument. This is important, it is precisely here that we see the suspended computation that was held in the closure being performed now that we have the argument we were waiting for.

In the case of a neutral value $\underline{n}$, the value application is replaced by a neutral application. the spine:

$$\begin{array}{lll} \lambda t[\vec{v}, -] & @ \quad a & = \quad \mathbf{eval}\, t\, (\vec{v}; a) \\ \underline{n} & @ \quad a & = \quad \underline{n\, a} \end{array}$$

Notice that this is not yet a properly explained definition. **eval** and @ are mutually defined. If we were to inline @ we would see that we have a nested recursive call: in the case of an application we evaluate the components and then may (in the case of a non-neutral function) call **eval** again on the result of evaluating the function.

Introducing fresh variables

To define quotation (and the identity environment for that matter) we must be able to introduce fresh variables. We do this by defining a weakening operation which introduces a new variable at the end of the context. For values this is very straightforward. Weakening can be defined by mutual recursion for neutral values, values, and environments. Such a definition is all we need for the algorithm, but for the proof we need to weaken normal forms. To weaken normal forms we must go under binders. Under a binder, the new variable is no longer at the end of the context so we must generalise

our notion of weakening. In the case of values, and in particular λ-closures, we do not have to go under the binder, we can just weaken the closure by weakening its environment. We declare ordinary weakening for values now omitting its definition. We will define it later as a special case of the more general operation of *order preserving embedding*.

$$\frac{v \;:\; \mathsf{Val}\;\Gamma\;\sigma}{v^{+\tau} \;:\; \mathsf{Val}\;(\Gamma;\tau)\;\sigma} \qquad v^{+\tau} \;=\; \dots$$

Given a suitable notion weakening for environments we can define the identity environment, which is used by **nf**, by recursion over Γ:

$$\frac{\Gamma \;:\; \mathsf{Con}}{\mathbf{id}_\Gamma \;:\; \mathsf{Env}\;\Gamma\;\Gamma}$$

$$\begin{aligned}\mathbf{id}_\varepsilon &= \varepsilon\\ \mathbf{id}_{(\Gamma;\sigma)} &= (\mathbf{id}_\Gamma)^{+\sigma};\underline{\hat{\varnothing}}\end{aligned}$$

Quotation

We are ready to define **quote** for values simultaneously with $\overline{\mathbf{quote}}$ for neutral values:

$$\frac{v \;:\; \mathsf{Val}\;\Gamma\;\sigma}{\mathbf{quote}_\sigma\;v \;:\; \mathsf{Nf}\;\Gamma\;\sigma} \qquad \frac{n \;:\; \mathsf{Ne}^{\mathsf{Val}}\;\Gamma\;\sigma}{\overline{\mathbf{quote}}\;n \;:\; \mathsf{Ne}^{\mathsf{Nf}}\;\Gamma\;\sigma}$$

$$\begin{aligned}\mathbf{quote}_\bullet\quad & \underline{n} &&= \underline{\overline{\mathbf{quote}}\;n}\\ \mathbf{quote}_{(\sigma\to\tau)}\quad & f &&= \lambda^\sigma\mathbf{quote}_\tau\;(f^{+\sigma}\;@\;\underline{\hat{\varnothing}})\\ \overline{\mathbf{quote}}\quad & \hat{x} &&= \hat{x}\\ \overline{\mathbf{quote}}\quad & n\;v &&= (\overline{\mathbf{quote}}\;n)\;(\mathbf{quote}\;v)\end{aligned}$$

Note that we define **quote** by recursion over the type, η-expanding functions that might be neutral. This can be avoided, if we are only interested in β-normal forms. In this case we would define **quote** as follows:

$$\begin{aligned}\mathbf{quote}^\beta\quad & \lambda^\sigma\,t[\vec{v},-] &&= \lambda^\sigma\mathbf{quote}^\beta_\tau\;(\mathbf{eval}\;t\;(\vec{v}^{+\sigma};\underline{\hat{\varnothing}}))\\ \mathbf{quote}^\beta\quad & \underline{n} &&= \underline{\overline{\mathbf{quote}}\;n}\end{aligned}$$

Quote for neutrals would remain the same.

4.4 Big-step semantics

We will now apply the Bove-Capretta technique to the recursive definition of normalisation from the previous section. The big-step semantics is given by

the following mutually inductive defined relations:

$$\frac{t \,:\, \mathsf{Tm}\,\Delta\,\sigma \quad \vec{v} \,:\, \mathsf{Env}\,\Gamma\,\Delta \quad v \,:\, \mathsf{Val}\,\Gamma\,\sigma}{\mathsf{eval}\;t\;\vec{v} \Downarrow v \;:\; \mathsf{Prop}}$$

$$\frac{\vec{t} \,:\, \mathsf{Sub}\,\Delta\,\Sigma \quad \vec{v} \,:\, \mathsf{Env}\,\Gamma\,\Delta \quad \vec{w} \,:\, \mathsf{Env}\,\Gamma\,\Sigma}{\vec{\mathsf{eval}}\;\vec{t}\;\vec{v} \Downarrow \vec{w} \;:\; \mathsf{Prop}}$$

$$\frac{f \,:\, \mathsf{Val}\,\Gamma\,(\sigma \to \tau) \quad a \,:\, \mathsf{Val}\,\Gamma\,\sigma \quad v \,:\, \mathsf{Val}\,\Gamma\,\tau}{f \mathbin{@} a \Downarrow v \;:\; \mathsf{Prop}}$$

$$\frac{v \,:\, \mathsf{Val}\,\Gamma\,\sigma \quad n \,:\, \mathsf{Nf}\,\Gamma\,\sigma}{\mathsf{quote}\;v \Downarrow n \;:\; \mathsf{Prop}} \qquad \frac{v \,:\, \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma \quad n \,:\, \mathsf{Ne}^{\mathsf{Nf}}\,\Gamma\,\sigma}{\overline{\mathsf{quote}}\;v \Downarrow n \;:\; \mathsf{Prop}}$$

$$\frac{t \,:\, \mathsf{Tm}\,\Gamma\,\sigma \quad n \,:\, \mathsf{Nf}\,\Gamma\,\sigma}{\mathsf{nf}\;t \Downarrow n \;:\; \mathsf{Prop}}$$

The inductive definition of these relations is straightforward from the recursive definition of the functions in the previous section because they are first order. To illustrate this we give the constructors for $\mathsf{eval}\;t\;\vec{v} \Downarrow v$:

$$\overline{\mathsf{rlam} \;:\; \mathsf{eval}\;(\lambda^{\sigma}\,t)\;\vec{v} \Downarrow \lambda^{\sigma}\,t[\vec{v}, -]} \qquad \overline{\mathsf{rvar} \;:\; \mathsf{eval}\;\varnothing\;(\vec{v}; v) \Downarrow v}$$

$$\frac{p \,:\, \vec{\mathsf{eval}}\;\vec{t}\;\vec{v} \Downarrow \vec{v}' \quad q \,:\, \mathsf{eval}\;t\;\vec{v}' \Downarrow v}{\mathsf{rsubs}\;p\;q \;:\; \mathsf{eval}\;(t[\vec{t}])\;\vec{v} \Downarrow v}$$

$$\frac{p \,:\, \mathsf{eval}\;t\;\vec{v} \Downarrow f \quad q \,:\, \mathsf{eval}\;u\;\vec{v} \Downarrow v \quad r \,:\, f \mathbin{@} v \Downarrow w}{\mathsf{rapp}\;p\;q\;r \;:\; \mathsf{eval}\;(t\;u)\;\vec{v} \Downarrow w}$$

We can now augment our evaluation algorithm, making it structurally recursive on the big-step relation.

$$\frac{t \,:\, \mathsf{Tm}\,\Delta\,\sigma \quad \vec{v} \,:\, \mathsf{Env}\,\Gamma\,\Delta \quad p \,:\, \mathsf{eval}\;t\;\vec{v} \Downarrow v}{\mathbf{eval}^{\mathbf{str}}\;t\;\vec{v}\;p \;:\; \Sigma v' \!:\! \mathsf{Val}\,\Gamma\,\sigma\,.\,v' = v}$$

$$\frac{\vec{t} \,:\, \mathsf{Sub}\,\Delta\,\Sigma \quad \vec{v} \,:\, \mathsf{Env}\,\Gamma\,\Delta \quad p \,:\, \vec{\mathsf{eval}}\;\vec{t}\;\vec{v} \Downarrow \vec{w}}{\vec{\mathbf{eval}}^{\mathbf{str}}\;\vec{t}\;\vec{v}\;p \;:\; \Sigma w' \!:\! \mathsf{Env}\,\Gamma\,\Sigma\,.\,w' = w}$$

$$\frac{f \,:\, \mathsf{Val}\,\Gamma\,(\sigma \to \tau) \quad a \,:\, \mathsf{Val}\,\Gamma\,\sigma \quad p \,:\, f \mathbin{@} a \Downarrow v}{\mathbf{app}^{\mathbf{str}}\;f\;a\;p \;:\; \Sigma v' \!:\! \mathsf{Val}\,\Gamma\,\tau\,.\,v' = v}$$

$$\frac{v \,:\, \mathsf{Val}\,\Gamma\,\sigma \quad p \,:\, \mathsf{quote}\;v \Downarrow n}{\mathbf{quote}^{\mathbf{str}}\;v\;p \;:\; \Sigma n' \!:\! \mathsf{Nf}\,\Gamma\,\sigma\,.\,n' = n} \qquad \frac{m \,:\, \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma \quad p \,:\, \overline{\mathsf{quote}}\;m \Downarrow n}{\overline{\mathbf{quote}}^{\mathbf{str}}\;m\;p \;:\; \Sigma n' \!:\! \mathsf{Ne}^{\mathsf{Nf}}\,\Gamma\,\sigma\,.\,n' = n}$$

$$\frac{p \;:\; \mathsf{nf}\ t \Downarrow n}{\mathbf{nf}^{\mathbf{str}}\ t\ p \;:\; \mathsf{Nf}\,\Gamma\sigma}$$

The definition of the structural recursive operator proceeds along the lines of our example $\mathbf{f}^{\mathbf{str}}$, e.g. in the case of $\mathbf{eval}^{\mathbf{str}}$ this becomes:

$$\begin{array}{llllll}
\mathbf{eval}^{\mathbf{str}} & \varnothing & (\vec{v}; v) & \mathsf{rvar} & = & (v, \mathsf{refl}) \\
\mathbf{eval}^{\mathbf{str}} & t[\vec{t}] & \vec{v} & (\mathsf{rsubs}\ p\ q) & \underline{\text{with}} & \vec{\mathbf{eval}}^{\mathbf{str}}\ \vec{t}\ \vec{v}\ p \\
 & & & & \mid & (\vec{v}', \mathsf{refl}) \quad = \quad \mathbf{eval}^{\mathbf{str}}\ t\ \vec{v}'\ q \\
\mathbf{eval}^{\mathbf{str}} & \lambda t & \vec{v} & \mathsf{rlam} & = & (\lambda t[\vec{v}, -], \mathsf{refl}) \\
\mathbf{eval}^{\mathbf{str}} & t\ u & \vec{v} & (\mathsf{rapp}\ p\ q\ r) & \underline{\text{with}} & \mathbf{eval}^{\mathbf{str}}\ t\ \vec{v}\ p \quad \mid \quad \mathbf{eval}^{\mathbf{str}}\ u\ \vec{v}\ q \\
 & & & & \mid & (f, \mathsf{refl}) \quad \mid \quad (a, \mathsf{refl}) \\
 & = \mathbf{app}^{\mathbf{str}}\ f\ a\ r & & & &
\end{array}$$

4.5 Order preserving embeddings

Order preserving embeddings (OPEs) were first used by Altenkirch, Hofmann and Streicher [12] and also appear as an exercise in McBride's AFP notes [69], although he consider the more general notion of order preserving functions, of which embeddings (injective functions) are a special case. OPEs are a first order presentation of renamings which do not refer to individual variables. Their use here has lead to a simplification of our formalisation. In the original formalisation that accompanied the paper "Big Step Normalisation" we implemented weakening of normal forms using conventional renaming (variable substitutions represented as simple sequences of variables) and defined (Kripke) strong computability by referring to context extension (concatenation of contexts). The most difficult part of the formalisation required intricate reasoning about renaming and context extension. In particular we had to regularly appeal to associativity of context extension which does not hold definitionally. Also, in reasoning about renaming, we often had to reason about functions rather than first order structures.

We now define OPE which is a first order presentation of operations from one context to another which could be considered a renaming. It is, in effect, a first order representation of a function space. The idea is to walk through the context from beginning to end saying what to keep and what to skip (indicating a new variable). McBride's exercise asks for a unique representation

of order preserving functions as the uniqueness simplifies their use. We adopt a unique representation of OPEs by defining the trivial done (defined below) instead of the more general identity OPE as used by Altenkirch, Hofmann and Streicher.

We now define the type OPE:

$$\underline{\text{data}}\quad \frac{\Gamma,\ \Delta\ :\ \mathsf{Con}}{\mathsf{OPE}\ \Gamma\ \Delta\ :\ \star}\quad \underline{\text{where}}\quad \frac{}{\mathsf{done}\ :\ \mathsf{OPE}\ \varepsilon\ \varepsilon}$$

$$\frac{\sigma\ :\ \mathsf{Ty}\quad o\ :\ \mathsf{OPE}\ \Gamma\ \Delta}{\mathsf{keep}\ \sigma\ o\ :\ \mathsf{OPE}\ (\Gamma;\sigma)\ (\Delta;\sigma)}\qquad \frac{\sigma\ :\ \mathsf{Ty}\quad o\ :\ \mathsf{OPE}\ \Gamma\ \Delta}{\mathsf{skip}\ \sigma\ o\ :\ \mathsf{OPE}\ (\Gamma;\sigma)\ \Delta}$$

We illustrate its use by defining the action on variables. We define it by recursion on f.

$$\frac{f\ :\ \mathsf{OPE}\ \Gamma\ \Delta\quad x\ :\ \mathsf{Var}\ \Delta\ \sigma}{f\ {}^*\ x\ :\ \mathsf{Var}\ \Gamma\ \sigma}$$

$$\begin{array}{lcll}
\mathsf{done} & {}^* & () & \\
(\mathsf{keep}\ \sigma\ f) & {}^* & \varnothing & = \varnothing \\
(\mathsf{keep}\ \sigma\ f) & {}^* & (x^{+\sigma}) & = (f\ {}^*\ x)^{+\sigma} \\
(\mathsf{skip}\ \sigma\ f) & {}^* & x & = (f\ {}^*\ x)^{+\sigma}
\end{array}$$

There is no right hand side for done as in the empty contexts there can be no variables. In the case of keep we simply preserve the position of the variable constructors ($\varnothing$ and $-^{+-}$). In the case of skip we add an extra $-^{+-}$ and carry on, the skip is essentially the weakening. It should also be noted that this operation cannot contract the context, it can just insert new types in arbitrary positions and modify variables accordingly.

Before going any further we explain how to embed OPEs into substitutions:

$$\frac{f\ :\ \mathsf{OPE}\ \Gamma\ \Delta}{\ulcorner f\urcorner\ :\ \mathsf{Sub}\ \Gamma\ \Delta}\qquad \begin{array}{lcl}
\ulcorner\mathsf{done}\urcorner & = & \mathbf{id}_{\varepsilon} \\
\ulcorner\mathsf{keep}\ \sigma\ f\urcorner & = & (\ulcorner f\urcorner \circ \uparrow^{\sigma});\varnothing \\
\ulcorner\mathsf{skip}\ \sigma\ f\urcorner & = & \ulcorner f\urcorner \circ \uparrow^{\sigma}
\end{array}$$

Next we define the identity OPE recursively:

$$\frac{\Gamma\ :\ \mathsf{Con}}{\mathbf{id}_{\Gamma}\ :\ \mathsf{OPE}\ \Gamma\ \Gamma}\qquad \begin{array}{lcl}
\mathbf{id}_{\varepsilon} & = & \mathsf{done} \\
\mathbf{id}_{\Gamma;\sigma} & = & \mathsf{keep}\ \sigma\ \mathbf{id}_{\Gamma}
\end{array}$$

Having defined identity we can easily define the special case OPE that corresponds to ordinary weakening:

$$\frac{\Gamma\ :\ \mathsf{Con}\quad \tau\ :\ \mathsf{Ty}}{\mathbf{weak}\ \tau\ :\ \mathsf{OPE}\ (\Gamma;\tau)\ \Gamma}\qquad \mathbf{weak}\ \ \tau\ \ =\ \ \mathsf{skip}\ \tau\ \mathbf{id}$$

Next we define composition of OPEs.

$$\frac{f \;:\; \mathsf{OPE}\; B\; \Gamma \qquad g \;:\; \mathsf{OPE}\; \Gamma\; \Delta}{f \circ g \;:\; \mathsf{OPE}\; B\; \Delta}$$

$$\begin{array}{lcll} \mathsf{done} & \circ & \mathsf{done} & = \quad \mathsf{done} \\ (\mathsf{skip}\; \sigma\; f) & \circ & g & = \quad \mathsf{skip}\; \sigma\; (f \circ g) \\ (\mathsf{keep}\; \sigma\; f) & \circ & (\mathsf{keep}\; \sigma\; g) & = \quad \mathsf{keep}\; \sigma\; (f \circ g) \\ (\mathsf{keep}\; \sigma\; f) & \circ & (\mathsf{skip}\; \sigma\; g) & = \quad \mathsf{skip}\; \sigma\; (f \circ g) \end{array}$$

Having defined identity and composition we can now show that OPEs form a category:

Lemma 4. *OPEs form a category. The objects are contexts and the morphisms are OPEs. I.e. the following properties hold*

$$\begin{array}{lcll} f & \circ\; \mathbf{id} & = \; f & \textit{right identity} \\ \mathbf{id} & \circ\; f & = \; f & \textit{left identity} \\ (f \circ g) & \circ\; h & = \; f \circ (g \circ h) & \textit{associativity} \end{array}$$

Proof. Separately by induction on f in each case. □

We can also show that that identity and composition interact appropriately with the action on variables. We highlight this for variables but the same applies for the action on values, neutral values, environments, normal forms and neutral normal forms.

