University of Bergen
Department of informatics

λ' is Confluent

Author: Yan Passeniouk
Supervisor: Håkon Gylterud

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

November, 2023

Abstract

The λ-calculus is a well-known model of computation, characterised by its simplicity and adapted for the implementation of functional programming languages. We present an extension of the λ-calculus proposed by Gylterud, λ'-calculus, with primitive quotation operations to allow for internalised self-interpretation, a form of metaprogramming. Using the proof assistant Agda, we give a formalisation of its syntax and operational semantics and prove confluence, a property which gives any reducible term a unique normal form.

Acknowledgements

I would first like to express my deep gratitude to my supervisor, Håkon Gylterud, for everything he has done to make this thesis possible, including, but not limited to: being incredibly generous with his time, sharing his vast amount of knowledge and staying positive and encouraging whilst showing great patience.

I'm also thankful for the rest of the Programming Languages Group, for answers to my questions, planned and unplanned talks, companionable lunches and the lending of books I was unable to find otherwise.

I was also fortunate enough to go through multiple iterations of the Jafu reading hall, giving me enjoyable distractions and opportunities to play some small part in their problem-solving processes.

Finally, and most of all, I would like to thank my mother, for her steadfast love and support.

Yan Passeniouk
19th of October, 2023

Contents

Chapter 1

Introduction

1.1 Background

Metaprogramming is the concept of a program which takes other programs as input to transform or analyse them, extracting properties, enforcing invariants and possibly generating other programs[15]. Metaprogramming can be implemented in various ways, including templates [2], dependent types[14] (a version of which, Martin-Löf Type Theory, is described below) and preprocessing [12].

The technique of metaprogramming we consider is self-representation, which encodes a version of the language using the language itself, exemplified by quotation in the LISP-family of languages ([1], Ch. 2). quotation in LISP-languages was an inspiration behind the subject of this thesis, the λ'-calculus, as in these languages the user is unable to reduce expressions which should be equivalent due to a failure of confluence, e.g. `((lambda (x) (eval 'x)) 3)` will return `3`, while the β-equivalent expression `(((lambda (y) (lambda (x) y)) (eval 'x)) 3)` will not.

To remedy this, Gylterud ([10]) introduced an extended λ-calculus with a primitive notion of quotation encoded as an internal operation, in addition to as function abstraction and application.

This thesis is structured as follows: An introduction which presents the meta-theory we will be working in along with previous work in the area of self-quotation; a chapter on λ-calculus, giving an overview of the topics required for a presentation of λ'-calculus along with a motivated explanation of its intended purpose; with the next chapter giving a guided tour through the formalisation of our proof of confluence, using the Agda proof assistant and Unimath library and finally a conclusion with some prospects for the future of λ'.

1.2 Previous work

The encoding of self-interpretation/quotation in λ-calculus has been studied continuously for at least 30 years, though the general concept of quotation stems back at least as far as 1936 [13]. All methods require some form of encoding of the λ-calculus as terms in the λ-calculus, usually an encoding of it into terms with a certain structure (usually $\mathbb{N}$ or Combinatory logic) and a specific decoding term (defined in/internal to the λ-calculus) which transforms it back into the encoded term. The earliest encoding which allows for a term to be successfully evaluated back into the top level calculus was Mogensen's approach [19] defines the encoding as a metatheoretic function on terms using higher order abstract syntax.

More recent approaches, such as that of Brown and Palsberg [5] (one of the inspirations behind λ') extend the quotation mechanism into a typed λ-calculus (System Fω, with naturals ω and a fixed point operator) which they showed to be normalisable, using a pre-representation of both types and terms along with a Church-based encoding (explained in the following chapter) of these terms.

Jay [11] gives an intensional λ-calculus which is confluent and strongly normalising using a number of standard and non-standard combinators (in the sense of combinatory logic), to encode the reduction relation and internal quotaion, keeping the calculus and resulting type theory on a single "level".

The approach of λ'-calculus is most similar to Mogensen's approach, but our inclusion of the internal quotation operator allows for multiple levels of quotation to be represented internally, whereas Mogensen's construction requires the syntax of the internal representation to be represented metatheoretically. We avoid the complications of the large hierarchy of inference rules and presentations involved with Brown and Palsberg's approach.

1.3 Martin-Lof Type Theory

We formalise our proof of confluence in the proof assistant Agda [9], which implements a version of Martin-Lof Type Theory (MLTT), meaning we are using it as our metatheory. It is a dependent type theory, descended from sequent calculi in the tradition of Gentzen, meaning logical deductions are produced through inference rules connecting atomic statements named *judgements*.

The interested reader may consult [21] (which we base the following exposition on) for further details, but we will give a brief and incomplete summary of what induction rules constitute a type, along with a short mention of *The Curry-Howard Isomorphism* which gives

an interpretation of proofs as programs using types to represent basic logical propositions (specifically of intuitionistic logic).

The first judgement is that of a context Γ ctx, a list of terms and types which might be empty (then called the empty context). A type A is judged to be a type for some context by stating $\Gamma \vdash A$ Type, A type may have terms which inhabit it, which are said to be *of* that type, e.g. $\Gamma \vdash a : A$ is a term a of type A in the context Γ.

Notice how both of these are defined with a context, which encodes the dependency of the right hand side of $\vdash$ on the left hand side of $\vdash$. Terms and types can also be judged to be definitionally equal, $\Gamma \vdash A \doteq B$, $\Gamma \vdash a \doteq a' : A$, which differs from the intensional equality defined below using the Identity type, in that it carries no proof relevant information i.e. it is simply asserted, allowing us to substitute terms and types freely across the definitional equality as one would with any equivalence relation.

The context dependency of MLTT-types allows us to define *type families* , families of types dependent on variables of other types, or perhaps more intuitively, "functions into the type level", which are denoted as (for a fixed type A) $\Gamma, a : A \vdash B(a) :$ Type and formed using the inference rule: Which is read as the type B(a) being *indexed* by terms of type A (or more accurately, the family defines sections of the type B(a)).

This in turn allows us to define the type of dependent functions. The non-dependent version is often seen in functional languages such as Haskell as `(->)`, the function type. We start by giving a *formation rule*, the rule to form the type in some context Γ:

$$\Pi\text{-Form}\quad \frac{\Gamma, a : A \vdash B(a) \text{ Type}}{\Gamma \vdash \Pi_{a:A} B(a) \text{ Type}}$$

Which can be read as "Given a type family B indexed by terms of A, we can produce a type $\Pi_{x:A} B(x)$". A corresponding *introduction rule* allows us to produce terms of the Π-type, which correspond to functions where the the type of the output may depend on the term of the input. So the introduction rule for the dependent function type reads:

$$\Pi\text{-Intro}\quad \frac{\Gamma, a : A \vdash b(a) : B \text{ Type}}{\Gamma \vdash \lambda a.\ b(a) : \Pi_{a:A}\ B(a)}$$

Which we can informally express as "Given a type family B(a) indexed by A, we can produce a term $\lambda a.b(a)$ of type $\Pi_{a:A}\ B(a)$", allowing the type family to be encoded as a function on

the term/value level. To produce a term of the type we parametrise , we can use *elimination rules*, which state the requirements to do so. The function type has the following elimination rule:

$$\text{Π-Elim}\quad \frac{\Gamma \vdash f : \Pi_{a:A} B(a)}{\Gamma, a : A \vdash f(a) : B}$$

Which reads "Given a term of the function type from A to B, we can produce a term f(a) of type B indexed by terms a : A", intuitively evaluating the dependent function at the value (a : A). elimination rules are also referred to as "induction principles", as they have the same expressive power as structural induction for the structure of the type in question. However, this does not ensure that the relevant definitional equalities, which is done by *computation rules*,

$$\eta\quad \frac{\Gamma \vdash b : \Pi_{a:A}\ B(a)}{\Gamma \vdash \lambda a.\ b(a) \doteq b : \Pi_{a:A} B(a)} \qquad \beta\quad \frac{\Gamma,\ a : A \vdash b(a) : B(a)}{\Gamma, a : A \vdash b(a) \doteq (\lambda\ x.b(x))(a) : B(x)}$$

Stipulating a given term of dependent function $\Pi_{x:A} B(a)$is definitionally equal to a λ-term applying the dependent function (η-expansion), and that given a type family $B(a)$ indexed by type A, the term of the corresponding dependent function type is definitionally equal to a term of the type family (β-reduction).