Lemma 5. *The action of the identity OPE on variables is the identity.*

$$\mathbf{id}^{\,*}\, x = x$$

Proof. By induction on x. □

Lemma 6. *Composition of actions is the same as the action of composition of OPEs.*

$$f^{\,*}\, (g^{\,*}\, x) = (f \circ g)^{\,*}\, x$$

Proof. By induction on f. □

Later we will need the following property about the identity OPE:

Lemma 7. *The embedding of the identity OPE is convertible to the identity substitution.*

$$\ulcorner \mathbf{id}_\Gamma \urcorner \simeq \mathsf{id}_\Gamma$$

Proof. By induction on the context Γ. Using sid rule. □

Next we want to be able to fill in the definitions of ordinary weakening for values and normal forms that we omitted earlier. We define the action of OPEs on values, neutral values, environment, normal forms and neutral terms. After we have done this we can define ordinary weakening as a special case.

We mutually define the action of OPEs (*) for values, neutral values and environments. Notice that in the below definitions the OPE (f) is pushed through the structure of values, neutral values and environments unchanged. This is because we do not have to go under binders for these data structures as opposed to the definitions for normal forms that follow.

$$\frac{f \;:\; \mathsf{OPE}\,\Gamma\,\Delta \qquad v \;:\; \mathsf{Val}\,\Delta\,\sigma}{f \mathbin{^*} v \;:\; \mathsf{Val}\,\Gamma\,\sigma}$$

$$\begin{aligned} f \;\; {}^* \;\; \lambda t[\vec{v}, -] &= \lambda t[f \mathbin{^*} \vec{v}, -] \\ f \;\; {}^* \;\; \underline{n} &= \underline{f \mathbin{^*} n} \end{aligned}$$

$$\frac{f \;:\; \mathsf{OPE}\,\Gamma\,\Delta \qquad n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Delta\,\sigma}{f \mathbin{^*} n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma}$$

$$\begin{aligned} f \;\; {}^* \;\; \hat{x} &= \widehat{f \mathbin{^*} x} \\ f \;\; {}^* \;\; (n\; v) &= (f \mathbin{^*} n)\,(f \mathbin{^*} v) \end{aligned}$$

$$\frac{f \;:\; \mathsf{OPE}\,B\,\Gamma \qquad \vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{f \mathbin{^*} \vec{v} \;:\; \mathsf{Env}\,B\,\Delta}$$

$$\begin{aligned} f \;\; {}^* \;\; \varepsilon &= \varepsilon \\ f \;\; {}^* \;\; (\vec{v}; v) &= (f \mathbin{^*} \vec{v}); (f \mathbin{^*} v) \end{aligned}$$

We now complete our definition of $- \mathbin{^*} -$ by defining it for normal forms and

neutral normal forms:

$$\frac{f \;:\; \mathsf{OPE}\,\Gamma\,\Delta \qquad n \;:\; \mathsf{Nf}\,\Delta\,\sigma}{f \mathbin{*} n \;:\; \mathsf{Nf}\,\Gamma\,\sigma}$$

$$\begin{array}{lcl} f \mathbin{*} (\lambda\, n) & = & \lambda\,((\mathsf{keep}\ \_\ f) \mathbin{*} n) \\ f \mathbin{*} \underline{n} & = & \underline{f \mathbin{*} n} \end{array}$$

$$\frac{f \;:\; \mathsf{OPE}\,\Gamma\,\Delta \qquad n \;:\; \mathsf{Ne}^{\mathsf{Nf}}\,\Delta\,\sigma}{f \mathbin{*} n \;:\; \mathsf{Ne}^{\mathsf{Nf}}\,\Gamma\,\sigma}$$

$$\begin{array}{lcl} f \mathbin{*} \hat{x} & = & \widehat{f \mathbin{*} x} \\ f \mathbin{*} (n\, v) & = & (f \mathbin{*} n)\,(f \mathbin{*} v) \end{array}$$

In the case of $f \mathbin{*} (\lambda\, n)$ above where we have to push f under the binder λ we add a keep. This is the payoff of OPEs: we can easily push them under a binder.

Having defined the action on values we can complete the definition of ordinary weakening for values we omitted earlier:

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{v^{+\tau} \;:\; \mathsf{Val}\,(\Gamma;\tau)\,\sigma} \qquad v^{+\tau} \;=\; (\mathbf{weak}\,\tau) \mathbin{*} v$$

We also need a completeness-like result for OPEs which states that applying an OPE to a variable (for example) and then embedding the result into the syntax is convertible to embedding the variable, embedding the OPE (to produce a substitution) and then applying the resulting substitution to the variable. Here, again, we highlight the property for variables but the same holds for values, normal forms, etc.

Lemma 8. *Completeness property for OPEs.*

$$\frac{f \;:\; \mathsf{OPE}\,\Gamma\,\Delta \qquad x \;:\; \mathsf{Var}\,\Delta\,\sigma}{\ulcorner f \mathbin{*} x \urcorner \simeq \ulcorner x \urcorner [\ulcorner f \urcorner]}$$

Proof. By induction on f. □

Lastly we need to show that we can push OPEs through the big-step semantics and also through the definitions of the recursive functions.

Lemma 9. * *commutes with* $\mathsf{eval} - - \Downarrow -$, $- @ - \Downarrow -$, $\mathsf{quote} - \Downarrow -$ *and* $\overline{\mathsf{quote}} - \Downarrow -$

Proof. By induction on big-step derivations. □

Lemma 10. *The action of an OPE (*) commutes with* **eval**, @, **quote** *and* $\overline{\textbf{quote}}$.

Proof. The proofs follow the structure of the function definitions. E.g. **eval** is defined by recursion on the structure terms so the proof is by induction on the structure of terms. □

The last lemma is required for the soundness property of normalisation. When we prove soundness we will know that the recursive functions are terminating.

4.6 Termination and completeness

In this section we show that our normalisation algorithm terminates and that it satisfies the completeness property. As in the previous chapter we use strong computability to show termination. Values get used in contexts other than the one in which they are computed. That is because a reference to a variable can be under several new binders since its own. We need to be able to shift to more informative contexts so we introduce a Kripke style extension of computability at higher type. The Kripke style extension is defined using OPEs which abstract over the computational details of context extension. This simplifies the formalisation. If we had used context extension directly we would have to reason about associativity of context extension in the formalisation. E.g. The type $\mathsf{Val}\,((B + \Gamma) + \Delta)\,\sigma$ is propositionally equal to the type $\mathsf{Val}\,(B + (\Gamma + \Delta))\,\sigma$.

We define the predicate $\mathbf{SCV}_{\Gamma,\sigma}$ by recursion over the type σ.

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathbf{SCV}_{\Gamma,\sigma}\,v \;:\; \mathsf{Prop}}$$

$$\mathbf{SCV}_{\Gamma,\bullet}\quad \underline{n} \;=\; \exists m : \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\bullet.\,(\overline{\mathsf{quote}}\,n \Downarrow m) \wedge (\ulcorner n \urcorner \simeq \ulcorner m \urcorner)$$

$$\mathbf{SCV}_{\Gamma,(\sigma\to\tau)}\quad f \;=\; \forall o : \mathsf{OPE}\,B\,\Gamma.\,\forall v : \mathsf{Val}\,B\,\sigma.\,\mathbf{SCV}_{B,\sigma}\,v \to$$
$$\exists w : \mathsf{Val}\,B\,\tau.\,(o\,{}^*\,f\,@\,v \Downarrow w) \wedge (\ulcorner o\,{}^*\,f \urcorner\,\ulcorner v \urcorner \simeq \ulcorner w \urcorner) \wedge \mathbf{SCV}_{B,\tau}\,w$$

Notice that, whilst we allow neutrals to be embedded into values at any type, all values of base type must be neutral. Hence we can pattern match on $\underline{n}$ in the definition of **SCV** at base type $\bullet$.

It is straightforward to extend strong computability to environments:

$$\frac{\vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\mathbf{SCE}\,\vec{v} \;:\; \mathsf{Prop}}$$

$$\begin{aligned}\mathbf{SCE}\quad \varepsilon \quad &= \quad \mathbf{True}\\ \mathbf{SCE}\quad (\vec{v}; v) \quad &= \quad \mathbf{SCE}\,\vec{v} \wedge \mathbf{SCV}\,v\end{aligned}$$

We will need that strong computability is closed under OPE for values:

Lemma 11.

$$\frac{f \;:\; \mathsf{OPE}\,B\,\Gamma \quad \mathbf{SCV}_{\Gamma,\sigma}\,v}{\mathbf{SCV}_{B,\sigma}\,(f\,{}^*\,v)}$$

Proof. By induction over σ. In the base case $\bullet$ we need lemma 8 and lemma 9. In the arrow case $\sigma \to \tau$ we need lemma 6. □

We need the corresponding property for environments too:

Lemma 12.

$$\frac{f \;:\; \mathsf{OPE}\,B\,\Gamma \quad \mathbf{SCE}_{\Gamma,\Delta}\,\vec{v}}{\mathbf{SCE}_{B,\Delta}\,(f\,{}^*\,\vec{v})}$$

Proof. By induction over $\vec{v}$. The case for the empty environment is trivial. The non-empty case follows by inductive hypothesis and lemma 11. □

Our main technical lemma is that **quote** terminates for all strongly computable values and that the result is $\beta\eta$-convertible to the input (q). Our proof proceeds by induction over the type. To deal with the negative occurrence of types, we show at the same time that termination of quote for neutral terms implies strong computability (u). At base type strong computability *is* quotability and at at higher type strongly computable is applicability. Neutrals are trivial to apply and hence it is not surprising that (u) should hold.

This structure of establishing two propositions by mutual induction over types is common to conventional strong normalisation proofs and can also be found in the normalisation by evaluation constructions.

Lemma 13.

$$\frac{\mathbf{SCV}_{\Gamma,\sigma}\,v}{\exists m : \mathsf{Nf}\,\Gamma\sigma.\,\mathsf{quote}_{\Gamma,\sigma}\,v{\Downarrow}m \wedge \ulcorner v \urcorner \simeq \ulcorner m \urcorner}(q) \qquad \frac{\overline{\mathsf{quote}}_{\Gamma,\sigma}\,n{\Downarrow}m \quad \ulcorner n \urcorner \simeq \ulcorner m \urcorner}{\mathbf{SCV}_{\Gamma,\sigma}\,n}(u)$$

Proof. By mutual induction over σ. In the base case both implications follow trivially from the definition of **SCV** and the observation that all values of base type are neutral. Consider $(\sigma \to \tau)$:

(q) Given $\mathbf{SCV}_{\Gamma,(\sigma\to\tau)}\, f$ (1). Using inductive hypothesis (u) for σ we can show that $\mathbf{SCV}_{(\Gamma;\sigma),\sigma}\, \underline{\hat{\varnothing}}$, and hence $f^{+\sigma} @ \underline{\hat{\varnothing}} \Downarrow v$ (2), $\ulcorner f^{+\sigma} \urcorner\, \varnothing \simeq \ulcorner v \urcorner$ (3) and $\mathbf{SCV}_{(\Gamma;\sigma),\tau}\, v$ follow from (1). Now, using inductive hypothesis (q) for τ we know that $\mathsf{quote}\, v \Downarrow n$ (4) and $\ulcorner v \urcorner \simeq \ulcorner n \urcorner$ (5). By the definition of the big-step semantics and (2,4) we can infer that $\mathsf{quote}_{\Gamma,(\sigma\to\tau)}\, f \Downarrow \lambda^{\sigma} n$. Using the conversion rules in conjunction with (3),(5), lemma 7 and lemma 8 we can show that $\ulcorner f \urcorner \simeq \lambda^{\sigma} \ulcorner n \urcorner$.

(u) Given $\overline{\mathsf{quote}}_{\Gamma,(\sigma\to\tau)}\, n \Downarrow m$ (1) and $\ulcorner n \urcorner \simeq \ulcorner m \urcorner$ (2). By definition of **SCV** at arrow type, to show $\mathbf{SCV}_{\Gamma,(\sigma\to\tau)}\, n$ assume as given $f : \mathsf{OPE}\, B\, \Gamma$ and $\mathbf{SCV}_{B,\sigma}\, v$. Certainly $f * n @ v \Downarrow (f * n)\, v$ since n is neutral. By inductive hypothesis (q) for σ we know that $\mathsf{quote}_{B,\sigma}\, v \Downarrow u$ (3) and $\ulcorner v \urcorner \simeq \ulcorner u \urcorner$ (4). Hence $\overline{\mathsf{quote}}_{B,\tau}\, (f * n)\, v \Downarrow (f * m)\, u$ (5) from (1), (3) and lemma 9. From (2), (4) and lemma 8 we can infer $\ulcorner (f * n)\, v \urcorner \simeq \ulcorner (f * m)\, u \urcorner$ (6). $\mathbf{SCV}_{B,\tau}\, ((f * n)\, v)$ follows from (5) and (6) by inductive hypothesis (u) for τ.

□

A simple consequence of the 2nd component of the lemma is that variables are strongly computable and hence the identity environment is strongly computable.

Corollary 2.

$$\frac{x : \mathsf{Var}\,\Gamma\,\sigma}{\mathbf{SCV}\, \underline{\hat{x}}}(1) \qquad \frac{\Gamma : \mathsf{Con}}{\mathbf{SCE}\, \mathbf{id}_{\Gamma}}(2)$$

Proof.

(1) Since $\overline{\mathsf{quote}}_{\Gamma,\sigma}\, \hat{x} \Downarrow \hat{x}$, we just have to apply (u) of lemma 13.

(2) By induction over Γ using (1) and lemma 11.

□

Next we prove the fundamental theorem for our notion of strong computability which has to be shown mutually for terms and substitutions:

Theorem 4.

$$\frac{t \;:\; \mathsf{Tm}\,\Delta\,\sigma \quad \mathbf{SCE}_{\Gamma,\Delta}\,\vec{v}}{\exists v : \mathsf{Val}\,\Gamma\,\sigma.\, \mathsf{eval}\,t\,\vec{v} \Downarrow v \wedge t[\ulcorner \vec{v} \urcorner] \simeq \ulcorner v \urcorner \wedge \mathbf{SCV}\,v}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\,\Delta\,E \quad \mathbf{SCE}_{\Gamma,\Delta}\,\vec{v}}{\exists\, \vec{w} \;:\; \mathsf{Env}\,\Gamma\,E.\, \vec{\mathsf{eval}}\,\vec{t}\,\vec{v} \Downarrow \vec{w} \wedge \vec{t} \circ \ulcorner \vec{v} \urcorner \simeq \ulcorner \vec{w} \urcorner \wedge \mathbf{SCE}\,\vec{w}}$$

Proof. By induction over $t \;:\; \mathsf{Tm}\,\Delta\,\sigma$ and $\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta$ using the laws of the conversion relation and the definition of the big-step reduction relation. We assume as given $\mathbf{SCE}_{\Gamma,\Delta}\,\vec{v}$.

$\lambda^\sigma t$: We have immediately that $\mathsf{eval}\,(\lambda^\sigma t)\,\vec{v} \Downarrow \lambda^\sigma t[\vec{v}, -]$ and the equational component holds trivially. It remains to show that $\lambda^\sigma t[\vec{v}, -]$ is strongly computable. We assume $f : \mathsf{OPE}\,B\,\Gamma$ and $v : \mathsf{Val}\,\Gamma\,\sigma$ such that $\mathbf{SCV}_{B,\sigma}\,v$. By induction hypothesis for t with $\mathbf{SCE}_{B,\Delta}\,(f * \vec{v})$ (by lemma 12) there is a $w : \mathsf{Val}\,B\,\tau$ such that $\mathsf{eval}\,t\,(f * \vec{v}; v) \Downarrow w$ (1), $t[\ulcorner f * \vec{v}; v \urcorner] \simeq \ulcorner w \urcorner$ (2) and $\mathbf{SCV}_B\,w$. By definition of the big-step relation and (1) we have $(\lambda^\sigma t[f * \vec{v}, -]) \mathbin{@} v \Downarrow w$. By (2) and the rules of the conversion relation we can show that $((\lambda^\sigma t)[\ulcorner f * \vec{v} \urcorner]) \ulcorner v \urcorner \simeq \ulcorner w \urcorner$.

$(t\,u)$: By induction hypothesis for t we get $\mathsf{eval}\,t\,\vec{v} \Downarrow f$ (1), $t[\ulcorner \vec{v} \urcorner] \simeq \ulcorner f \urcorner$ (2) and $\mathbf{SCV}_{\Gamma,(\sigma\to\tau)}\,f$ (3). By induction hypothesis for u we can infer $\mathsf{eval}\,u\,\vec{v} \Downarrow v$ (4), $u[\ulcorner \vec{v} \urcorner] \simeq_{\mathsf{w}\sigma} \ulcorner v \urcorner$ (5) and $\mathbf{SCV}_{\Gamma,\sigma}\,v$ (6). By (3) with $\mathbf{id} : \mathsf{OPE}\,\Gamma\,\Gamma$, (6) and lemma 5 (for values) we get that $f \mathbin{@} v \Downarrow w$ (7), $\ulcorner f \urcorner \ulcorner v \urcorner \simeq \ulcorner w \urcorner$ (8) and $\mathbf{SCV}_{\Gamma,\tau}\,w$. Using the definition of the big-step semantics and (1,4,7) we can show that $\mathsf{eval}\,(t\,u)\,\vec{v} \Downarrow w$ and $(t\,u)[\ulcorner \vec{v} \urcorner] \simeq \ulcorner w \urcorner$ using the rules of conversion and (2,5,8).

$(\vec{t}; t)$: By induction hypothesis for $\vec{t}$ we get $\vec{\mathsf{eval}}\,\vec{t}\,\vec{v} \Downarrow \vec{w}$ (1), $\vec{t} \circ \ulcorner \vec{v} \urcorner \simeq \ulcorner \vec{w} \urcorner$ (2) and $\mathbf{SCE}\,\vec{w}$ (3). Using the latter with the induction hypothesis for t we have that $\vec{\mathsf{eval}}\,t\,\vec{v} \Downarrow v$ (4), $t[\ulcorner \vec{v} \urcorner] \simeq \ulcorner v \urcorner$ (5) and $\mathbf{SCV}\,v$ (6). The definition of the big-step reduction and (1,4) imply that $\vec{\mathsf{eval}}\,(\vec{t}; t)\,\vec{v} \Downarrow (\vec{w}; v)$. Using the conversion rules and (2,5) we can show $(\vec{t}; t) \circ \ulcorner \vec{v} \urcorner \simeq \ulcorner \vec{w}; v \urcorner$ and $\mathbf{SCE}\,(\vec{w}, v)$ by (3,6).

□

We now combine the results to infer that **nf** terminates and produces a normal form which is $\beta\eta$-equivalent to its input.

Proposition 4.

$$\frac{t \;:\; \mathsf{Tm}\,\Delta\,\sigma}{\exists n : \mathsf{Nf}\,\Delta\,\sigma\,.\,\mathsf{nf}\;t \Downarrow n \;\wedge\; t \simeq \ulcorner n \urcorner}$$

Proof. By the fundamental theorem 4, corollary 2, lemma 7 and lemma 13. □

Since we now know that our functions terminate, we can from now on use the total functions defined in section 4.4 together with the termination proofs given in this section.

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathbf{nf}\,t \;:\; \mathsf{Nf}\,\Gamma\,\sigma} \quad \underline{\text{where}} \quad \mathbf{nf}\,t \;\Rightarrow\; \mathbf{nf}^{\mathbf{str}}\;t\;(\pi_0\;(\mathbf{prop4}\;t))$$

To ease notation we will omit the proof terms altogether but make sure that we only use strongly computable values and environments.

4.7 Soundness

It remains to be shown that normalisation maps $\beta\eta$-equivalent terms to equal normal forms. We define a logical relation on values which is preserved by the values obtained from convertible terms and which is mapped to identical normal forms by quote.

$$\frac{v, w \;:\; \mathsf{Val}\,\Gamma\,\sigma}{v \sim_{\Gamma,\sigma} w \;:\; \mathsf{Prop}}$$

$$\begin{array}{lcll} m & \sim_{\Gamma,\bullet} & n & = \quad \overline{\mathbf{quote}}\;m = \overline{\mathbf{quote}}\;n \\ f & \sim_{\Gamma,(\sigma\to\tau)} & g & = \quad \forall o : \mathsf{OPE}\,B\,\Gamma\,.\,\forall v, w : \mathsf{Val}\,B\,\sigma\,.\,v \;\sim_{B,\sigma}\; w \;\to \\ & & & \qquad o^* f \mathbin{@} v \;\sim_{B,\tau}\; o^* g \mathbin{@} w \end{array}$$

The pointwise extension to environments is straightforward:

$$\frac{\vec{v}, \vec{w} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\vec{v} \sim \vec{w} \;:\; \mathsf{Prop}}$$

$$\begin{array}{lcll} \varepsilon & \sim & \varepsilon & = \quad \mathbf{True} \\ (\vec{v}; v) & \sim & (\vec{w}; w) & = \quad \vec{v} \sim \vec{w} \wedge v \sim w \end{array}$$

As before for strong computability we will need that $\sim$ is closed under OPE:

Lemma 14.

$$\frac{v \sim_{\Gamma,\sigma} w \quad o \;:\; \mathsf{OPE}\,B\,\Gamma}{o^*\,v \sim_{B,\sigma} o^*\,w}(1) \qquad \frac{\vec{v} \sim_{\Gamma,E} \vec{w} \quad o \;:\; \mathsf{OPE}\,B\,\Gamma}{o^*\,v \sim_{B,E} o^*\,w}(2)$$

Proof.

1. By induction over σ. In the base case we need lemma 10 for $\overline{\mathbf{quote}}$ and in the arrow case we need lemma 6.

2. By induction over E. The case for the empty context is trivial and the case for the non-empty context follows from inductive hypothesis and (1).

□

Lemma 15. *$-\sim_{\Gamma,\sigma}-$ for values (1) and $-\sim_{\Gamma,E}-$ for environments (2) are partial equivalence relations.*

Proof.

1. We first show symmetry and then transitivity, both are by induction on the type σ. In the arrow case for transitivity we exploit the property of partial equivalence relations that any element in the relation is related to itself. We prove symmetry first because the mentioned property exploits it.

2. Both symmetry and transitivity are by induction on E and require symmetry and transitivity of the value relation respectively in the non-empty case.

□

Before we can establish the fundamental theorem for logical relations we have to show an *identity extension lemma*:

Lemma 16.

$$\frac{t : \mathsf{Tm}\ \Gamma\ \sigma \qquad \vec{v} \sim \vec{w}}{\mathbf{eval}\ t\ \vec{v} \sim \mathbf{eval}\ t\ \vec{w}}(1) \qquad \frac{\vec{t} : \mathsf{Sub}\ \Gamma\ \Delta \qquad \vec{v} \sim \vec{w}}{\vec{\mathbf{eval}}\ \vec{t}\ \vec{v} \sim \vec{\mathbf{eval}}\ \vec{u}\ \vec{w}}(2)$$

Proof. By simultaneous induction over t and $\vec{t}$.

1. In the case for lambda $(\lambda\ t)$ we need lemma 14. In the case for application $(t\ u)$ we need lemma 5.

2. The proof is straightforward.

□

To show that quote maps equivalent values to equal normal forms, we have to simultaneously establish a dual property, as before for strong computability.

Lemma 17.

$$\frac{v \sim_{\Gamma,\sigma} w}{\mathbf{quote}_{\Gamma,\sigma}\, v = \mathbf{quote}_{\Gamma,\sigma}\, w}(q) \qquad \frac{\overline{\mathbf{quote}}_{\Gamma,\sigma}\, m = \overline{\mathbf{quote}}_{\Gamma,\sigma}\, n}{\underline{m} \sim_{\Gamma,\sigma} \underline{n}}(u)$$

Proof. By induction over σ. For base types both properties follow directly from the definition of $\sim$ and the observation that all values of base type are neutral. We show both properties for $(\sigma \to \tau)$:

(q) Given $f \sim_{\Gamma,(\sigma\to\tau)} g$ (1), we need that $\mathbf{quote}_{\Gamma,(\sigma\to\tau)}\, f = \mathbf{quote}_{\Gamma,(\sigma\to\tau)}\, g$. This computes to $\lambda^{\sigma}\mathbf{quote}_{(\Gamma;\sigma),\tau}\,(f^{+\sigma}@\hat{\varnothing}) = \lambda^{\sigma}\mathbf{quote}_{(\Gamma;\sigma),\tau}\,(g^{+\sigma}@\hat{\varnothing})$ (2). Using inductive hypothesis (u) for σ we can show $\hat{\varnothing} \sim_{(\Gamma;\sigma),\sigma} \hat{\varnothing}$ (3) and hence by the definition of $\sim$ applied to the weakening OPE ($\mathbf{weak}\,\sigma$) and (3) we get $f^{+\sigma}@\hat{\varnothing} \sim_{(\Gamma;\sigma),\tau} g^{+\sigma}@\hat{\varnothing}$ (4). Applying inductive hypothesis (q) for τ to (4) gives $\mathbf{quote}_{(\Gamma;\sigma),\tau}\,(f^{+\sigma}@\hat{\varnothing}) = \mathbf{quote}_{(\Gamma;\sigma),\tau}\,(f^{+\sigma}@\hat{\varnothing})$, and our goal (2) is a simple consequence.

(u) Given $\overline{\mathbf{quote}}_{\Gamma,(\sigma\to\tau)}\, m = \overline{\mathbf{quote}}_{\Gamma,(\sigma\to\tau)}\, n$ (1) we show $\underline{m} \sim_{\Gamma,(\sigma\to\tau)} \underline{n}$. Unfolding the definition of $\sim$ this means that given $f : \mathsf{OPE}\, B\, \Gamma$, $v \sim_{B,\sigma} w$ (2) we have to show that $f * \underline{m}@v \sim_{B,\tau} f * \underline{n}@w$ which computes to $\underline{(f * m)}\, v \sim_{B,\tau} \underline{(f * n)}\, w$ Using the induction hypothesis (u) for τ this is reduced to showing that $\overline{\mathbf{quote}}_{B,\tau}\,(f * m)\, v = \overline{\mathbf{quote}}_{B,\tau}\,(f * n)\, w$. This follows from (1) by lemma 10 and $\mathbf{quote}_{B,\sigma}\, v = \mathbf{quote}_{B,\sigma}\, w$ which we can show by using induction hypothesis (q) for σ with (2).

□

We can exploit the second property (q) to show that variables are related to themselves and the identity environment is related to itself.

Corollary 3.

$$\frac{x : \mathsf{Var}\,\Gamma\,\sigma}{\hat{x} \sim \hat{x}} \qquad \frac{\Gamma : \mathsf{Con}}{\mathbf{id}_\Gamma \sim \mathbf{id}_\Gamma}$$

Next we show the fundamental theorem of logical relations:

Theorem 5.

$$\frac{t \simeq u \qquad \vec{v} \sim \vec{w}}{\mathbf{eval}\, t\, \vec{v} \sim \mathbf{eval}\, u\, \vec{w}} \qquad \frac{\vec{t} \simeq \vec{u} \qquad \vec{v} \sim \vec{w}}{\vec{\mathbf{eval}}\, \vec{t}\, \vec{v} \sim \vec{\mathbf{eval}}\, \vec{u}\, \vec{w}}$$

Proof. By mutual induction over the derivation of $t \simeq u$ and $\vec{t} \simeq \vec{u}$, as before we consider some typical cases. We assume that $\vec{v} \sim \vec{w}$(H).

refl**:** Reflexivity follows from lemma 16.

sym**:** Symmetry follows from lemma 15.

trans**:** Transitivity follows from lemma 15.

ξ**:** We have to show $\mathbf{eval}\,(\lambda^\sigma t)\,\vec{v} \sim_{B,(\sigma\to\tau)} \mathbf{eval}\,(\lambda^\sigma u)\,\vec{w}$. This computes to showing $\lambda^\sigma t[\vec{v}, -] \sim_{B,(\sigma\to\tau)} \lambda^\sigma u[\vec{w}, -]$. Given $f : \mathsf{OPE}\ B\ \Gamma$ and $v \sim_{B,\sigma} w$ we have to show that $\lambda^\sigma t[f^*\,\vec{v}, -]@v \sim_{B,\tau} \lambda^\sigma u[f^*\,\vec{w}, -]@w$ which computes to $\mathbf{eval}\,t\,(f^*\,\vec{v}; v) \sim_{B,\sigma} \mathbf{eval}\,u\,(f^*\,\vec{w}; w)$ this follows from the induction hypothesis, and lemma 14 applied to (H).

β**:** We have to show $\mathbf{eval}\,((\lambda^\sigma t)\,u)\,\vec{v} \sim \mathbf{eval}\,(t[\mathsf{id}; u])\,\vec{w}$. This reduces to having to show $\mathbf{eval}\,t\,(\vec{v}; \mathbf{eval}\,u\,\vec{v}) \sim \mathbf{eval}\,t\,(\vec{w}; \mathbf{eval}\,u\,\vec{w})$. This follows from applying lemma 16 to u and (H) to give $\mathbf{eval}\,u\,\vec{v} \sim \mathbf{eval}\,u\,\vec{w}$ (1), and lemma 16 to t and (H) paired with (1).

assoc**:** We have to show $\vec{\mathbf{eval}}\,((\vec{s} \circ \vec{t}) \circ \vec{u})\,\vec{v} \sim \vec{\mathbf{eval}}\,(\vec{s} \circ (\vec{t} \circ \vec{u}))\,\vec{w}$. This reduces to showing $\vec{\mathbf{eval}}\,\vec{s}\,(\vec{\mathbf{eval}}\,\vec{t}\,(\vec{\mathbf{eval}}\,\vec{u}\,\vec{v})) \sim \vec{\mathbf{eval}}\,\vec{s}\,(\vec{\mathbf{eval}}\,\vec{t}\,(\vec{\mathbf{eval}}\,\vec{u}\,\vec{w}))$, this follows again by lemma 16. It is applied first to $\vec{u}$ and (H) to give (1) then to $\vec{t}$ and (1) to give (2) and finally to $\vec{s}$ and (2).

□

By putting everything together we can establish soundness of the normalisation function:

Proposition 5.

$$\frac{t \simeq u}{\mathbf{nf}\ t = \mathbf{nf}\ u}$$

Proof. Using corollary 3 and theorem 5 we can infer that $\mathbf{eval}\,t\,\mathbf{id} \sim \mathbf{eval}\,u\,\mathbf{id}$ and by lemma 17 we obtain the result. □

4.8 Extensions

In this section we extend simply typed λ-calculus with finite products (with $\beta\eta$-equality) and with natural numbers (with β-equality rules). Extending the system to include $\beta\eta$-equality for finite coproducts is much more demanding and this is left for future work.

Finite Products

We add the unit type and binary products to the definition of types:

$$\frac{}{1 \;:\; \mathsf{Ty}} \qquad \frac{\sigma \;:\; \mathsf{Ty} \qquad \tau \;:\; \mathsf{Ty}}{\sigma \times \tau \;:\; \mathsf{Ty}}$$

Next we extend the syntax with the constant void (the only element of the unit type), pairing and projection:

$$\frac{}{[] \;:\; \mathsf{Tm}\,\Gamma\,1} \qquad \frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma \qquad u \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathsf{pr}\,t\,u \;:\; \mathsf{Tm}\,\Gamma\,(\sigma \times \tau)}$$

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,(\sigma \times \tau)}{\mathsf{fst}\,t \;:\; \mathsf{Tm}\,\Gamma\,\sigma} \qquad \frac{t \;:\; \mathsf{Tm}\,\Gamma\,(\sigma \times \tau)}{\mathsf{snd}\,t \;:\; \mathsf{Tm}\,\Gamma\,\tau}$$

To the conversion rule we add congruence rules for pairing and projection. We also add rules for pushing substitutions through pairing and projection

$$\begin{array}{lcll}
[][\vec{t}] & \simeq & [] & \mathsf{convoid} \\
(\mathsf{pr}\,t\,u)[\vec{t}] & \simeq & \mathsf{pr}\,(t[\vec{t}])\,(u[\vec{t}]) & \mathsf{conpr} \\
(\mathsf{fst}\,t)[\vec{t}] & \simeq & \mathsf{fst}\,(t[\vec{t}]) & \mathsf{confst} \\
(\mathsf{snd}\,t)[\vec{t}] & \simeq & \mathsf{snd}\,(t[\vec{t}]) & \mathsf{consnd}
\end{array}$$

and β and η for pairs and η for unit

$$\begin{array}{lcll}
\mathsf{fst}\,(\mathsf{pr}\,t\,u) & \simeq & t & \mathsf{confst}\beta \\
\mathsf{snd}\,(\mathsf{pr}\,t\,u) & \simeq & u & \mathsf{consnd}\beta \\
t & \simeq & \mathsf{pr}\,(\mathsf{fst}\,t)\,(\mathsf{snd}\,t) & \mathsf{conpr}\eta \\
t & \simeq & [] & \mathsf{convoid}\eta
\end{array}$$

We add constructors to Val for the constructor forms of finite products: void and pairing.

$$\frac{}{()^{\mathsf{v}} \;:\; \mathsf{Val}\,\Gamma\,1} \qquad \frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma \qquad w \;:\; \mathsf{Val}\,\Gamma\,\tau}{\mathsf{vpr}\,v\,w \;:\; \mathsf{Val}\,\Gamma\,(\sigma \times \tau)}$$

We add neutral terms to represent stuck projections:

$$\frac{n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,(\sigma \times \tau)}{\mathsf{nfst}\,n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma} \qquad \frac{n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,(\sigma \times \tau)}{\mathsf{nsnd}\,n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\tau}$$

We add functions to compute the projections:

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,(\sigma \times \tau)}{\mathbf{vfst}\,v \;:\; \mathsf{Val}\,\Gamma\,\sigma} \qquad \begin{array}{lll} \mathbf{vfst} & (\mathsf{vpr}\,v\,w) & = \; v \\ \mathbf{vfst} & \underline{n} & = \; \underline{\mathsf{nfst}\,n} \end{array}$$

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,(\sigma \times \tau)}{\mathbf{vsnd}\,v \;:\; \mathsf{Val}\,\Gamma\,\tau} \qquad \begin{array}{lll} \mathbf{vsnd} & (\mathsf{vpr}\,v\,w) & = \; w \\ \mathbf{vsnd} & \underline{n} & = \; \underline{\mathsf{nsnd}\,n} \end{array}$$

Correspondingly the following clauses are added to the evaluator:

$$\begin{array}{llll} \mathbf{eval} & (\mathsf{pr}\ t\ u) & \vec{v} & = & \mathsf{vpr}\ (\mathbf{eval}\ t\ \vec{v})\ (\mathbf{eval}\ u\ \vec{v}) \\ \mathbf{eval} & (\mathsf{fst}\ t) & \vec{v} & = & \mathbf{vfst}\ (\mathbf{eval}\ t\ \vec{v}) \\ \mathbf{eval} & (\mathsf{snd}\ t) & \vec{v} & = & \mathbf{vsnd}\ (\mathbf{eval}\ t\ \vec{v}) \end{array}$$

Normal forms (and neutral normal forms) are extended analogously to values (and neutral values):

$$\frac{}{()^{\mathsf{n}}\ :\ \mathsf{Nf}\ \Gamma\ 1} \qquad \frac{m\ :\ \mathsf{Nf}\ \Gamma\ \sigma \quad n\ :\ \mathsf{Nf}\ \Gamma\ \tau}{\mathsf{npr}\ m\ n\ :\ \mathsf{Nf}\ \Gamma\ (\sigma\times\tau)}$$

$$\frac{n\ :\ \mathsf{Ne}^{\mathsf{Nf}}\ \Gamma\ (\sigma\times\tau)}{\mathsf{nfst}\ n\ :\ \mathsf{Ne}^{\mathsf{Nf}}\ \Gamma\ \sigma} \qquad \frac{n\ :\ \mathsf{Ne}^{\mathsf{Nf}}\ \Gamma\ (\sigma\times\tau)}{\mathsf{nsnd}\ n\ :\ \mathsf{Ne}^{\mathsf{Nf}}\ \Gamma\ \tau}$$

We add the following clauses to **quote**:

$$\begin{array}{lll} \mathbf{quote}_1 & v & = & ()^{\mathsf{n}} \\ \mathbf{quote}_{\sigma\times\tau} & (\mathsf{vpr}\ v\ w) & = & \mathsf{npr}\ (\mathbf{quote}_\sigma\ v)\ (\mathbf{quote}_\tau\ w) \end{array}$$

And the following clauses to $\overline{\mathbf{quote}}$:

$$\begin{array}{lll} \overline{\mathbf{quote}} & (\mathsf{nfst}\ n) & = & \mathsf{nfst}\ (\overline{\mathbf{quote}}\ n) \\ \overline{\mathbf{quote}} & (\mathsf{nsnd}\ n) & = & \mathsf{nsnd}\ (\overline{\mathbf{quote}}\ n) \end{array}$$

The following clauses are added to **SCV**:

$$\begin{array}{lll} \mathbf{SCV}_1 & v & = \ \mathbf{True} \\ \mathbf{SCV}_{\sigma\times\tau} & p & = \\ \multicolumn{3}{l}{\quad (\Sigma v : \mathsf{Val}\ \Gamma\ \sigma.\ \mathsf{fst}\ p \Downarrow v\ \wedge\ \mathbf{SCV}_\sigma\ v\ \wedge\ \mathsf{fst}\ \ulcorner p \urcorner \simeq \ulcorner v \urcorner) \wedge} \\ \multicolumn{3}{l}{\quad (\Sigma v : \mathsf{Val}\ \Gamma\ \tau.\ \mathsf{snd}\ p \Downarrow v\ \wedge\ \mathbf{SCV}_\tau\ v\ \wedge\ \mathsf{snd}\ \ulcorner p \urcorner \simeq \ulcorner v \urcorner)} \end{array}$$

The soundness relation is extended as follows:

$$\begin{array}{llll} v & \sim_1 & w & = \ \mathbf{True} \\ v & \sim_{\sigma\times\tau} & w & = \ (\mathbf{vfst}\ v \sim_\sigma \mathbf{vfst}\ w) \wedge (\mathbf{vsnd}\ v \sim_\tau \mathbf{vsnd}\ w) \end{array}$$

All the other operations and proofs are easily extended.