If two types A, B are in the same context, we can reconstruct the regular function type as a special case of the dependent using *weakening* on the level of judgements (and the notation := to mean "defined as", which we use as a renaming without asserting a definitional equality):

$$\frac{\dfrac{\Gamma \vdash A\ \text{Type} \qquad \Gamma \vdash B\ \text{Type}}{\Gamma, x : A \vdash B\ \ \text{Type}}}{\Gamma \vdash A \to B := \Pi_{x:A}\ B}$$

and with corresponding rules still holding, while B not being dependent on variables of type A.

The dependent function type is a part of the encoding often refered to as the Curry-Howard Correspondence (or Isomorphism), which stipulates a connection between types in a programming language and logical formulae (specifically, intuitionistic logic, which lacks the axiom of excluded middle). We summarise the correspondence in the table below:

Logical relation	Type	Logical Symbol	Type Signature
Truth	Unit type	$\top$	$\mathbb{1}$
Falsity	Empty type	$\bot$	$\mathbb{0}$
Negation	Function to empty type	$\neg A$	$A \to \mathbb{0}$
Universal quantification	Dependent function	$\forall a.B(a)$	$\Pi_{x:A}B(a)$
Existential quantification	Dependent pair	$\exists a.B(a)$	$\Sigma_{x:A}B(a)$
Conjunction	Product type	$A \wedge B$	$A \times B$
Disjunction	Coproduct type	$A \vee B$	$A + B$
Implication	Function type	$A \to B$	$A \to B$
Equality	Identity type	$a = b$	Id_Aa b

With the proviso that the typed representations of logical operations encode for proof terms of their constituents, e.g. a term of the (cartesian) product type $(a, b) : A \times B$ is read as "A term providing a proof a type A and proof b of type B", instead of $A \wedge B$ being read as the claim "A and B are true". Each of the types having introduction, formation, elimination and (possibly) computation rules.

One of the key properties of MLTT is the intensional identity type, which through a simple set of inference rules gives intensional MLTT the ability to express mathematical notions as abstract as homotopy theory by permitting reasoning about identities as terms of a type.

The formation and introduction rules for the identity type are given as:

$$\textsc{Id-Form}\quad \frac{\Gamma \vdash a : A}{\Gamma, x : A \vdash Id_A\ a\ x} \qquad\qquad \textsc{Id-Intro}\quad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a\ :\ \mathrm{Id}_A\ a\ a}$$

The formation rule indexes the identity type for a type A by all terms (a : A), while the introduction rule establishes a type family witnessing the reflexivity of a each variable (a : A) with the term $\mathsf{refl_a}$. This, along with the elimination and computation rules:

$$\textsc{Id-Elim}\quad \frac{\Gamma \vdash a : A \qquad \Gamma,\ x : A,\ p : \mathrm{Id}_A\ a\ x \vdash P(a, p) :\ \mathrm{Type}}{\Gamma \vdash \text{ind-eq}_a : P(x, \mathsf{refl_a}) \to \Pi_{x:A}\Pi_{p:Id\ a\ x}P(x, p)}$$

$$\textsc{Id-Comp}\quad \frac{\Gamma \vdash a : A \qquad \Gamma, x : A,\ p : \mathrm{Id}_A\ a\ x \vdash P(x, p) : \mathrm{Type}}{\Gamma, u : P(a, \mathsf{refl_a}) \vdash \text{ind-eq}_a(u, a, \mathsf{refl_a}) \doteq u : P(a, \mathsf{refl_a})}$$

allows us to generate terms parametrised by the identity type (and consequently, each term of the type A it is an identity for), with the elimination rule/induction principle assuming a term a : A and some type family P(x, p) indexed by a term (a : A) and the corresponding term of the identity type and generating dependent function type corresponding to that family, with a natural definitional equality being stipulated by the computation rule.

These fairly simple rules allow us to define and give terms representing everything from identities between terms of any other type, to the transitivity and symmetry of the identity type itself, permitting the construction of equational proofs as types by encoding the assumptions and identities and providing terms for through applications of the properties and other identities, as will be exemplified in the chapter on formalising and proving confluence.