Natural numbers

It is straightforward to extend our system to include a type for natural numbers with β-equality. This time we will replace the base type with the type of

natural numbers. We add N to types (removing $\bullet$) and extend the syntax of terms with zero (zero), successor (suc) and primitive recursion (prec).

$$\frac{}{\mathsf{zero} \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N}} \qquad \frac{t \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N}}{\mathsf{suc}\,t \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N}}$$

$$\frac{n \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N} \quad f \;:\; \mathsf{Tm}\,\Gamma\,(\mathsf{N}{\to}\sigma{\to}\sigma) \quad z \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathsf{prec}\,n\,f\,z \;:\; \mathsf{Tm}\,\Gamma\,\sigma}$$

We add the following $\simeq$ rules to the equational theory (and congruences for suc and prec):

$$\begin{array}{lcll} \mathsf{prec}\,\mathsf{zero}\,f\,z & \simeq & z & \mathsf{cprimrecz} \\ \mathsf{prec}\,(\mathsf{suc}\,n)\,f\,z & \simeq & f\,n\,(\mathsf{prec}\,n\,f\,z) & \mathsf{cprimrecs} \end{array}$$

Values Val and normal forms are extended with zero and suc and neutral terms Ne with a constructor to represent primitive recursion applied to a neutral natural number:

$$\frac{}{\mathsf{zero} \;:\; \mathsf{Val}\,\Gamma\,\mathsf{N}} \quad \frac{v \;:\; \mathsf{Val}\,\Gamma\,\mathsf{N}}{\mathsf{suc}\,v \;:\; \mathsf{Val}\,\Gamma\,\mathsf{N}} \quad \frac{}{\mathsf{zero} \;:\; \mathsf{Nf}\,\Gamma\,\mathsf{N}} \quad \frac{n \;:\; \mathsf{Nf}\,\Gamma\,\mathsf{N}}{\mathsf{suc}\,n \;:\; \mathsf{Nf}\,\Gamma\,\mathsf{N}}$$

$$\frac{n \;:\; \mathsf{Ne}^T\,\Gamma\,\mathsf{N} \quad f \;:\; T\,\Gamma\,(\mathsf{N}{\to}\sigma{\to}\sigma) \quad z \;:\; T\,\Gamma\,\sigma}{\mathsf{prec}\,n\,f\,z \;:\; \mathsf{Ne}^T\,\Gamma\,\sigma}$$

A separate semantic primitive recursor $\mathbf{pr}$ is added and $\mathbf{eval}$ extended to accommodate it:

$$\frac{n \;:\; \mathsf{Val}\Gamma\mathsf{N} \quad f \;:\; \mathsf{Val}\Gamma(\mathsf{N}{\to}\sigma{\to}\sigma) \quad z \;:\; \mathsf{Val}\Gamma\sigma}{\mathbf{pr}\,n\,f\,z \;:\; \mathsf{Val}\Gamma\sigma}$$

$$\begin{array}{llllll} \mathbf{pr} & \mathsf{zero} & f & z & \Rightarrow & z \\ \mathbf{pr} & (\mathsf{suc}\,n) & f & z & \Rightarrow & f\,@\,n\,@\,(\mathbf{pr}\,n\,f\,z) \end{array}$$

$$\begin{array}{llll} \mathbf{eval} & \mathsf{zero} & \vec{v} & \Rightarrow \quad \mathsf{zero} \\ \mathbf{eval} & (\mathsf{suc}\,n) & \vec{v} & \Rightarrow \quad \mathsf{suc}\,(\mathbf{eval}\,n\,\vec{v}) \\ \mathbf{eval} & (\mathsf{prec}\,n\,f\,z) & \vec{v} & \Rightarrow \quad \mathbf{pr}\,(\mathbf{eval}\,n\,\vec{v})\,(\mathbf{eval}\,f\,n)\,(\mathbf{eval}\,z\,n) \end{array}$$

For $\mathbf{quote}$ we replace the case for $\mathbf{quote}_\bullet$ with cases for $\mathbf{quote}_\mathsf{N}$:

$$\begin{array}{llll} \mathbf{quote}_\mathsf{N} & \mathsf{zero} & \Rightarrow & \mathsf{zero} \\ \mathbf{quote}_\mathsf{N} & (\mathsf{suc}\,n) & \Rightarrow & \mathsf{suc}\,(\mathbf{quote}_\mathsf{N}\,n) \\ \mathbf{quote}_\mathsf{N} & \underline{n} & \Rightarrow & \underline{\mathbf{quote}\,n} \end{array}$$

Next we replace the base case $\bullet$ in the definitions of $\mathbf{SCV}$ and $\sim$:

$$\begin{array}{lll} \mathbf{SCV}_{\Gamma,\mathsf{N}}\ \mathsf{zero} & = & \mathbf{True} \\ \mathbf{SCV}_{\Gamma,\mathsf{N}}\ (\mathsf{suc}\ n) & = & \mathbf{SCV}_{\Gamma,\mathsf{N}}\ n \\ \mathbf{SCV}_{\Gamma,\mathsf{N}}\ \underline{n} & = & \overline{\mathsf{quote}}\ n \Downarrow m \wedge \ulcorner n \urcorner \simeq \ulcorner m \urcorner \end{array}$$

$$\begin{array}{llll} \mathsf{zero} & \sim_{\mathsf{N}} & \mathsf{zero} & = & \mathbf{True} \\ \mathsf{suc}\ m & \sim_{\mathsf{N}} & \mathsf{suc}\ n & = & m \sim_{\mathsf{N}} n \\ \underline{m} & \sim_{\mathsf{N}} & \underline{n} & = & \overline{\mathbf{quote}}\ m = \overline{\mathbf{quote}}\ n \end{array}$$

We also require an extra lemma to prove the fundamental theorem in the case for $\mathsf{prec}\ z\ s\ t$. This is needed for the extra induction we must do on the value of t.

Lemma 18.

$$\frac{\mathbf{SCV}_{\Gamma,(\mathsf{N}\to\sigma\to\sigma)}\ f \quad \mathbf{SCV}_{\Gamma\ \sigma}\ z \quad \mathbf{SCV}_{\Gamma,\mathsf{N}}\ n}{\exists v : \mathsf{Val}\,\Gamma\,\sigma\ .\ \mathsf{pr}\ f\ z\ n \Downarrow v \wedge \mathsf{prec}\ \ulcorner f \urcorner\ \ulcorner z \urcorner\ \ulcorner n \urcorner \simeq \ulcorner v \urcorner \wedge \mathbf{SCV}\ v}$$

Proof. By induction over n. □

Having built the necessary machinery we can easily extend the rest of the operations and proofs to accommodate natural numbers.

4.9 Chapter summary

In this chapter we have defined a terminating, sound and complete $\beta\eta$-normaliser for simply typed λ-calculus, extended with finite products and natural numbers. In the next section we give a partial definition of a normaliser for a system with dependent types.

The next step with this work would be to consider finite coproducts. η-equality for coproducts is notoriously difficult to deal with [45, 11].

Chapter 5

Dependently typed λ-calculi

In this chapter we generalise the simple function space ($\sigma \to \tau$) of the previous chapter to the dependent function space ($\Pi x\!:\!S.\,T$). Apart from this, the idea is to stick closely to the recipe we have followed in the previous chapter, write a normalisation algorithm, and then show it is terminating, sound and complete. I stop short of carrying this out in full and just show the algorithm and its completeness property here. These aspects have been formalised in Agda. Even this is of equivalent size to the entire formalisation for simple types. Danielsson has carried out a similar formalisation [33] and proves termination and completeness (but not soundness) in Agda's immediate predecessor Agda-Light. His formalisation is approximately 10,000 lines: ten times the size of my formalisation for simple types and my partial formalisation for dependent types (which is presented here).

I have been careful to design the algorithm in such a way that the technique we have been developing throughout this thesis is applicable. By this I mean sticking closely to the simply typed version, using a first order representation of values, and a first order implementation of the evaluator. I have also been careful to try to use only relatively well understood facilities of the Agda system.

Danielsson's formalisation is closely related: we both formalise Martin-Löf's logical framework, we both use well-typed syntax and we both use explicit substitutions. We differ in a number of ways:

- His formalisation is more complete as he proves termination.
- He formalises NBE as opposed to BSN.

- He uses mutual definitions that go beyond what is understood to be induction-recursion.
- He uses a negative inductive definition of values.
- He does not use explicit substitutions at the type level so his syntax is not first order.

The syntax I use is heavily inspired by Dybjer's "categories with families". In part it is almost identical to a syntactic realisation of them. The main difference is that I use a heterogeneous notion of definitional equality as this simplifies implementation significantly.

Relationship to published paper This chapter is based on the paper "Type theory should eat itself" [24]. A section on β-normalisation has been added before the section on $\beta\eta$-normalisation. In the version presented in the paper I wanted to push the development as far as possible. However, indexing values by syntactic types make type directed operations awkward and hence the β-only version (where this is not necessary) is much cleaner. I tried and failed to complete an implementation with values indexed by value types. I hope to pursue this further in the future.

5.1 Syntax

The system presented here is the Martin-Löf's logical framework with explicit substitutions. The judgments are defined mutually as follows:

$$\begin{array}{lcl} \mathsf{Con} & : & \star \\ \mathsf{Ty} & : & \mathsf{Con} \to \star \\ \mathsf{Tm} & : & (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \star \\ \mathsf{Sub} & : & \mathsf{Con} \to \mathsf{Con} \to \star \end{array}$$

Notice firstly that types are indexed by their contexts and terms are indexed by their type and their context. Notice secondly that, as before, there are no definitions for raw syntax. In type theory terms always have a type and by giving the constructors of a type we explain what it means to be a term of that type. Terms on their own do not have a meaning. Also, we cannot separate the equation syntax from the syntax of well typed terms. Hence the

corresponding equality judgements must also be mutually defined:

$$\begin{aligned}
\simeq &: \mathsf{Con} \to \mathsf{Con} \to \star \\
\simeq &: \mathsf{Ty}\,\Gamma \to \mathsf{Ty}\,\Gamma' \to \star \\
\simeq &: \mathsf{Tm}\,\Gamma\,\sigma \to \mathsf{Tm}\,\Gamma'\,\sigma' \to \star \\
\simeq &: \mathsf{Sub}\,\Gamma\,\Delta \to \mathsf{Sub}\,\Gamma'\,\Delta' \to \star
\end{aligned}$$

This is because of coercion constructors defined below. At the level of types, terms and substitutions there is a constructor which takes a proof of equality as an argument and hence the for eight judgements must be defined simultaneously. By including the definitional equality and types and contexts in the syntax we are in effect encoding typing derivations as part of the syntax.

Note also that this is the decidable definitional equality that a typechecker would use. We do not consider propositional equality here which is for things that are provably equal.

We now explain one at a time how to construct elements of these eight sets, starting with contexts. Contexts are left-to-right sequences of types and are nameless as we use de Bruijn indices [36], as before. There are two ways to construct a context either it is the empty context or it is an existing context given together with a type indexed by that context.

$$\underline{\mathsf{data}}\quad \frac{}{\mathsf{Con} : \star}\quad \underline{\mathsf{where}}\quad \frac{}{\varepsilon : \mathsf{Con}}\quad \frac{\Gamma : \mathsf{Con}\quad \sigma : \mathsf{Ty}\,\Gamma}{(\Gamma;\sigma) : \mathsf{Con}}$$

Types are indexed by contexts:

$$\underline{\mathsf{data}}\quad \frac{\Gamma : \mathsf{Con}}{\mathsf{Ty}\,\Gamma : \star}\quad \underline{\mathsf{where}}$$

The syntax is fully explicit about uses of the context and type coercions. By coercions I mean casts from one context, or one type to another. These correspond to the conversion rules in the traditional syntax. There is a constructor for this even at the level of types. Given a type in context Γ, and if Γ and Δ are equal contexts then we have a type in Δ.

$$\frac{\sigma : \mathsf{Ty}\,\Gamma\quad p : \Gamma = \Delta}{\mathsf{coe}\,\sigma\,p : \mathsf{Ty}\,\Delta}$$

The behaviour of coercions aregoverned by coherence rules (defined below) which state that a coerced object is definitionally equal to the uncoerced self.

Carrying on with the constructors for types we have a constructor for explicit substitutions at the level of types.

$$\frac{\sigma \;:\; \mathsf{Ty}\,\Delta \quad \vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta}{\sigma[\vec{t}] \;:\; \mathsf{Ty}\,\Gamma}$$

In Danielsson's presentation of the syntax of this system he does not include explicit substitutions at the level of types. He has explicit substitutions at the level of terms but at the level of types he chooses to define substitution recursively. This recursive definition then forms part of the definition of the syntax as it must be defined mutually. His approach has the advantage of simplifying the treatment of type equality and reduces the number of properties that must be postulated in the syntax. However this cannot readily be extended to more complex systems such as having a universe closed under Π-types.

The remaining constructors cover the universe (which contains only neutral terms), embedding codes for types from the universe into types and dependent functions (Π-types). Notice that the second argument (the range) to Π has an extra variable in its context (the domain).

$$\frac{}{\mathsf{U} \;:\; \mathsf{Ty}\,\Gamma} \qquad \frac{\sigma \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{U}}{\mathsf{El}\,\sigma \;:\; \mathsf{Ty}\,\Gamma} \qquad \frac{\sigma \;:\; \mathsf{Ty}\,\Gamma \quad \tau \;:\; \mathsf{Ty}\,(\Gamma;\sigma)}{\Pi\,\sigma\,\tau \;:\; \mathsf{Ty}\,\Gamma}$$

Terms are indexed by context and type and include explicit constructors for coercions and substitutions:

$$\underline{\text{data}} \quad \frac{\Gamma \;:\; \mathsf{Con} \quad \sigma \;:\; \mathsf{Ty}\,\Gamma}{\mathsf{Tm}\,\Gamma\,\sigma \;:\; \star}\,\underline{\text{where}}$$

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma \quad p \;:\; \sigma \simeq \sigma'}{\mathsf{coe}\,t\,p \;:\; \mathsf{Tm}\,\Gamma'\,\sigma'} \qquad \frac{t \;:\; \mathsf{Tm}\,\Delta\,\sigma \quad \vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\sigma}{t[\vec{t}] \;:\; \mathsf{Tm}\,\Gamma\,(\sigma[\vec{t}])}$$

Variables are represented analogously to the treatment in the previous chapter. In the syntax we do not single out variables. Instead we have $\varnothing$ which is the first bound variable and then other variables are obtained by applications of the weakening substitution (which is called $\uparrow$ and is introduced below) ($\varnothing[\uparrow^\sigma]$, $\varnothing[\uparrow^\sigma \bullet \uparrow^\tau]$, etc.) to $\varnothing$. The difference to the previous chapter is that we have to weaken the types of variables in the types of their constructors. In the definition of $\varnothing$ below we weaken the type σ by itself to move it from the context Γ to the context $(\Gamma;\sigma)$.

$$\frac{}{\varnothing \;:\; \mathsf{Tm}\,(\Gamma;\sigma)\,(\sigma[\uparrow^\sigma])}$$

The categorical combinator for application is included rather than conventional application as it simplifies the presentation of the syntax and evaluation. The standard notion of application is defined later as a convenience.

$$\frac{t \ : \ \mathsf{Tm}\,(\Gamma;\sigma)\,\tau}{\lambda\, t \ : \ \mathsf{Tm}\,\Gamma\,(\Pi\,\sigma\,\tau)} \qquad \frac{t \ : \ \mathsf{Tm}\,\Gamma\,(\Pi\,\sigma\,\tau)}{\mathsf{ap}\, t \ : \ \mathsf{Tm}\,(\Gamma;\sigma)\,\tau}$$

Next we define substitutions. Instead of presenting substitutions as just sequences of terms and then defining identity and composition recursively we give constructors for all the necessary operations. This gives us a first order syntax, otherwise we would need to define the operations recursively, and mutually with the syntax.

$$\underline{\text{data}} \quad \frac{\Gamma \ : \ \mathsf{Con} \quad \Delta \ : \ \mathsf{Con}}{\mathsf{Sub}\,\Gamma\,\Delta} \quad \underline{\text{where}} \quad \frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta \quad p \ : \ \Gamma \simeq \Gamma' \quad q \ : \ \Delta \simeq \Delta'}{\mathsf{coe}\,\vec{t}\,p\,q \ : \ \mathsf{Sub}\,\Gamma'\,\Delta'}$$

$$\frac{}{\mathsf{id} \ : \ \mathsf{Sub}\,\Gamma\,\Gamma} \qquad \frac{\sigma \ : \ \mathsf{Ty}\,\Gamma}{\uparrow\sigma \ : \ \mathsf{Sub}\,(\Gamma;\sigma)\,\Gamma}$$

$$\frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta \quad t \ : \ \mathsf{Tm}\,\Gamma\,(\sigma[\vec{t}])}{(\vec{t};t) \ : \ \mathsf{Sub}\,\Gamma\,(\Delta;\sigma)} \qquad \frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta \quad \vec{u} \ : \ \mathsf{Sub}\,B\,\Gamma}{(\vec{t} \bullet \vec{u}) \ : \ \mathsf{Sub}\,B\,\Delta}$$

We introduce some *smart constructors* (really functions) as a notational convenience. This actually makes our syntax inductive recursive and therefore not first order but the smart constructors could just be expanded to avoid this. We give only their type signatures here. The first two are one place substitution on types and terms respectively. The next allows us to apply El to a term whose type is the constant U applied to a substitution without using a coercion directly. The next one is conventional application, and a variation including substitutions which is useful for embedding neutral applications back into terms. Finally we have a weakening substitution which allows us to push

substitutions under binders more easily than using $\uparrow$ and $\varnothing$ directly.

$$\frac{\tau \;:\; \mathsf{Ty}\,(\Gamma;\sigma) \quad a \;:\; \mathsf{Tm}\Gamma\,\sigma}{\mathbf{sub}^{+}\,\tau\,a \;:\; \mathsf{Ty}\,\Gamma} \qquad \frac{t \;:\; \mathsf{Tm}\,(\Gamma;\sigma)\,\tau \quad a \;:\; \mathsf{Tm}\Gamma\,\sigma}{\mathbf{sub}\,t\,a \;:\; \mathsf{Tm}\,\Gamma\,(\mathbf{sub}^{+}\,\tau\,a)}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta \quad \sigma \;:\; \mathsf{Tm}\,\Gamma\,(\mathsf{U}[\vec{t}])}{\mathbf{El}^{\mathbf{s}}\,\sigma \;:\; \mathsf{Ty}\,\Gamma} \qquad \frac{t \;:\; \mathsf{Tm}\,\Gamma\,(\Pi\,\sigma\,\tau) \quad u \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{(t\,\$\,u) \;:\; \mathsf{Tm}\,\Gamma\,(\mathbf{sub}^{+}\,\tau\,u)}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta \quad t \;:\; \mathsf{Tm}\,\Gamma\,((\Pi\,\sigma\,\tau)[\vec{t}]) \quad u \;:\; \mathsf{Tm}\,\Gamma\,(\sigma[\vec{t}])}{(t\,\$^{\mathbf{s}}\,u) \;:\; \mathsf{Tm}\,\Gamma\,(\tau[\vec{t};u])}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta \quad \sigma \;:\; \mathsf{Ty}\,\Delta}{(\vec{t} \nearrow \sigma) \;:\; \mathsf{Sub}\,(\Gamma;\sigma[\vec{t}])\,(\Delta;\sigma)}$$

Having described the first four judgment forms we go on to consider their corresponding equality judgments which are defined mutually as dictated by the coercions. The definitions are quite long and we omit various details here. Many of the rules that make up the equality relations might be described as *boilerplate* and this has a variety of sources. Firstly there are rules for equivalence and congruences for data constructors. Then there are rules induced by the explicit substitutions interacting with the data constructors. The coercions induce coherence conditions on types, terms, and substitutions similar to those present in heterogeneous families of setoids. The remaining rules might be called computation rules and those are what we focus on.