Chapter 2

The λ'-calculus

The λ-calculus was first introduced in 1935 by Alonzo Church as a logical system for the study of partial recursive functions and with the intention of capturing the essence of function application and composition as used in other areas of mathematics. Alan Turing used the λ-calculus to define the expressive power of his eponymous machines [24] (giving Turing completeness in terms of λ-definability, as the λ-calculus preceeded Turing machines), and so has been adapted as a suitable model of computation for functional programming languages.

2.1 Definitions and Intuitions

The λ-calculus distinguishes itself from other models of computation (Turing machines etc.) by its remarkable simplicity, as seen from its definition (based on [23], as with the following definitions, rewritten into inference rules to conform better with the formalisation[1]). Given a set X, we define ($\setminus$ denoting set difference):

$$\frac{x : X}{x : \Lambda X} \qquad \frac{x : \Lambda X \qquad y : \Lambda X}{(xy) : \Lambda X} \qquad \frac{x : X \qquad M : \Lambda X}{(\lambda\, x.M) : \Lambda(X \setminus \{x\})}$$

Interpreting these rules, terms of λ-calculus are generated by letting all elements of the set X be considered variables and applying two operations, function application (ab) and its left inverse λ-abstraction $\lambda a.a$, which "binds" a variable, removing it from the set of free variables. Parantheses denote the order in which application is performed, e.g. one may

[1] And avoid the issue of which set theory is being employed

read $\lambda a.a(ab)$ as a function taking a function a, applying it to b, then applying a again to the result of the previous application. We call the terms above the line (in the antecedent) the *subterms* of the terms below it (in the consequent). The set X is the set of free variables, with the binding discussed above removing the bound variables from the set.

Substitution into terms of the λ-calculus is defined as a function $\_[\_ := \_] : \Lambda\ X \to X \to \Lambda X \to \Lambda\ X$ in the λ-calculus, as long as we respect the structure of the terms we substitute into:

$$\begin{aligned} x[x := N] &\equiv N \\ y[x := N] &\equiv y \text{ when } x \neq y \\ (M_1 M_2)[x := N] &\equiv (M_1[x := N])(M_2[x := N]) \\ \lambda y.M[x := N] &\equiv \lambda y.M[x := N] \end{aligned}$$

The first two clauses of the definition simply state that substituting into a term consisting of 1 variable will yield the term we are substituting if it is the variable we are substituting on. The third clause propagates the substitution down both sides of an application, and the final clause prevents bound variables from being captured by the substitution.

Substitution raises the issue of "substitution capture", as a substitution might result in an already bound variable being substituted into the term, e.g. $(\lambda y.yx)[x := y] \equiv \lambda y.yy$. We avoid this by specifically removing bound variables from the set of free variables, and by the type signature given, the variable we are substituting will never be in the set of free variables.

In the formalisation we handle the question of substitution capture and equivalence of terms with a different technique, De Bruijn indices, explained below.

Defining substitution gives the main computational operation of the λ-calculus, β-reduction:

$$\frac{\beta}{\dfrac{N : \Lambda X \qquad M : \Lambda(X \setminus \{x\})}{((\lambda\ x.\ N)M) \rightsquigarrow N\ [x := M]}}$$

Intuitively, computation is performed by substitution into every application for which the left argument is a λ-abstraction. We call subterms of the form $(\lambda x.N)M)$ redexes. From β-reduction, we define the $\rightsquigarrow$ relation over terms $\rightsquigarrow$, using the rules for generating terms to locate redexes as subterms and relating two terms if they the right term β-reduces to

the left, which we interpret as one "step" of a computation, giving what's referred to as the "operational" semantics of λ-calculus.

In the case of two terms "operating" in the same way, i.e. defining the same function, such as the terms $\lambda a\ b.a$ and $\lambda x\ y.\ x$, being present in the same term, we would like to avoid name collisions and so require the λ-calculus to be closed under α-equivalence, so $\lambda a\ b.a \equiv \lambda x\ y.\ x$.

If we wish to consider more steps of the computation, we define a transitive and reflexive closure of the reduction relation $\rightsquigarrow^*$.

$$x \rightsquigarrow^* x \qquad\qquad \frac{x \rightsquigarrow y \qquad y \rightsquigarrow^* z}{x \rightsquigarrow^* z}$$

We can interpret this reflexive and transitive closure as the "eventual outcome" of the computation of the term or the application of the reduction relation 0 or more times.