We omit the definition of context equality altogether. It is just the least congruence on the type Con.

$$\underline{\text{data}} \quad \frac{\Gamma,\,\Delta \;:\; \mathsf{Con}}{\Gamma \simeq \Delta \;:\; \star}$$

Context equality is not always included in presentations of type theory and of its models: Hofmann [54] and Streicher [78] include it; Dybjer does not [37]; and Martin-Löf omits it from his presentations of type theory [63, 65] but includes it in the substitution calculus [64].

Type equality, and equality for terms and substitutions, can be represented homogeneously or heterogeneously in the sense of whether we equate types in the same context or different contexts. Here it is presented heterogeneously as it reduces some of the bureaucracy of dealing with coercions. For example we get only one coherence condition. Also the antisymmetry of the homogeneous

version makes it more difficult to write the normaliser. The heterogeneous character is inspired by McBride's treatment of propositional equality: "John Major Equality" [67].

Congruence rules and equivalence rules are omitted from the definition of type equality:

$$\underline{\text{data}} \quad \frac{\sigma \ : \ \mathsf{Ty}\,\Gamma \quad \sigma' \ : \ \mathsf{Ty}\,\Gamma'}{\sigma \simeq \sigma' \ : \ \star}$$

The type level coercion induces a coherence condition:

$$\frac{\sigma \ : \ \mathsf{Ty}\,\Gamma \quad p \ : \ \Gamma \simeq \Gamma'}{\mathsf{coh}\,\sigma\ p \ : \ \mathsf{coe}\,\sigma\ p \simeq \sigma}$$

Next are rules that ensure that types interact appropriately with substitutions:

$$\frac{\sigma \ : \ \mathsf{Ty}\,\Gamma}{\mathsf{rid} \ : \ \sigma[\mathsf{id}] \simeq \sigma} \qquad \frac{\sigma \ : \ \mathsf{Ty}\,\Delta \quad \vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta \quad \vec{u} \ : \ \mathsf{Sub}\,B\,\Gamma}{\mathsf{assoc} \ : \ \sigma[\vec{t}][\vec{u}] \simeq \sigma[\vec{t} \bullet \vec{u}]} \qquad \frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta}{\mathsf{U}[] \ : \ \mathsf{U}[\vec{t}] \simeq \mathsf{U}_\Gamma}$$

$$\frac{t \ : \ \mathsf{Tm}\,\Delta\,\mathsf{U} \quad \vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta}{\mathsf{El}[] \ : \ (\mathsf{El}\,t)[\vec{t}] \simeq \mathbf{El^s}\,(t[\vec{t}])} \qquad \frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta}{\Pi[] \ : \ (\Pi\,\sigma\,\tau)[\vec{t}] \simeq \Pi\,(\sigma[\vec{t}])\,(\tau[\vec{t} \nearrow \sigma])}$$

Semantic application requires projection from equations between Π-types so the following constructors are added to the definitional equality. Danielsson's simpler treatment of type equality avoids this issue.

$$\frac{p \ : \ \Pi\,\sigma\,\tau \simeq \Pi\,\sigma'\,\tau'}{\mathsf{dom}\,p \ : \ \sigma \simeq \sigma'} \qquad \frac{p \ : \ \Pi\,\sigma\,\tau \simeq \Pi\,\sigma'\,\tau'}{\mathsf{cod}\,p \ : \ \tau \simeq \tau'}$$

The term equality proceeds analogously to the type equality with rules for coherence, congruence, equivalence and substitutions which are omitted. The remaining rules are the computation rules: β, η and projection from a substitution:

$$\underline{\text{data}} \quad \frac{t \ : \ \mathsf{Tm}\,\Gamma\,\sigma \quad t' \ : \ \mathsf{Tm}\,\Gamma'\,\sigma'}{t \simeq t' \ : \ \star} \quad \underline{\text{where}}$$

$$\frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta \quad t \ : \ \mathsf{Tm}\,\Gamma\,(\sigma[\vec{t}])}{\varnothing; \ : \ \varnothing[\vec{t}; t] \simeq t} \qquad \frac{t \ : \ \mathsf{Tm}\,(\Gamma; \sigma)\,\tau}{\beta \ : \ \mathsf{ap}\,(\lambda\,t) \simeq t} \qquad \frac{t \ : \ \mathsf{Tm}\,\Gamma\,(\Pi\,\sigma\,\tau)}{\eta \ : \ \lambda\,(\mathsf{ap}\,t) \simeq t}$$

Omitting the same sets of rules for substitutions leaves the following rules:

$$\underline{\text{data}} \quad \frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta \quad \vec{t}' \;:\; \mathsf{Sub}\,\Gamma'\,\Delta'}{\vec{t} \simeq \vec{t}' \;:\; \star} \quad \underline{\text{where}} \quad \frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta}{\mathsf{lid} \;:\; \mathsf{id} \bullet \vec{t} \simeq \vec{t}}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta \quad t \;:\; \mathsf{Tm}\,\Gamma\,(\sigma[\vec{t}])}{\uparrow; \;:\; \uparrow^{\sigma} \bullet (\vec{t}; t) \simeq \vec{t}}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta \quad t \;:\; \mathsf{Tm}\,\Gamma\,(\sigma[\vec{t}]) \quad \vec{u} \;:\; \mathsf{Sub}\,B\,\Gamma}{\bullet; \;:\; (\vec{t}; t) \bullet \vec{u} \simeq (\vec{t} \bullet \vec{u}); (\mathsf{coe}\; t[\vec{u}])\;\mathsf{assoc})} \quad \frac{\sigma \;:\; \mathsf{Ty}\,\Gamma}{\uparrow\varnothing \;:\; (\uparrow^{\sigma}; \varnothing) \simeq \mathsf{id}}$$

When defining each equality relation equations between the indices could have been included. This can be avoided by defining the following operations which recover these equations:

$$\frac{\sigma \;:\; \mathsf{Ty}\,\Gamma \quad \sigma' \;:\; \mathsf{Ty}\,\Gamma' \quad p \;:\; \sigma \simeq \sigma'}{\mathbf{fog}\; p \;:\; \Gamma \simeq \Gamma'}$$

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma \quad t' \;:\; \mathsf{Tm}\,\Gamma'\,\sigma' \quad p \;:\; t \simeq t'}{\mathbf{fog}\; p \;:\; \sigma \simeq \sigma'}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta \quad \vec{t}' \;:\; \mathsf{Sub}\,\Gamma'\,\Delta' \quad p \;:\; \vec{t} \simeq \vec{t}'}{\mathbf{fog}\; p \;:\; \Gamma \simeq \Gamma'}$$

This concludes the treatment of the syntax.

5.2 Values and evaluation

Values are indexed by syntactic types and contexts. The definition is very similar to our simply typed version in the previous chapter. It is very tempting to try to index values by value contexts and value types. I tried this but it led to to a very heavily inductive-recursive definition of values where value contexts, value types, values, the partial evaluator itself and various necessary properties must be mutually defined. Following this approach seems to simplify some of the inevitable equational reasoning imposed by the coercions but it is not clear if this is an advantage when compared with the extra complexity of the definition.

The definition presented here only requires induction-recursion to provide embeddings from values to syntax. Inevitably values must appear in (and

hence be embedded into) types due to type dependency. On the other hand values are a subset of terms and given tool support to express this or, perhaps, just a different formulation we might not need the mutually defined embeddings.

First variables are defined. Then values, neutral terms and environments are defined mutually with their respective embeddings.

$$\underline{\text{data}} \quad \frac{\Gamma \;:\; \star \quad \sigma \;:\; \mathsf{Ty}\,\Gamma}{\mathsf{Var}\,\Gamma\,\sigma \;:\; \star} \quad \underline{\text{where}}$$

$$\frac{}{\varnothing \;:\; \mathsf{Var}\,(\Gamma;\sigma)\,(\sigma[\uparrow^{\sigma}])} \qquad \frac{\tau \;:\; \mathsf{Ty}\,\Gamma \quad x \;:\; \mathsf{Var}\,\Gamma\,\sigma}{x^{+\tau} \;:\; \mathsf{Var}\,(\Gamma;\tau)\,(\sigma[\uparrow^{\tau}])}$$

Variables are defined as de Bruijn indices as they are for simple types except their types must be weakened so that they are in the appropriate contexts. Their embedding operation is defined as follows:

$$\frac{x \;:\; \mathsf{Var}\,\Gamma\,\sigma}{\mathbf{emb}\,x \;:\; \mathsf{Tm}\,\Gamma\,\sigma}$$

$$\begin{aligned} \mathbf{emb} \quad \varnothing \quad &= \quad \varnothing \\ \mathbf{emb} \quad x^{+\tau} \quad &= \quad (\mathbf{emb}\,x)[\uparrow^{\tau}] \end{aligned}$$

Except for the more sophisticated treatment of types the only addition to the simply typed definitions of values and neutral terms are the coercion constructors. Environments are just sequences of values so we can easily define coercion $\mathbf{coev^s}$ (mutually with coherence $\mathbf{cohv^s}$) recursively. They do not play a role in the the actual definition of values and environments so they can be defined separately.

$$\underline{\text{data}} \quad \frac{\Gamma \;:\; \mathsf{Con} \quad \sigma \;:\; \mathsf{Ty}\,\Gamma}{\mathsf{Val}\,\Gamma\,\sigma \;:\; \star} \quad \underline{\text{where}} \quad \frac{t \;:\; \mathsf{Tm}\,(\Delta;\sigma)\,\tau \quad \vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\lambda\mathsf{v}\,t\,\vec{v} \;:\; \mathsf{Val}\,\Gamma\,((\Pi\,\sigma\,\tau)[\mathbf{emb}\,\vec{v}])}$$

$$\frac{n \;:\; \mathsf{NeV}\,\Gamma\,\sigma}{\mathsf{nev}\,n \;:\; \mathsf{Val}\,\Gamma\,\sigma} \qquad \frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma \quad p \;:\; \sigma \simeq \sigma'}{\mathsf{coev}\,v\,p \;:\; \mathsf{Val}\,\Gamma'\,\sigma'}$$

Values are either closures, neutral terms or coercions:

$$\frac{x \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathbf{emb}\,x \;:\; \mathsf{Tm}\,\Gamma\,\sigma}$$

$$\begin{aligned} \mathbf{emb} \quad (\lambda\mathsf{v}\,t\,\vec{v}) \quad &= \quad \lambda\,t[\mathbf{emb}\,\vec{v}] \\ \mathbf{emb} \quad (\mathsf{nev}\,n) \quad &= \quad \mathbf{emb}\,n \\ \mathbf{emb} \quad (\mathsf{coev}\,v\,p) \quad &= \quad \mathsf{coe}\,(\mathbf{emb}\,v)\,p \end{aligned}$$

Neutral terms are either variables, stuck applications or coercions:

$$\underline{\mathsf{data}}\quad \frac{\Gamma\ :\ \mathsf{Con}\quad \sigma\ :\ \mathsf{Ty}\,\Gamma}{\mathsf{NeV}\,\Gamma\,\sigma\ :\ \star}\quad \underline{\mathsf{where}}\quad \frac{x\ :\ \mathsf{Var}\,\Gamma\,\sigma}{\mathsf{var}\,x\ :\ \mathsf{NeV}\,\Gamma\,\sigma}$$

$$\frac{\vec{t}\ :\ \mathsf{Sub}\,\Gamma\,\Delta\quad n\ :\ \mathsf{NeV}\,\Gamma\,(\Pi\,\sigma\,\tau[\vec{t}])\quad v\ :\ \mathsf{Val}\,\Gamma\,(\sigma[\vec{t}])}{\mathsf{app}\,n\,v\ :\ \mathsf{NeV}\,\Gamma\,(\tau[\vec{t};\mathbf{emb}\ v])}$$

$$\frac{n\ :\ \mathsf{NeV}\,\Gamma\,\sigma\quad p\ :\ \sigma\simeq\sigma'}{\mathsf{coev}\,n\,p\ :\ \mathsf{NeV}\,\Gamma'\,\sigma'}$$

$$\frac{n\ :\ \mathsf{NeV}\,\Gamma\,\sigma}{\mathbf{emb}\,n\ :\ \mathsf{Tm}\,\Gamma\,\sigma}$$

$$\begin{array}{lll}\mathbf{emb} & (\mathsf{var}\,x) & = \mathbf{emb}\,x\\ \mathbf{emb} & (\mathsf{app}\,n\,v) & = \mathbf{emb}\,n\,\$^{\mathbf{s}}\,\mathbf{emb}\,v\\ \mathbf{emb} & (\mathsf{coev}\,n\,p) & = \mathsf{coe}\,(\mathbf{emb}\,n)\,p\end{array}$$

Environments are simple sequences of values:

$$\underline{\mathsf{data}}\quad \frac{\Gamma,\,\Delta\ :\ \mathsf{Con}}{\mathsf{Env}\,\Gamma\,\Delta\ :\ \star}\quad \underline{\mathsf{where}}$$

$$\frac{}{\varepsilon\ :\ \mathsf{Env}\,\Gamma\,\varepsilon}\qquad \frac{\vec{v}\ :\ \mathsf{Env}\,\Gamma\,\Delta\quad v\ :\ \mathsf{Val}\,\Gamma\,(\sigma[\mathbf{emb}\,\vec{v}])}{\vec{v};v\ :\ \mathsf{Env}\,\Gamma\,(\Delta;\sigma)}$$

$$\frac{\vec{v}\ :\ \mathsf{Env}\,\Gamma\,\Delta}{\mathbf{emb}_\Gamma\,\vec{v}\ :\ \mathsf{Sub}\,\Gamma\,\Delta}$$

$$\begin{array}{lll}\mathbf{emb}_\Gamma & (\vec{v};v) & = \mathbf{emb}_\Gamma\,\vec{v};\mathbf{emb}_\Gamma\,v\\ \mathbf{emb}_\varepsilon & \varepsilon & = \mathsf{id}_\varepsilon\\ \mathbf{emb}_{\Gamma;\sigma} & \varepsilon & = \mathbf{emb}_\Gamma\,\varepsilon\bullet\uparrow^\sigma\end{array}$$

Now we define weakening. As we are only writing an algorithm in this chapter (and not carrying out the normalisation proof) we only need to weaken values and not normal forms. Therefore we can define weakening directly without referring to renaming or OPE. Also, as we have defined the embeddings mutually with the values, every time we define an operation on values we must show mutually that the operation interacts naturally with the embeddings.

We define weakening for values, neutral terms and environments simultaneously with each other and the required properties:

$$\frac{\tau \;:\; \mathsf{Ty}\,\Gamma \quad v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathbf{wk}\,\tau\,v \;:\; \mathsf{Val}\,(\Gamma;\tau)\,(\sigma[\uparrow^\tau])} \qquad \frac{\tau \;:\; \mathsf{Ty}\,\Gamma \quad v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathbf{comwk}\,\tau\,v \;:\; (\mathbf{emb}\,v)[\uparrow^\tau] \simeq \mathbf{emb}\,(\mathbf{wk}\,\tau\,v)}$$

$$\frac{\tau \;:\; \mathsf{Ty}\,\Gamma \quad n \;:\; \mathsf{NeV}\,\Gamma\,\sigma}{\mathbf{wk}\,\tau\,n \;:\; \mathsf{NeV}\,(\Gamma;\tau)\,(\sigma[\uparrow^\tau])} \qquad \frac{\tau \;:\; \mathsf{Ty}\,\Gamma \quad n \;:\; \mathsf{NeV}\,\Gamma\,\sigma}{\mathbf{comwk}\,\tau\,n \;:\; (\mathbf{emb}\,n)[\uparrow^\tau] \simeq \mathbf{emb}\,(\mathbf{wk}\,\tau\,n)}$$

$$\frac{\tau \;:\; \mathsf{Ty}\,\Gamma \quad \vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\mathbf{wk}\,\tau\,\vec{v} \;:\; \mathsf{Env}\,(\Gamma;\tau)\,\Delta} \qquad \frac{\tau \;:\; \mathsf{Ty}\,\Gamma \quad \vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\mathbf{comwk}\,\tau\,\vec{v} \;:\; (\mathbf{emb}\,\vec{v}) \bullet \uparrow^\tau \simeq \mathbf{emb}\,(\mathbf{wk}\,\tau\,\vec{v})}$$

They are defined by induction on the structure of values, environment, and neutral terms respectively. Next we define the identity environment by induction on the structure of contexts. We start to omit the equational reasoning proofs that usually appear as arguments to coercions. They are easy to write and difficult to read, we replace them by _ in the definitions:

$$\frac{\Gamma \;:\; \mathsf{Con}}{\mathbf{vid} \;:\; \mathsf{Env}\,\Gamma\,\Gamma}$$

$$\begin{aligned}\mathbf{vid}_\varepsilon &= \varepsilon\\ \mathbf{vid}_{\Gamma;\sigma} &= \mathbf{wk}\,\sigma\,\mathbf{vid}_\Gamma;\mathsf{coev}\,(\mathsf{nev}\,(\mathsf{var}\,\varnothing))\;\_\end{aligned}$$

In the case of the identity environment we require the following property which is proved mutually with the definition of **vid** by induction on the structure of contexts:

$$\frac{\Gamma \;:\; \mathsf{Con}}{\mathbf{comvid} \;:\; \mathsf{id}_\Gamma \simeq \mathbf{emb}\,(\mathbf{vid}_\Gamma)}$$

Evaluation for terms **ev**, substitutions **ev**, and semantic application **vapp** (of value functions to value arguments) are mutually defined. The use of syntactic types and explicit coercions forces us to define evaluation mutually with a coherence property: evaluating the term in an environment and then embedding it back into the syntax must give a term definitionally equal to the

original term substituted by the environment embedded back into the syntax.

$$\frac{t \;:\; \mathsf{Tm}\;\Delta\;\sigma \qquad \vec{v} \;:\; \mathsf{Env}\;\Gamma\;\Delta}{\mathbf{ev}\;t\;\vec{v} \;:\; \mathsf{Val}\;\Gamma\;(\sigma[\mathbf{emb}\;\vec{v}])}$$

$$\begin{array}{llll}
\mathbf{ev} & (\mathsf{coe}\;t\;p) & \vec{v} & = \mathsf{coev}\;(\mathbf{ev}\;t\;(\mathsf{coev}\;\vec{v}\;_\;_))\;_ \\
\mathbf{ev} & (t[\vec{t}]) & \vec{v} & = \mathsf{coev}\;(\mathbf{ev}\;t\;(\mathbf{ev}\;\vec{t}\;\vec{v}))\;_ \\
\mathbf{ev} & \varnothing & (\vec{v};v) & = \mathsf{coev}\;v\;_ \\
\mathbf{ev} & (\lambda\;t) & \vec{v} & = \lambda\mathsf{v}\;t\;\vec{v} \\
\mathbf{ev} & (\mathsf{ap}\;t) & \vec{v} & = \mathbf{vapp}\;(\mathbf{ev}\;t\;\vec{v})\;\mathsf{refl}\;v
\end{array}$$

$$\frac{\vec{v} \;:\; \mathsf{Sub}\;\Gamma\;\Delta \qquad \vec{w} \;:\; \mathsf{Env}\;B\;\Gamma}{\mathbf{ev}\;\vec{v}\;\vec{w} \;:\; \mathsf{Env}\;B\;\Delta}$$

$$\begin{array}{llll}
\mathbf{ev} & (\mathsf{coev}\;\vec{t}\;p\;q) & \vec{v} & = \mathsf{coev}\;(\mathbf{ev}\;\vec{t}(\mathsf{coev}\;\vec{v}\;_\;_))\;_\;_ \\
\mathbf{ev} & (\vec{t}\bullet\vec{u}) & \vec{v} & = \mathbf{ev}\;\vec{t}\;(\mathbf{ev}\;\vec{u}\;\vec{v}) \\
\mathbf{ev} & \mathsf{id} & \vec{v} & = \vec{v} \\
\mathbf{ev} & \uparrow^{\sigma} & (\vec{v};v) & = \vec{v} \\
\mathbf{ev} & (\vec{t};t) & \vec{v} & = \mathbf{ev}\;\vec{t}\;\vec{v};\mathsf{coev}\;(\mathbf{ev}\;t\;\vec{v})\;_
\end{array}$$

The semantic application **vapp** has a very liberal type. It takes values whose types are definitionally equal to function types rather than values whose types are function types. This is necessary for the coercion case: The value v in this case has an arbitrary type which is equal to a function type and it cannot be show at this stage that this must be a function type so instead we accumulate the coercions.

$$\frac{\vec{t} \;:\; \mathsf{Sub}\;\Gamma\;\Delta \qquad f \;:\; \mathsf{Val}\;\Gamma'\;\rho \qquad p \;:\; \rho \simeq (\Pi\;\sigma\;\tau[\vec{t}]) \qquad a \;:\; \mathsf{Val}\;\Gamma\;(\sigma[\vec{t}])}{\mathbf{vapp}\;f\;p\;a \;:\; \mathsf{Val}\;\Gamma\;(\tau[\vec{t};\mathbf{emb}\;a])}$$

$$\begin{array}{llllll}
\mathbf{vapp} & (\lambda\mathsf{v}\;t\;\vec{v}) & p & a & = \mathsf{coev}\;(\mathbf{ev}\;t\;(\vec{v};\mathsf{coev}\;a\;_))\;_ \\
\mathbf{vapp} & (\mathsf{nev}\;n) & p & a & = \mathsf{nev}\;(\mathsf{app}\;(\mathsf{coev}\;n\;p)\;a) \\
\mathbf{vapp} & (\mathsf{coev}\;v\;p) & q & a & = \mathbf{vapp}\;v\;(\mathsf{trans}\;p\;q)\;a
\end{array}$$

The corresponding coherence properties are defined mutually. Their defini-

tions are omitted.

$$\frac{t \;:\; \mathsf{Tm}\;\Delta\;\sigma \qquad \vec{v} \;:\; \mathsf{Env}\;\Gamma\;\Delta}{\mathbf{comev}\;t\;\vec{v} \;:\; t[\mathbf{emb}\;\vec{v}] \simeq \mathbf{emb}\;(\mathbf{ev}\;t\;\vec{v})}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\;\Gamma\;\Delta \qquad \vec{v} \;:\; \mathsf{Env}\;B\;\Gamma}{\mathbf{comev}\;\vec{t}\;\vec{v} \;:\; \vec{t} \bullet (\mathbf{emb}\;\vec{v}) \simeq \mathbf{emb}\;(\mathbf{ev}\;\vec{t}\;\vec{v})}$$

$$\frac{\vec{t} \;:\; \mathsf{Sub}\;\Gamma\;\Delta \qquad f \;:\; \mathsf{Val}\;\Gamma'\;\rho \qquad p \;:\; \rho \simeq \Pi\;\sigma\;\tau[\vec{t}] \qquad v \;:\; \mathsf{Val}\;\Gamma\;(\sigma[\vec{t}])}{\mathbf{comvapp}\;f\;p\;a \;:\; \mathsf{coe}\;(\mathbf{emb}\;f)\;p\;\$^{\mathbf{s}}\;\mathbf{emb}\;v \simeq \mathbf{emb}\;(\mathbf{vapp}\;f\;p\;v)}$$

Later we define a type directed quotation operation which produces $\beta\eta$-normal forms. For this purpose we need to define value types. Weak-head normal forms are exactly what is required to perform type directed operations as they tell you what the outer constructor is. The definition of type values is quite simple. We have value versions of U and El, and Π is represented as a closure like λ.

$$\underline{\text{data}}\quad \frac{\Gamma \;:\; \mathsf{Con}}{\mathsf{VTy}\;\Gamma \;:\; \star}\quad \underline{\text{where}}\quad \frac{}{\mathsf{VU} \;:\; \mathsf{VTy}\;\Gamma}$$

$$\frac{\sigma \;:\; \mathsf{Val}\;\Gamma\;\mathsf{U}}{\mathsf{VEl}\;\sigma \;:\; \mathsf{VTy}\;\Gamma} \qquad \frac{\sigma \;:\; \mathsf{VTy}\;\Gamma \qquad \tau \;:\; \mathsf{VTy}\;(\Gamma;\sigma) \qquad \vec{v} \;:\; \mathsf{Env}\;\Gamma\;\Delta}{\mathsf{V\Pi}\;\sigma\;\tau\;\vec{v} \;:\; \mathsf{VTy}\;\Gamma}$$

$$\frac{\sigma \;:\; \mathsf{VTy}\;\Gamma}{\mathbf{emb}\;\sigma \;:\; \mathsf{Ty}\;\Gamma}$$

$$\begin{array}{llcl}
\mathbf{emb} & \mathsf{VU} & = & \mathsf{U} \\
\mathbf{emb} & (\mathsf{VEl}\;\sigma) & = & \mathsf{El}\;(\mathbf{emb}\;\sigma) \\
\mathbf{emb} & (\mathsf{V\Pi}\;\sigma\;\tau\;\vec{v}) & = & \Pi\;\sigma\;\tau[\mathbf{emb}\;\vec{v}]
\end{array}$$

Last is the definition of the evaluator for types:

$$\frac{\sigma \;:\; \mathsf{Ty}\;\Delta \qquad \vec{v} \;:\; \mathsf{Env}\;\Gamma\;\Delta}{\mathbf{ev}\;\sigma\;\vec{v} \;:\; \mathsf{VTy}\;\Gamma}$$

$$\begin{array}{llcl}
\mathbf{ev} & (\mathsf{coe}\;\sigma\;p) & \vec{v} & = & \mathbf{ev}\;\sigma\;(\mathsf{coev}\;\vec{v}\;\mathsf{refl}\;(\mathsf{sym}\;p)) \\
\mathbf{ev} & (\sigma[\vec{t}]) & \vec{v} & = & \mathbf{ev}\;\sigma\;(\mathbf{ev}\;\vec{t}\;\vec{v}) \\
\mathbf{ev} & \mathsf{U} & \vec{v} & = & \mathsf{VU} \\
\mathbf{ev} & (\mathsf{El}\;\sigma) & \vec{v} & = & \mathsf{VEl}\;(\mathsf{coev}\;(\mathbf{ev}\;\sigma\;\vec{v})\;\mathsf{U}[]) \\
\mathbf{ev} & (\Pi\;\sigma\;\tau) & \vec{v} & = & \mathsf{V\Pi}\;\sigma\;\tau\;\vec{v}
\end{array}$$

5.3 β-normal forms and β-quote

β-normal forms are defined mutually with neutral terms and their corresponding embeddings back into syntax. For β we can embed neutral terms and any type.

$$\underline{\text{data}}\quad \frac{\Gamma\ :\ \mathsf{Con}\quad \sigma\ :\ \mathsf{Ty}\ \Gamma}{\mathsf{Nf}\ \Gamma\ \sigma\ :\ \star}\quad \underline{\text{where}}\quad \frac{n\ :\ \mathsf{Nf}\ (\Gamma;\sigma)\ \sigma}{\lambda\mathsf{n}\ n\ :\ \mathsf{Nf}\ \Gamma\ \sigma}$$

$$\frac{n\ :\ \mathsf{NeN}\ \Gamma\ \sigma}{\mathsf{nen}\ n\ :\ \mathsf{Nf}\ \Gamma\ \sigma}\quad \frac{n\ :\ \mathsf{Nf}\ \Gamma\ \sigma\quad p\ :\ \sigma\simeq\sigma'}{\mathsf{ncoe}\ n\ p\ :\ \mathsf{Nf}\ \Gamma'\ \sigma}$$

$$\frac{n\ :\ \mathsf{Nf}\ \Gamma\ \sigma}{\mathbf{nemb}\ n\ :\ \mathsf{Tm}\ \Gamma\ \sigma}$$

$$\begin{aligned}
\mathbf{nemb}\quad (\lambda\mathsf{n}\ n) &= \lambda\ (\mathbf{nemb}\ n)\\
\mathbf{nemb}\quad (\mathsf{nen}\ n) &= \mathbf{nemb}\ n\\
\mathbf{nemb}\quad (\mathsf{ncoe}\ n\ p) &= \mathsf{coe}\ (\mathbf{nemb}\ n)\ p
\end{aligned}$$

$$\underline{\text{data}}\quad \frac{\Gamma\ :\ \mathsf{Con}\quad \sigma\ :\ \mathsf{Ty}\ \Gamma}{\mathsf{NeN}\ \Gamma\ \sigma\ :\ \star}\quad \underline{\text{where}}\quad \frac{x\ :\ \mathsf{Var}\ \Gamma\ \sigma}{\mathsf{nvar}\ x\ :\ \mathsf{NeN}\ \Gamma\ \sigma}$$

$$\frac{n\ :\ \mathsf{NeN}\ \Gamma\ (\Pi\ \sigma\ \tau)\quad n'\ :\ \mathsf{Nf}\ \Gamma\ \sigma}{\mathsf{napp}\ n\ n'\ :\ \mathsf{NeN}\ \Gamma\ (\tau[\mathsf{id};\mathbf{nemb}\ n[\mathsf{id}]])}\quad \frac{n\ :\ \mathsf{NeN}\ \Gamma\ \sigma\quad p\ :\ \sigma\simeq\sigma'}{\mathsf{ncoe}\ n\ p\ :\ \mathsf{NeN}\ \Gamma'\ \sigma}$$

$$\frac{n\ :\ \mathsf{NeN}\ \Gamma\ \sigma}{\mathbf{nemb}\ n\ :\ \mathsf{Tm}\ \Gamma\ \sigma}$$

$$\begin{aligned}
\mathbf{nemb}\quad (\mathsf{var}\ x) &= \mathbf{emb}\ x\\
\mathbf{nemb}\quad (\mathsf{napp}\ n\ n') &= \mathbf{nemb}\ n\ \$\ \mathbf{nemb}\ n'\\
\mathbf{nemb}\quad (\mathsf{ncoe}\ n\ p) &= \mathsf{coe}\ (\mathbf{nemb}\ n)\ p
\end{aligned}$$

We also need to be able to embed into values:

$$\frac{n\ :\ \mathsf{Nf}\ \Gamma\ \sigma}{\mathbf{vembn}\ n\ :\ \mathsf{Val}\ \Gamma\ \sigma}$$

$$\begin{aligned}
\mathbf{vembn}\quad (\lambda\mathsf{n}\ n) &= \mathsf{coev}\ (\lambda\mathsf{v}\ (\mathbf{nemb}\ n)\ \mathbf{vid})\ _\\
\mathbf{vembn}\quad (\mathsf{nen}\ n) &= \mathsf{nev}\ (\mathbf{vembn}\ n)\\
\mathbf{vembn}\quad (\mathsf{coev}\ n\ p) &= \mathsf{coev}\ (\mathbf{vembn}\ n)\ p
\end{aligned}$$

And we must show at the same time that these operations interact naturally with the embeddings directly into terms:

$$\frac{n \;:\; \mathsf{Nf}\;\Gamma\;\sigma}{\mathbf{comnemb}\;n \;:\; \mathbf{emb}\;(\mathbf{vembn}\;n) \simeq \mathbf{nemb}\;n}$$

$$\frac{n \;:\; \mathsf{NeN}\;\Gamma\;\sigma}{\mathbf{comnemb}\;n \;:\; \mathbf{emb}\;(\mathbf{vembn}\;n) \simeq \mathbf{nemb}\;n}$$

Now we are ready to define quote which is defined mutually for values **quote** and neutral terms $\overline{\mathbf{quote}}$:

$$\frac{v \;:\; \mathsf{Val}\;\Gamma\;\sigma}{\mathbf{quote}\;v \;:\; \mathsf{Nf}\;\Gamma\;\sigma}$$

$$\begin{aligned}
&\mathbf{quote}\quad(\lambda \mathsf{v}\;t\;\vec{v}) \quad = \\
&\qquad \mathsf{coev}\;(\lambda \mathsf{n}\;(\mathbf{quote}\;(\mathbf{eval}\;t\;(\mathbf{wk}\;_\;\vec{v}; \mathsf{coev}\;(\mathsf{nev}\;(\mathsf{var}\;\varnothing))\;_))))\;_\\
&\mathbf{quote}\quad(\mathsf{nen}\;n) \quad = \quad \mathsf{nen}\;(\overline{\mathbf{quote}}\;n)\\
&\mathbf{quote}\quad(\mathsf{coev}\;v\;p) \quad = \quad \mathsf{coev}\;(\mathbf{quote}\;v)\;p
\end{aligned}$$

$$\frac{v \;:\; \mathsf{NeV}\;\Gamma\;\sigma}{\overline{\mathbf{quote}}\;v \;:\; \mathsf{NeN}\;\Gamma\;\sigma}$$

$$\begin{aligned}
&\overline{\mathbf{quote}}\quad(\mathsf{var}\;x) \quad = \quad \mathsf{nvar}\;x\\
&\overline{\mathbf{quote}}\quad(\mathsf{app}\;n\;v) \quad = \quad \mathsf{coev}\;(\mathsf{napp}\;(\overline{\mathbf{quote}}\;n)\;(\mathbf{quote}\;v))\;_\\
&\overline{\mathbf{quote}}\quad(\mathsf{coev}\;n\;p) \quad = \quad \mathsf{coev}\;(\overline{\mathbf{quote}}\;n)\;p
\end{aligned}$$

5.4 $\beta\eta$-normal forms and $\beta\eta$-quote

The definition of $\beta\eta$-normal forms is defined mutually with neutral terms and again with their corresponding embeddings back into the syntax. The types of the constructors neu and nel ensure that neutral terms only appear at base type in normal forms.

$$\underline{\text{data}}\quad \frac{\Gamma \;:\; \mathsf{Con}\qquad \sigma \;:\; \mathsf{Ty}\;\Gamma}{\mathsf{Nf}\;\Gamma\;\sigma \;:\; \star}\quad \underline{\text{where}}\quad \frac{n \;:\; \mathsf{Nf}\;\Gamma\;\sigma}{\lambda \mathsf{n}\;n \;:\; \mathsf{Nf}\;\Gamma\;\sigma}$$

$$\frac{n \;:\; \mathsf{NeN}\;\Gamma\;\mathsf{U}}{\mathsf{neu}\;n \;:\; \mathsf{Nf}\;\Gamma\;\mathsf{U}} \qquad \frac{n \;:\; \mathsf{NeN}\;\Gamma\;(\mathsf{El}\;\sigma)}{\mathsf{nel}\;n \;:\; \mathsf{Nf}\;\Gamma\;(\mathsf{El}\;\sigma)} \qquad \frac{n \;:\; \mathsf{Nf}\;\Gamma\;\sigma \qquad p \;:\; \sigma \simeq \sigma'}{\mathsf{ncoe}\;n\;p \;:\; \mathsf{Nf}\;\Gamma'\;\sigma}$$

The definition of neutral terms remains the same. We have just refined where they can appear. We omit details of changes to the embeddings.