We refer to a term which cannot be β-reduced further as "normal", which we interpret as a terminated computation, Some terms do not normalise, e.g.

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &\rightsquigarrow (\lambda x.x\ x)(\lambda x.x\ x)\\ &\rightsquigarrow (\lambda x.x\ x)(\lambda x.x\ x)\\ &\rightsquigarrow \dots\end{aligned}$$

which is interpreted as a non-terminating computation.

We will use standard shorthand for representing terms, with $\lambda x.\lambda y.\lambda z.xyz$ written as $\lambda xyz.xyz$, where the λ associates to the right, and parentheses being omitted when application terms are left-associative i.e. $(((MN)O)P) = MNOP$. We also use upper-case (M,N) to refer to terms and lower-case to refer to variables (x, y, z).

2.2 Encodings and other constructions

Since there are no primitives for primitive recursive or general recursion in λ-calculus, looping is performed by recursive calls in the term e.g. $\lambda\ y.y\ y$ will capture a y and apply y to itself, while the term $(\lambda\ y.yy)(\lambda\ y.yy)$ reduces as shown before, never reaching a normal form.

This is the prototypical example of a non-terminating computation, meaning it will never terminate, and an illustrative example of the "reduction" relation not reducing the length of a term. Non-termination of terms is also dependent on the reduction strategy employed,

e.g. the term $(\lambda xy.\ y)\ (\Omega)$ where $\Omega = (\lambda\ x.x\ x)(\lambda\ x.x\ x)$, which will not terminate if the Ω reductions are performed, while giving the identity $\lambda y.y$ if the leftmost reduction is chosen.

Using a *fixed point combinator*, which are terms fulfilling the property $Fix\ f \rightsquigarrow^* f(Fix\ f)$, such as $Y := \lambda\ f.(\lambda x.f(xx))(\lambda x.f(xx))$. We perform general recursion with any other closed term by applying Y (or any other fixed point combinator, as many have been discovered) to it.

With these basic notions, we can begin to define terms which correspond to already familiar constructions in mathematics as datatypes in λ-calculus. The terms $T := \lambda xy.x$ and $F := \lambda xy.y$ can be seen as Booleans, either choosing x (True) or y (False). This is done without mention/typing the terms, as the λ-calculus is entirely syntactic, and has no inherent typing information (or, one might consider it to be unityped). We can define other boolean functions as terms, taking the terms above as arguments, such as $not := \lambda b.b\ F\ T$.

$$\begin{aligned} not\ T &\rightsquigarrow (\lambda b.b\ F\ T)T \\ &\rightsquigarrow T\ F\ T \\ &\rightsquigarrow (\lambda xy.x)\ F\ T \\ &\rightsquigarrow (\lambda y.F)\ T \\ &\rightsquigarrow F \end{aligned}$$

along with $and := \lambda ab.abF$, $or := \lambda ab.aTb$ and so on.

We can perform a case split, equivalent to an `if-then-else` statement (or implication $\rightarrow$) with a term $\lambda t\ x\ y.t\ x\ y$, where t is a predicate (a term reducing to some boolean) applied to x and y. We can also define the natural numbers in a similar fashion, due to a construction by Church: Define $SUC := \lambda n\ f\ s.f\ (n\ f\ s)$, the successor function $(+1)$ then the numerals are given by $0 = \lambda\ f\ x.x$, and repeated application of SUC, or more explicitly:

$$\begin{aligned} 0 &:= \lambda fx.\ x \\ 1 &:= \lambda fx.(f\ x) \\ 2 &:= \lambda fx.f\ (f\ x) \end{aligned}$$

and so on. The issue with this encoding (which Church encountered) is the definition of a predecessor function, which (apocryphally) eluded him until his student Kleene came with a solution, which still took n steps to reduce.