As $\beta\eta$-quotation is type directed and the values are indexed only be syntactic types we require an operation which replaces the type by the result of embedding its evaluated counterpart. As before the proof arguments to coercions are omitted and a coherence property is required.

$$\frac{v \;:\; \mathsf{Val}\;\Gamma\;\sigma}{\mathbf{rep}\;v \;:\; \mathsf{Val}\;\Gamma\;(\mathbf{emb}\;(\mathbf{eval}\;\sigma\;\mathbf{vid}))}$$

$$\begin{array}{lll} \mathbf{rep} & (\lambda\mathsf{v}\;t\;\vec{v}) & = \;\lambda\mathsf{v}\;t\;(\mathbf{eval}\;(\mathbf{emb}\;\vec{v})\;\mathbf{vid}) \\ \mathbf{rep} & (\mathsf{nev}\;n) & = \;\mathsf{nev}\;(\mathbf{nrep}\;n) \\ \mathbf{rep} & (\mathsf{coev}\;v\;p) & = \;\mathsf{coev}\;(\mathbf{rep}\;v)\;\_ \end{array}$$

$$\frac{n \;:\; \mathsf{NeV}\;\Gamma\;\sigma}{\mathbf{nrep}\;n \;:\; \mathsf{Val}\;\Gamma\;(\mathbf{emb}\;(\mathbf{eval}\;\sigma\;\mathbf{vid}))}$$

$$\begin{array}{lll} \mathbf{nrep} & (\mathsf{var}\;x) & = \;\mathsf{coev}\;(\mathsf{var}\;x)\;\_ \\ \mathbf{nrep} & (\mathsf{app}\;n\;v) & = \;\mathsf{coev}\;(\mathsf{app}\;(\mathbf{nrep}\;n)\;(\mathsf{coev}\;(\mathbf{rep}\;v)\;\_))\;\_ \\ \mathbf{nrep} & (\mathsf{coev}\;n\;p) & = \;\mathsf{coev}\;(\mathbf{rep}\;n)\;\_ \end{array}$$

$$\frac{v \;:\; \mathsf{Val}\;\Gamma\;\sigma}{\mathbf{comrep}\;v \;:\; \mathbf{emb}\;(\mathbf{rep}\;v) \simeq \mathbf{emb}\;(\mathbf{eval}\;(\mathbf{emb}\;v)\;\mathbf{vid})}$$

$$\frac{n \;:\; \mathsf{NeV}\;\Gamma\;\sigma}{\mathbf{comnrep}\;n \;:\; \mathbf{emb}\;(\mathbf{nrep}\;n) \simeq \mathbf{emb}\;(\mathbf{eval}\;(\mathbf{emb}\;n)\;\mathbf{vid})}$$

Quote for values is defined by recursion on the type and is mutual with neutral quote which is defined by recursion on the structure of neutral terms. Quote

for neutral terms is unchanged from β-quote so we omit it.

$$\frac{\sigma \;:\; \mathsf{VTy}\,\Gamma \quad v \;:\; \mathsf{Val}\,\Gamma\,(\mathbf{emb}\,\sigma)}{\mathbf{quote}\,\sigma\,v \;:\; \mathsf{Nf}\,\Gamma\,(\mathbf{emb}\,\sigma)}$$

$$\begin{array}{lllll}
\mathbf{quote} & (\mathsf{V}\Pi\,\sigma\,\tau\,\vec{v}) & f & = & \\
\multicolumn{5}{l}{\quad \mathsf{coev}\,(\lambda \mathsf{n}\,(\mathbf{quote}} \\
\multicolumn{5}{l}{\quad\quad (\mathbf{eval}\,\tau\,(\mathbf{eval}\,(\mathbf{emb}\,\vec{v})\,(\mathbf{wk}\,(\sigma[\mathbf{emb}\,\vec{v}])\,\mathbf{vid}); \mathsf{coev}\,(\mathsf{nev}\,(\mathsf{var}\,\varnothing))\,_))} \\
\multicolumn{5}{l}{\quad\quad (\mathbf{rep}\,(\mathbf{vapp}\,(\mathbf{wk}\,(\sigma[\mathbf{emb}\,\vec{v}])\,(\mathsf{coev}\,f\,_))\,\mathsf{refl}\,(\mathsf{nev}\,(\mathsf{var}\,\varnothing))))))} \\
\mathbf{quote} & \mathsf{VU} & (\mathsf{nev}\,n) & = & \mathsf{neu}\,(\mathbf{quote}\,n) \\
\mathbf{quote} & \mathsf{VU} & (\mathsf{coev}\,v\,p) & = & \mathsf{coev}\,(\mathbf{quote}\,(\mathbf{eval}\,_\,\mathbf{vid})\,(\mathbf{rep}\,v))\,_ \\
\mathbf{quote} & (\mathsf{VEl}\,\sigma) & (\mathsf{nev}\,n) & = & \mathsf{neu}\,(\mathbf{quote}\,n) \\
\mathbf{quote} & (\mathsf{VEl}\,\sigma) & (\mathsf{coev}\,v\,p) & = & \mathsf{coev}\,(\mathbf{quote}\,(\mathbf{eval}\,_\,\mathbf{vid})\,(\mathbf{rep}\,v))\,_
\end{array}$$

5.5 Normaliser

We can now define the normaliser and its coherence condition:

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathbf{nf}\,t \;:\; \mathsf{Nf}\,\Gamma\,(\mathbf{eval}\,\sigma\,\mathbf{vid})}$$

$$\mathbf{nf}\quad t \quad = \quad \mathbf{quote}\,(\mathbf{eval}\,t\,\mathbf{vid})$$

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathbf{comnf}\,t \;:\; t \simeq \mathbf{emb}\,(\mathbf{nf}\,t)}$$

Notice that the coherence property for the normaliser is the usual completeness property for normalisation and follows from the coherence properties for **eval**, **quote** and **rep**.

5.6 Extension

In this section we extend the system to include codes for Π-types in the universe. We add the following constructor to terms

$$\frac{\sigma \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{U} \quad \tau \;:\; \mathsf{Tm}\,(\Gamma; \mathsf{El}\,\sigma)\,\mathsf{U}}{\Pi\mathsf{u}\,\sigma\,\tau \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{U}}$$

and the following rule to type equality:

$$\frac{\sigma \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{U} \quad \tau \;:\; \mathsf{Tm}\,(\Gamma; \mathsf{El}\,\sigma)\,\mathsf{U}}{\Pi\mathsf{El} \;:\; \mathsf{El}\,(\Pi\mathsf{u}\,\sigma\,\tau) \simeq \Pi\,(\mathsf{El}\,\sigma)\,(\mathsf{El}\,\tau)}$$

The value type is also extended with a new constructor:

$$\frac{\sigma \ : \ \mathsf{Tm}\,\Delta\,\mathsf{U} \quad \tau \ : \ \mathsf{Tm}\,(\Delta;\mathsf{El}\,\sigma)\,\mathsf{U} \quad \vec{v} \ : \ \mathsf{Env}\,\Gamma\,\Delta}{\Pi\mathsf{uv}\,\sigma\,\tau\,\vec{v} \ : \ \mathsf{Val}\,\Gamma\,(\mathsf{U}[\mathbf{emb}\,\vec{v}])}$$

This presents a new problem with the definition of semantic application **vapp**. The is now a case $\mathbf{vapp}\,(\Pi\mathsf{uv}\,\sigma\,\tau\,\vec{v})\,p\,a \simeq ?$. The equation p has the uninhabited type $\mathsf{U} \simeq \Pi\,\sigma'\,\tau$ but the type checker does not know that it is uninhabited. For this reason we must define an eliminator for this impossible equation in the syntax and carry it through to (neutral) values and (neutral) normal forms:

$$\frac{p \ : \ \mathsf{U} \simeq \Pi\,\sigma\,\tau \quad \rho \ : \ \mathsf{Ty}\,\Gamma}{\mathsf{bot}p\,\rho \ : \ \mathsf{Tm}\,\Gamma\,\rho}$$

$$\frac{p \ : \ \mathsf{U} \simeq \Pi\,\sigma\,\tau \quad \rho \ : \ \mathsf{Ty}\,\Gamma}{\mathsf{botn}p\,\rho \ : \ \mathsf{NeV}\,\Gamma\,\rho}$$

$$\frac{p \ : \ \mathsf{U} \simeq \Pi\,\sigma\,\tau \quad \rho \ : \ \mathsf{Ty}\,\Gamma}{\mathsf{nbot}p\,\rho \ : \ \mathsf{NeN}\,\Gamma\,\rho}$$

We must also define the following new equations:

$$\frac{p \ : \ \mathsf{U} \simeq \Pi\,\sigma\,\tau \quad t \ : \ \mathsf{Tm}\,(\Gamma;\sigma)\,\tau}{\mathsf{botEl}\,p \ : \ \mathsf{El}\,(\mathsf{bot}\,p\,\mathsf{U}) \simeq \mathsf{El}\,(\mathsf{coe}\,(\lambda\,t)\,(\mathsf{sym}\,p))}$$

$$\frac{\vec{t} \ : \ \mathsf{Sub}\,\Gamma\,\Delta \quad p \ : \ \mathsf{U} \simeq \Pi\,\sigma\,\tau}{\mathsf{bot}[]\,p \ : \ \mathsf{bot}\,p\,\rho[\vec{t}] \simeq \mathsf{bot}\,p\,(\rho[\vec{t}])}$$

$$\frac{p \ : \ \mathsf{U} \simeq \Pi\,\sigma'\,\tau' \quad a \ : \ \mathsf{Tm}\,\Gamma\,\sigma'}{\mathsf{botapp}\,p \ : \ \mathsf{coe}\,(\Pi\,\sigma\,\tau)\,p\,\$\,a \simeq \mathsf{bot}\,p\,(\Pi\,\sigma'\,\tau')\,\$\,a}$$

To define quote we need need a more sophisticated treatment of elements of the type $\mathsf{El}\,\sigma$: We need a semantic decoder. First we have to be more specific about neutral codes so we adapt the definition of value types to have a constructor for only neutral codes:

$$\frac{\sigma \ : \ \mathsf{NeV}\,\Gamma\,\mathsf{U}}{\mathsf{VEl}\,\sigma \ : \ \mathsf{VTy}\,\Gamma}$$

The decoder turns coded Π-types into real ones and deals with coercions and the impossible case where the code is a λ-term. Neutral codes are passed

straight through.

$$\frac{\vec{t} \;:\; \mathsf{Sub}\,\Gamma\,\Delta \quad \sigma \;:\; \mathsf{Val}\,\Gamma'\,\rho \quad p \;:\; \rho \simeq \mathsf{U}[\vec{t}]}{\mathbf{decode}\,\sigma\,p \;:\; \mathsf{VTy}\,\Gamma}$$

$$\mathbf{decode}\quad (\Pi\mathsf{uv}\,\sigma\,\tau\,\vec{v})\quad p \;=\; \mathsf{V}\Pi\,(\mathsf{El}\,\sigma)\,(\mathsf{El}\,\tau)\,(\mathsf{coev}\,\vec{v}\;\_\;\_)$$

$$\begin{aligned}
\mathbf{decode}\quad (\lambda\mathsf{v}\,t\,\vec{v})\quad p &\;=\; \mathsf{VEl}\,(\mathsf{botn}\;_\;\mathsf{U})\\
\mathbf{decode}\quad (\mathsf{nev}\,n)\quad p &\;=\; \mathsf{VEl}\,(\mathsf{coev}\,n\;_)\\
\mathbf{decode}\quad (\mathsf{coev}\,v\,p)\quad q &\;=\; \mathbf{decode}\,v\,(\mathsf{trans}\,p\,q)
\end{aligned}$$

There is also a coherence condition which states that the decoded type is equal to the original. We must extend the evaluation, replacement and quote operations to deal with the new eliminator but these are trivial changes.

5.7 Chapter summary

In this chapter we have given a partial definition of a β-normaliser for Martin-Löf's logical framework. This is extended first to a $\beta\eta$-normaliser and then the system is extended to include a universe closed under Π-types.

Chapter 6

Conclusions

The long term goal of this work is to write a verified type checker for type theory in type theory. The chapters that make up this thesis, and the papers on which they are based, can be considered to be prototypic developments in this direction:

- In chapter 2 we give a simply typed implementation of a type checker for a dependently typed language.
- In chapters 3 and 4 we give dependently typed verified implementations of normalisers for simply typed languages.
- In chapter 5 we give a dependently typed partially verified implementation of a normaliser (the central component of a type checker) for a dependently typed language.

Future work is to give a total implementation of the normaliser in chapter 5, build a verified type checker around it, and extend the dependently typed language it implements to include inductive types.

The are a number of underlying themes that characterise the approach taken in this thesis:

$\beta\eta$-equality In type theory there is a distinction between definitional equality, that the computer can decide, and propositional equality, that can be proven. Choosing a powerful notion of definitional equality with η-rules means that more programs will type check, reducing the burden on the programmer. Agda and Epigram both implement the powerful

$\beta\eta$-equality to simplify their use by programmers and also their implementation. The meta theory lagged behind this feature for sometime and proving decidability of $\beta\eta$-equality proved to be difficult. Recent work by Abel, Coquand and Dybjer has largely filled this gap [5] but I would like to go further and give a completely formal and executable account of decidability of $\beta\eta$-equality. I would like to extend their work to include inductive types. Given that $\beta\eta$-equality represents the cutting edge of type theory in practice and in meta theory I have chosen to study it in this thesis. I have used the type directed quotation operation familiar from NBE for both simply and dependently typed λ-calculi. In the case of simply typed λ-calculus an interesting and demanding case is that of $\beta\eta$-equality for coproducts which I hope to extend my work to in the future.

Big-Step normalisation In this thesis I have developed the technique of big-step normalisation. It represents a middle ground between traditional small-step strong normalisation proofs and NBE which makes use of higher order definitions of values. In BSN we write a normaliser first as a functional program and then prove that it terminates, rather than doing both at the same time, as in NBE. This allows us to focus on the computational aspect of the algorithm separately from termination, and follow a more step-by-step approach to development. This is realised in chapter 5 where I consider a relatively small implementation which focuses on the one aspect of computation. BSN uses an environment machine to perform evaluation and a first order definition of values, this makes the Bove-Capretta technique readily applicable.

Well typed syntax In chapter 3, 4 and 5 I consider only well typed terms and take definitions of the typing judgements to be also definitions of the syntax. This has the advantage that we avoid having to prove syntactic properties such as subject reduction. In the dependently typed case choosing well typed syntax and explicit substitutions naturally brings us close to the syntax of categories with families (CwFs). This is a significant advantage as we can use the principles which govern CwFs to guide our lengthy definitions. I think this is an important point as even defining the syntax of type theory is complex and still a matter of debate, being able to take an algebraic approach to the syntax and

the data structures which are used by the typechecker is an important technical tool. I also expect that this will be vitally important when inductive types are added.

In the immediate future I hope to contribute to the topic of "internal type theory" [37], examining type theory using its own notions and internalising its syntax and semantics. In this thesis I have chosen to use well typed syntax and have been guided by the categorical notions of categories with families [37]. In the future I intend to incorporate categorical approach to inductive types and propositional equality. I intend to verify a normalisation proof for the suitably extended theory in Agda and build a verified type checker around it.

In implementations of type theory there is no fixed line between what features can be coded in type theory and what must be coded in the implementation language (Haskell for example). McBride showed in his thesis [67] how to internalise pattern matching in type theory, extended with Streicher's axiom K [55] and extendable by inductive definitions. This approach is implemented in Epigram 1 [40]. The Epigram 2 [41] prototype internalises the notion of inductive definition into the (closed) type theory. Having a formally verified core theory, such as the one I propose to verify, paves the way for investigating further internalisation of components of an implementation of type theory such as, for example, verified tactics.

There is a synergy between design and implementation in programming languages. An early test of a new language is whether it is possible to write a compiler in the new language that can compile itself. The practice of writing the compiler often feeds back into the design of the language itself, GHC [46] is a prominent example. In type theory I propose there is a third element, meta theory. Induction-recursion appeared informally in normalisation proof of Martin-Löf. It was formalised by Dybjer and Setzer [39] and is now implemented in Agda [7]. In the future I hope to contribute to this synergy by continuing the work of this thesis and verifying a type checker for type theory in type theory.