Instead, we define both of the above structures to be more reminiscent of inductive datatypes in a functional language like Haskell [17]. Intuitively, we bind a variable for each of the constructors in a datatype, and then bind a variable for each of the cases with the variable

bound "decorating" the term and representing the constructors. For any recursively defined type, we add a recursive call in the form of calling the outermost (parameter) variables in the head, as these are part of the datatype we are defining. The natural numbers can then be defined by two constructors,

$$\texttt{Z} := \lambda sz.\ z$$
$$\texttt{SUC} := \lambda nsz.s\ n\ (n\ s\ z)$$

Z encoding the nullary constructor for 0, only envoking the z as with the Church encoding above and SUC encoding the unary constructor of successor, applying s (the successor case) to the parameter (natural number to be suceeded) and passing both the zero and successor cases to the previous number. This allows us to give a fairly simple encoding of the predecessor function, $\lambda t.t\ 0\ (\lambda x\ y.x)$. This notion of encoding datatypes coincides with the previous one for the booleans, as both of the constructors are nullary and hence do not require any recursive calls. This encoding seems to have its origins with Scott and was further extended by Mogensen [19] (for the definition of a notion of quotation supporting self-evaluation and self-reduction) as mentioned in Section 1.2, though was arrived at independently by Gylterud[10].

2.2.1 De Bruijn Representation

The De Bruijn representation of the lambda calculus removes the requirement for renaming, and therefore α-equivalence from λ-calculus. Intuitively, we do this by explicitly writing a λ for each bound variable and referring to the bound variables in the body of the term by the position of the variable in the list of λ's. Formally, we define the terms of De Bruijn representation analoguously to the definition of terms above, which we chose to index from the right:

$$\lambda x.x \mapsto \lambda.0$$
$$\lambda xy.y \mapsto \lambda\lambda.0$$
$$\lambda xyz.xz(yz) \mapsto \lambda\lambda\lambda(20)(10)$$
$$(\lambda yz.xz)(\lambda ab.a) \mapsto (\lambda\lambda(20))(\lambda\lambda.0)$$
$$\lambda fx.fx \mapsto \lambda\lambda 10$$
$$\lambda fx.ffx \mapsto \lambda\lambda 110$$

We give a type-theoretically flavoured definition (inspired by [6]) of De Bruijn indices, for a fixed type of free variables X, + denoting the coproduct type and 1 the unit type:

$$\frac{x : X}{x : \Lambda X} \qquad \frac{M : \Lambda X \qquad N : \Lambda X}{(MN) : \Lambda X} \qquad \frac{M : \Lambda(X+1)}{\lambda M : \Lambda X}$$

We interpret this as the singleton set being a single variable to be bound, where λ acts as a symbol pointing to this variable, the coproduct encoding the choice of the free variable, whereas the two other constructors do not require any variable to be free, and so are left identical to the definition above and interpreted to have "some" amount of free variables. Using De Bruijn indices, we have a representation of λ-terms which is α-equivalent for all terms (equivalent up to variable relabling). The definition above will serve as motivation for the formalisation of λ''-calculus in the following chapter, along with the more informal definition in section below.

Substitution for de Bruijn indices will have the same signature as substitution for the set-based λ calculus, and is expanded on in the next chapter, as it requires some machinery which is cumbersome to define using induction rules.

Substitution allowing us to define β-reduction similarly to above:

$$\frac{\begin{array}{l}\beta \\ N : \Lambda\ (X+1) \qquad M : \Lambda\ X\end{array}}{((\lambda\ N)\ M) \rightsquigarrow N\ [x := M]}$$

2.3 Properties of λ'-calculus

The λ-calculus may be extended beyond λ, () and β, with constructions such as products, reduction rules (such as η-extension) or even combinators, none of which will be necessary for λ'-calculus to encode our notion of quotation.

Instead, λ'-calculus is extended by a term constructor ($'$), which binds a finite list of n variables similarly to the λ-constructor, with all variables considered bound and therefore removed from the set of free variables. To avoid writing an extended list of variables, we instead use the notation $v : X^n$ to denote a length n list of free variables, along with $\{v_i\}_{i \leq n}$ as the (flattened) family generated by the vector and $\setminus$ denoting set difference.

$$\frac{}{x : \Lambda\ X}\ \textsc{Var} \qquad \frac{x : X \qquad M : \Lambda\ X}{\lambda x.\ M : \Lambda\ \{X - \{x\}\}}\ \textsc{Abs} \qquad \frac{x : \Lambda\ X \qquad y : \Lambda\ X}{(xy) : \Lambda\ X}\ \textsc{App} \qquad \frac{v : X^n \qquad M : \Lambda\ X}{\langle v \rangle' M : \Lambda\ (X \setminus \{v_i\}_{i<n})}\ {}'/\textsc{Quot}$$

Harkening back to the discussion of encodings above, we observe that even the λ-calculus itself can be encoded as terms, We want to use this constructor to denote quoted functions with the added property of being able to β-reduce under the quote, along with the quotation operator:

Definition 2.3.1 (Internal representation of λ-calculus)**.**

$$\begin{aligned}
\texttt{var} &:= \lambda x\ c_v\ c_l\ c_a\ c_q.c_v x \\
\texttt{lam} &:= \lambda t\ c_v\ c_l\ c_a\ c_q.c_l\ t\ (t\ c_v\ c_l\ c_a\ c_q) \\
\texttt{app} &:= \lambda t\ u\ c_v\ c_l\ c_a\ c_q.c_a\ t\ u\ (t\ c_v\ c_l\ c_a\ c_q)(u\ c_v\ c_l\ c_a\ c_q) \\
\texttt{quot} &:= \lambda n\ t\ c_v\ c_l\ c_a\ c_q.c_q\ n\ t\ (t\ c_v\ c_l\ c_a\ c_q)
\end{aligned}$$

Each of the c-variables represent the "constructors" of the λ'-calculus, c_v for variables, c_l for λ-abstraction and so on. The variable term `var` takes a value as input under the variable constructor, terminating the induction at some variable (encoded in de Bruijn indices as a natural number). The internal abstraction `lam`, as with the external λ, only accepts other terms as input, and as terms are generated inductively, we pass down each of the c-constructors. Application `app` binds two variables and performs the recursive subterm generation in the same way as the `lam` term. `quot` is the only constructor taking something other than the type of free variables and other terms of the internal λ-calculus, specifically a natural number, to conveniently define the list of quoted variables of a term as the cardinality of the list (or the highest de Bruijn index).

We need an axiomatised version of head normal form to express the reduction rules below, specifically the quoted application and double quotation, to allow for β-reduction to be performed under the quote. The shape of a head normal form is given by $\lambda x_1\ x_2\ \ldots\ x_n.y\ M_1\ M_2\ \ldots\ M_m$, the "head" in head normal refering to the variable being in the first position. We use a modified head normal form, which we refer to as either an abstractionless or λ-less head normal form, so we identify terms of the form $y\ M_1\ M_2\ \ldots\ M_m$ or $\langle x_1, x_2 \ldots \rangle' y\ M_1\ M_2\ \ldots\ M_m$.

We express head normal form, parametrising head normal$_y$ with the head variable y:

$$\text{HNF-Var}\ \frac{y : X}{\text{head normal}_y(y)} \qquad \text{HNF-App}\ \frac{\text{head-normal}_y(N) \qquad M : \Lambda X \qquad N : \Lambda X}{\text{head-normal}_y(NM)}$$

$$\text{HNF-Quot}\ \frac{N : \Lambda X \qquad \text{head-normal}_y(N) \qquad v : X^n \qquad y \notin v}{\text{head-normal}_y(\langle v \rangle\ '\ N)}$$

Which, along with β-reduction and the internal constructors, give the reduction relation (informally, as we are using "head-normal" to represent the procedure for checking for head normal form, with a term of the type representing the variable):

$$\frac{N : \Lambda\ X \qquad M : \Lambda\ (X \setminus x)}{((\lambda\ x.\ N)M) \rightsquigarrow N\ [x := M]}\ \beta \qquad \frac{v : X^n \qquad v_i : \Lambda\ X}{\langle v \rangle' x_i \rightsquigarrow (\texttt{var}\ (\texttt{numeral}\ (i))}\ '\text{-Var}$$

$$\frac{v :\ X^n \qquad M : \Lambda\ X \qquad x : X}{\langle v \rangle\ '\ (\lambda M) \rightsquigarrow \texttt{lam}\ \langle (v, x) \rangle M}\ '\text{-Abs} \qquad \frac{v : X^n \qquad M : \Lambda\ X \qquad N : \Lambda\ X \qquad \text{head-normal}_{v_i}(M)}{\langle v \rangle' (MN) \rightsquigarrow \texttt{app}(\langle v \rangle\ '\ M)(\langle v \rangle\ '\ N)}\ '\text{-App}$$