Appendix A

Formalisation of Combinatory Logic

```
module Syntax where

-- Types
data Ty : Set where
  ι : Ty
  _→_ : Ty -> Ty -> Ty

infixr 50 _→_

-- Terms
data Tm : Ty -> Set where
  K : forall {σ τ} -> Tm (σ → τ → σ)
  S : forall {σ τ ρ} -> Tm ((σ → τ → ρ) → (σ → τ) → σ → ρ)
  _$_ : forall {σ τ} -> Tm (σ → τ) -> Tm σ -> Tm τ

infixl 50 _$_

-- Definitional Equality
data _≡_ : forall {σ} -> Tm σ -> Tm σ -> Set where
  refl  : forall {σ}{t : Tm σ} -> t ≡ t
  sym   : forall {σ}{t t' : Tm σ} -> t ≡ t' -> t' ≡ t
  trans : forall {σ}{t t' t'' : Tm σ} -> t ≡ t' -> t' ≡ t'' -> t ≡ t''
```

```
  K≡    : forall {σ τ}{x : Tm σ}{y : Tm τ} -> K $ x $ y ≡ x
  S≡    : forall {σ τ ρ}{x : Tm (σ → τ → ρ)}{y : Tm (σ → τ)}{z : Tm σ} ->
          S $ x $ y $ z ≡ x $ z $ (y $ z)
  $≡    : forall {σ}{τ}{t t' : Tm (σ → τ)}{u u' : Tm σ} -> t ≡ t' -> u ≡ u' ->
          t $ u ≡ t' $ u'

-- Normal forms
data Nf : Ty -> Set where
  Kⁿ   : forall {σ τ} -> Nf (σ → τ → σ)
  Kⁿ¹  : forall {σ τ} -> Nf σ -> Nf (τ → σ)
  Sⁿ   : forall {σ τ ρ} -> Nf ((σ → τ → ρ) → (σ → τ) → σ → ρ)
  Sⁿ¹  : forall {σ τ ρ} -> Nf (σ → τ → ρ) -> Nf ((σ → τ) → σ → ρ)
  Sⁿ²  : forall {σ τ ρ} -> Nf (σ → τ → ρ) -> Nf (σ → τ) -> Nf (σ → ρ)

-- inclusion of normal forms in terms
⌜_⌝ : forall {σ} -> Nf σ -> Tm σ
⌜ Kⁿ      ⌝ = K
⌜ Kⁿ¹ x   ⌝ = K $ ⌜ x ⌝
⌜ Sⁿ      ⌝ = S
⌜ Sⁿ¹ x   ⌝ = S $ ⌜ x ⌝
⌜ Sⁿ² x y ⌝ = S $ ⌜ x ⌝ $ ⌜ y ⌝

{-#
  OPTIONS --no-termination-check #-}

module Recursive where
open import Syntax

-- Recursive normaliser
_$$_ : forall {σ τ} -> Nf (σ → τ) -> Nf σ -> Nf τ
Kⁿ      $$ x       = Kⁿ¹ x
Kⁿ¹ x   $$ y       = x
Sⁿ      $$ x       = Sⁿ¹ x
Sⁿ¹ x   $$ y       = Sⁿ² x y
Sⁿ² x y $$ z       = (x $$ z) $$ (y $$ z)
```

```
nf : {σ : Ty} -> Tm σ -> Nf σ
nf K = Kⁿ
nf S = Sⁿ
nf (t $ u) = nf t $$ nf u

module BigStep where
open import Syntax

-- Big step semantics (the graph of the recursive function)
data _$ⁿ_⇓_ : {σ τ : Ty} -> Nf (σ → τ) -> Nf σ -> Nf τ -> Set where
  rKⁿ  : {σ τ : Ty}{x : Nf σ} -> Kⁿ {σ} {τ} $ⁿ x ⇓ Kⁿ¹ x
  rKⁿ¹ : {σ τ : Ty}{x : Nf σ} -> {y : Nf τ} -> Kⁿ¹ x $ⁿ y ⇓ x
  rSⁿ  : {σ τ ρ : Ty} {x : Nf (σ → τ → ρ)} -> Sⁿ $ⁿ x ⇓ Sⁿ¹ x
  rSⁿ¹ : {σ τ ρ : Ty}{x : Nf (σ → τ → ρ)}{y : Nf (σ → τ)} ->
         Sⁿ¹ x $ⁿ y ⇓ Sⁿ² x y
  rSⁿ² : {σ τ ρ : Ty}{x : Nf (σ → τ → ρ)}{y : Nf (σ → τ)}{z : Nf σ}
         {u : Nf (τ → ρ)} -> x $ⁿ z ⇓ u -> {v : Nf τ} -> y $ⁿ z ⇓ v ->
         {w : Nf ρ} -> u $ⁿ v ⇓ w -> Sⁿ² x y $ⁿ z ⇓ w

data _⇓_ : {σ : Ty} -> Tm σ -> Nf σ -> Set where
  rK : {σ τ : Ty} -> K {σ} {τ} ⇓ Kⁿ
  rS : {σ τ ρ : Ty} -> S {σ} {τ} {ρ} ⇓ Sⁿ
  r$ : forall {σ τ}{t : Tm (σ → τ)}{f} -> t ⇓ f -> {u : Tm σ}
       {a : Nf σ} -> u ⇓ a -> {v : Nf τ} -> f $ⁿ a ⇓ v  -> (t $ u) ⇓ v

module StrongComp where
open import Utils
open import Syntax
open import BigStep

-- Strong Computability
SCN : forall {σ} -> Nf σ -> Set
SCN {ι}     n = True
SCN {σ → τ} f = forall a -> SCN a ->
  Σ (Nf τ) \n ->  (f $ⁿ a ⇓ n) ∧ SCN n ∧ (⌜ f ⌝ $ ⌜ a ⌝ ≡ ⌜ n ⌝)
```

```
prop1 : forall {σ} -> (n : Nf σ) -> SCN n
prop1 K^n        = \x sx -> sig (K^n1 x)
                            (tr rK^n (\y sy -> sig x (tr rK^n1 sx K≡)) refl)
prop1 (K^n1 x)   = \y sy -> sig x (tr rK^n1 (prop1 x) K≡)
prop1 S^n        = \x sx -> sig (S^n1 x)
                            (tr rS^n
                                (\y sy -> sig (S^n2 x y)
                                              (tr rS^n1
                                                  (\z sz ->
  let pxz = sx z sz
      pyz = sy z sz
      pxzyz = π1 (σ1 pxz) (σ0 pyz) (π1 (σ1 pyz))
  in  sig (σ0 pxzyz)
          (tr (rS^n2 (π0 (σ1 pxz)) (π0 (σ1 pyz)) (π0 (σ1 pxzyz)))
              (π1 (σ1 pxzyz))
              (trans S≡
                     (trans ($≡ (π2 (σ1 pxz)) (π2 (σ1 pyz)))
                            (π2 (σ1 pxzyz)))))) refl))
  refl)
prop1 (S^n1 x)   = \y sy -> sig (S^n2 x y) (tr rS^n1 (\z sz ->
  let sx = prop1 x
      pxz = sx z sz
      pyz = sy z sz
      pxzyz = π1 (σ1 pxz) (σ0 pyz) (π1 (σ1 pyz))
  in  sig (σ0 pxzyz)
          (tr (rS^n2 (π0 (σ1 pxz)) (π0 (σ1 pyz)) (π0 (σ1 pxzyz)))
              (π1 (σ1 pxzyz))
              (trans S≡
                     (trans ($≡ (π2 (σ1 pxz)) (π2 (σ1 pyz)))
                            (π2 (σ1 pxzyz))))))
  refl)
prop1 (S^n2 x y) = \z sz ->
  let sx = prop1 x
      sy = prop1 y
      pxz = sx z sz
      pyz = sy z sz
```

```
      pxzyz = π₁ (σ₁ pxz) (σ₀ pyz) (π₁ (σ₁ pyz))
  in  sig (σ₀ pxzyz)
          (tr (rS^n2 (π₀ (σ₁ pxz)) (π₀ (σ₁ pyz)) (π₀ (σ₁ pxzyz)))
              (π₁ (σ₁ pxzyz))
              (trans S≡
                     (trans ($≡ (π₂ (σ₁ pxz)) (π₂ (σ₁ pyz)))
                            (π₂ (σ₁ pxzyz)))))

SC : forall {σ} -> Tm σ -> Set
SC {σ} t = Σ (Nf σ) \n -> (t ⇓ n) ∧ SCN n ∧ (t ≡ ⌜ n ⌝)

prop2 : forall {σ} -> (t : Tm σ) -> SC t
prop2 K       = sig K^n (tr rK (prop1 K^n) refl)
prop2 S       = sig S^n (tr rS (prop1 S^n) refl)
prop2 (t $ u) with prop2 t          | prop2 u
prop2 (t $ u) | sig f (tr rf sf cf) | sig a (tr ra sa ca) with sf a sa
prop2 (t $ u) | sig f (tr rf sf cf) | sig a (tr ra sa ca) | sig v (tr rv sv cv)
  = sig v (tr (r$ rf ra rv) sv (trans ($≡ cf ca) cv))

module Structural where
open import Utils
open import Syntax
open import BigStep
open import StrongComp

_$$⁼_&_ : forall {σ τ}(f : Nf (σ → τ))(a : Nf σ){n} -> f $^n a ⇓ n ->
          Σ (Nf τ) \n' -> n' == n
.K^n         $$⁼ x & rK^n          = sig (K^n1 x) refl⁼
.(K^n1 x)    $$⁼ y & rK^n1 {x = x} = sig x  refl⁼
.S^n         $$⁼ x & rS^n          = sig (S^n1 x) refl⁼
.(S^n1 x)    $$⁼ y & rS^n1 {x = x} = sig (S^n2 x y) refl⁼
.(S^n2 x y) $$⁼ z & rS^n2 {x = x}{y = y} p q r with x $$⁼ z & p | y $$⁼ z & q
... | sig u refl⁼ | sig v refl⁼ = u $$⁼ v & r

nf⁼ : forall {σ}(t : Tm σ){n} -> t ⇓ n -> Σ (Nf σ) \n' -> n' == n
nf⁼ .K rK = sig K^n refl⁼
```

```
nf⁼ .S rS = sig Sⁿ refl⁼
nf⁼ .(t $ u) (r$ {t = t} p {u = u} q r) with nf⁼ t p | nf⁼ u q
... | sig f refl⁼ | sig a refl⁼ = f $$⁼ a & r

nf : forall {σ} -> Tm σ -> Nf σ
nf t = σ₀ (nf⁼ t (π₀ (σ₁ (prop2 t))))

complete : forall {σ}(t : Tm σ) -> t ≡ ⌜ nf t ⌝
complete t with nf⁼ t (π₀ (σ₁ (prop2 t)))
... | (sig ._ refl⁼) = π₂ (σ₁(prop2 t))

sound : forall {σ}{t u : Tm σ} -> t ≡ u -> nf t == nf u
sound refl        = refl⁼
sound (sym p)     = sym⁼ (sound p)
sound (trans p q) = trans⁼ (sound p) (sound q)
sound K≡          = refl⁼
sound S≡          = refl⁼
sound ($≡ p q)    = resp2 (sound p) (sound q) _$$_
```

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit Substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[2] Andreas Abel. Implementing a normalizer using sized heterogeneous types. In Connor McBride and Tarmo Uustalu, editors, *Workshop on Mathematically Structured Functional Programming, MSFP 2006, Kuressaare, Estonia, July 2, 2006*, electronic Workshop in Computing (eWiC). The British Computer Society (BCS), 2006.

[3] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. In Marcelo Fiore, editor, *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII), New Orleans, LA, USA, 11-14 April 2007*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 17–39. Elsevier, 2007.

[4] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. In *Typed Lambda Calculus and Applications*, pages 23–38, 2005.

[5] Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. In *22nd IEEE Symposium on Logic in Computer Science*, pages 3–12. IEEE Computer Society Press, 2007.

[6] Andreas Abel, Thierry Coquand, and Peter Dybjer. On the algebraic foundation of proof assistants for intuitionistic type theory. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Functional and Logic*

Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings, volume 4989 of *Lecture Notes in Computer Science*, pages 3–13. Springer-Verlag, 2008.

[7] Agda team. Agda, 2008. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.

[8] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Annual IEEE Symposium on Logic in Computer Science*, pages 412–420. IEEE Computer Society Press, 1999.

[9] Thorsten Altenkirch and James Chapman. Tait in one big step. In *Workshop on Mathematically Structured Functional Programming, MSFP 2006, Kuressaare, Estonia, July 2, 2006*, electronic Workshop in Computing (eWiC), Kuressaare, Estonia, 2006. The British Computer Society (BCS).

[10] Thorsten Altenkirch and James Chapman. Big-Step Normalisation. *Journal of Functional Programming*, 2008. Special Issue on Mathematically Structured Functional Programming. To appear.

[11] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310, 2001.

[12] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, LNCS 953, pages 182–199, 1995.

[13] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM.

[14] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, 1999.

[15] Lennart Augustsson. Cayenne—a language with dependent types. In *ACM International Conference on Functional Programming '98*. ACM, 1998.

[16] B. Barras. Verification of the interface of a small proof system in coq. In E. Gimenez and C. Paulin-Mohring, editors, *Proceedings of the 1996 Workshop on Types for Proofs and Programs*, pages 28–45, Aussois, France, December 1996. Springer-Verlag LNCS 1512.

[17] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[18] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Science Press, Los Alamitos, 1991.

[19] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[20] Ana Bove and Venanzio Capretta. Nested General Recursion and Partiality in Type Theory. In Richard Boulton and Paul Jackson, editor, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[21] Ana Bove and Peter Dybjer. Dependent types at work. February 2008. Draft notes for a graduate course on dependent types, Uraguay.

[22] Venanzio Capretta and Silvio Valentini. A general method to prove the normalization theorem for first and second order typed lambda-caluli. *Mathematical Structures in Computer Science*, 9:719–739, 1999.

[23] James Chapman. Formalisation of BSN for System T, 2007. `http://www.cs.nott.ac.uk/~jmc/BSN.html`.

[24] James Chapman. Type theory should eat itself. In A. Abel and C. Urban, editors, *LFMTP 2008: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

[25] James Chapman, Conor McBride, and Thorsten Altenkirch. Epigram reloaded: A standalone typechecker for ETT. In *Trends in Functional Programming 6*. Intellect, 2005.

[26] Catarina Coquand. A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions. *Higher Order Symbol. Comput.*, 15(1):57–90, 2002.

[27] Thierry Coquand. An algorithm for testing conversion in type theory. In *Logical frameworks*, pages 255–279. Cambridge University Press, New York, NY, USA, 1991.

[28] Thierry Coquand. Pattern Matching with Dependent Types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks (Båstad, Sweden)*, 1992.

[29] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

[30] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.

[31] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North Holland, 1958.

[32] Luis Damas and Robin Milner. Principal type-schemes for functional programming languages. In *Ninth Annual Symposium on Principles of Programming Languages (POPL) (Albuquerque, NM)*, pages 207–212. ACM, January 1982.

[33] Nils Anders Danielsson. A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, Nottingham, England, April 2006. Springer-Verlag LNCS 4502.

[34] N. G. de Bruijn. A survey of the project automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[35] N. G. de Bruijn. A plea for weaker frameworks. In *Logical frameworks*, pages 40–67. Cambridge University Press, New York, NY, USA, 1991.

[36] Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

[37] Peter Dybjer. Internal Type Theory. In *Types for Proofs and Programs, '95*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[38] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, April 1999.

[39] Peter Dybjer and Anton Setzer. Indexed induction-recursion. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001.

[40] Epigram team. Epigram, 2008. `http://www.e-pig.org`.

[41] Epigram team. Epigram 2, 2008. `http://www.e-pig.org`.

[42] R. O. Gandy. An early proof of normalisation by a. m. turing. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 453–455. Academic Press, 1980.

[43] François Garillot and Benjamin Werner. Simple Types in Type Theory: deep and shallow encodings. In *Theorem Proving in Higher Order Logics*, pages 368–382. Springer, 2007.

[44] Gerhard Gentzen. *The collected papers of Gerhard Gentzen.* North-Holland, 1969. Edited by Manfred Szabo.

[45] Neil Ghani. Beta-eta equality for coproducts. In *Proccedings of TLCA '95*, number 902 in Lecture Notes in Computer Science, pages 171–185. Springer-Verlag, 1995.

[46] GHC team. The Glasgow Haskell Compiler, 2008. `http://www.haskell.org/ghc`.

[47] J.Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These d'Etat, Paris VII, 1972.

[48] Kurt Gödel. Über eine bisher noch nicht benütze erweiterung des finiten standpunktes. *Dialectica*, 12:280–287, 1958.

[49] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, LFCS, 1994. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/94/ECS-LFCS-94-304/`.

[50] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87, Ithaca, NY, USA, 22–25 June 1987*, pages 194–204. IEEE Computer Society Press, New York, 1987.

[51] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 2007. (To appear.).

[52] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 6:61–101, January 2005.

[53] Haskell libraries team. HackageDB, 2008. `http://hackaage.haskell.org`.

[54] Martin Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, volume 14, pages 79–130. Cambridge University Press, Cambridge, 1997.

[55] Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *Proc. Ninth Annual Symposium on Logic in Computer Science (LICS) (Paris, France)*, pages 208–212. IEEE Computer Society Press, 1994.

[56] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[57] Gerard Huet. The constructive engine. In *A Perspective in Theoretical Computer Science*. World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

[58] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[59] Zhaohui Luo. *ECC: An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/90/ECS-LFCS-90-118/`.

[60] Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, LFCS, 1992.

[61] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, LNCS 806. Springer-Verlag, 1994. Selected papers from the Int. Workshop TYPES '93, Nijmegen, May 1993.

[62] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the 3rd Scandinavian Logic Symposium*, pages 81–109, 1975.

[63] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

[64] Per Martin-Löf. Substitution calculus, 1992. Notes from a lecture given in Göteborg.

[65] Per Martin-Löf. An intuitionistic theory of types. In G. Sambin and J. Smith, editors, *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*. Oxford University Press, 1998.

[66] Ralph Matthes. *Extentions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Ludwig-Maximilians University, Munich, 1998.

[67] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.

[68] Conor McBride. Epigram, 2005. `http://www.e-pig.org`.

[69] Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer-Verlag, 2005. Revised lecture notes from the International Summer School in Tartu, Estonia.

[70] Conor McBride, Healfdene Goguen, and James McKinna. A Few Constructions on Constructors. In *Types for Proofs and Programs, Paris, 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2005. accepted; to appear.

[71] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

[72] Ulf Norell. Dependently type programming in agda, 2008. Draft notes from AFP 2008.

[73] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM Press.

[74] Frank Pfenning and Carsten Schürmann. Twelf user's guide, version 1.4. 2002.

[75] Robert Pollack. *The Theory of LEGO.* PhD thesis, University of Edinburgh, 1995. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/95/ECS-LFCS-95-323/`.

[76] Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory.* Oxford Univ. Press, 1998.

[77] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathetische Annelen*, 92:305–316, 1924.

[78] Thomas Streicher. *Semantics of type theory: correctness, completeness, and independenc results.* Birkhauser Boston Inc., Cambridge, MA, USA, 1991.

[79] William W. Tait. Infinitely long terms of transfinite type. In J. N. Crossely and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*. North Holland, 1965.

[80] William W. Tait. Intensional interpretations of functionals of finite type. *Journal of Symbolic Logic*, 32:198–212, 1967.

[81] The Haskell Team. Haskell website. `http://www.haskell.org`.

[82] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction (vol I)*. Elsevier, 1988.

[83] A. M. Turing. Some theorems about church's system. unpublished.

[84] Christian Urban, James Cheney, and Stefan Berghofer. Mechanizing the metatheory of lf. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 45–56. IEEE Computer Society Press, 2008.

[85] Silvio Valentini. A note on a straightforward proof of normal form theorem for simply typed lambda-calculi. *Bollettino dell'Unione Matematica Italiana*, 8:207–213, 1994.

[86] Femke van Raamsdonk. *Confluence and Normalisation for Higher-Order Rewriting*. PhD thesis, Vrije University, Amsterdam, 1996.

[87] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A Concurrent Logical Framework: The Propositional Fragment. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, pages 355–377, Torino, Italy, 30 April – 4 May 2003. Springer-Verlag LNCS 3085.