$$\frac{v : \Lambda\ X^n \qquad w : X^m \qquad M : \Lambda\ X \qquad \text{head-normal}_{w_i}(M)}{\langle v \rangle\ '\langle w \rangle\ '\ M \rightsquigarrow \texttt{quote}\ (\texttt{numeral}\ m)(\langle (v, w) \rangle' M)}\ '\text{-Quot}$$

or defining λ'-calculus in using De Bruijn indices, assuming a base type X for free variables , natural numbers, coproducts and a unit type, with $+n$ defined by $X + 0 = X; X + (n+1) = (X + n) + 1$ and representing the variables bound by quotation:

$$\frac{x : X}{x : \Lambda\ X}\ \text{Var} \qquad \frac{M : \Lambda\ (X + 1)}{\lambda M : \Lambda\ X}\ \text{Abs} \qquad \frac{M : \Lambda\ X \qquad N : \Lambda\ X}{(MN) : \Lambda\ X}\ \text{App} \qquad \frac{M : \Lambda\ (X + n)}{\langle n \rangle' M : \Lambda\ X}\ \text{Quote}$$

The identification of head normal form can still be described using inference rulesand parametrising it by the head term as above, and using $y \geq n$ to indicate that the variable is free and so is not denoted by one of the units in the iterated sum.

$$\frac{y : X}{\text{head normal}_y(n)}\ \text{HNF-Var} \qquad \frac{\text{head-normal}_y(N) \qquad M : \Lambda X}{\text{head-normal}_y(NM)}\ \text{HNF-App} \qquad \frac{\text{head-normal}_y(N) \qquad y \geq n}{\text{head-normal}_y(\langle n \rangle' N)}\ \text{HNF-Quot}$$

As the bound variables of the abstraction and quotation cases are represented by repeated categorical sums with unit, we do not type them in the lists of bound variables for quotation,

we only type it for the reduction of the internal variable constructor, as it requires the variable to be bound in the list. We use $(+)$:

$$\frac{N : \Lambda\ X + 1 \qquad M : \Lambda\ X}{((\lambda\ N)M) \rightsquigarrow N\ [M]}\ \beta \qquad \frac{i : \Lambda\ (X + n) \qquad i < n}{\langle n \rangle\ '\ x \rightsquigarrow (\texttt{var}\ (\texttt{numeral}\ n))}\ '\text{-Var}$$

$$\frac{M\ :\ \Lambda\ ((X\ +n)\ +1)}{\langle n \rangle\ '\ (\lambda\ M) \rightsquigarrow \texttt{lam}(\langle n+1 \rangle' M)}\ '\text{-Lam}$$

$$\frac{N : \Lambda\ (X + n) \qquad M\ : \Lambda\ (X + n) \qquad \text{head normal}_i(M) \qquad i < n}{\langle n \rangle\ '\ (MN) \rightsquigarrow \texttt{app}\ (\langle n \rangle\ '\ M)\ (\langle n \rangle\ '\ N)}\ '\text{-App}$$

$$\frac{M : \Lambda\ ((X\ +m)\ +n) \qquad \text{head-normal}_i(M) \qquad n \leq i > m}{\langle n \rangle\ '\ \langle m \rangle\ '\ M \rightsquigarrow \texttt{quote}\ (\texttt{numeral}(m))\ (\langle n+m \rangle\ '\ M)}\ '\text{-Quot}$$

The intention of requiring the head normal form (without λ/abstraction) is to block the reduction into internal constructors until the leftmost subterm of an application is a variable, and therefore not being able to expand further through β-reductions, and so can be quoted safely. Any abstraction subterm which has not been part of a β-redex can be safely quoted (any quoted redex, being an application, must be reduced for the application to be in head normal form). A double quotation (quotation of a quotation) for which the inner term is in head normal form has the list of bound variables concaternated for further reduction and internally represented as a numeral argument, as explained above.

The existence of a head normal form is undecidable, as shown in [3], but deciding whether a term is in head normal form is, which is shown in Section 3.3. This is evident intuitively, as anyone may look at the head position of a λ'-term and note whether a variable occupies this position.

If the term β-normalises, the entire term will eventually be represented using the internal constructors, with all quotation constructors absent from the final term. In the case of a quoted non-normalisable subterm, the quote construction prevents the entire term from being non-normalisable, allowing for continued computation of the rest of the term, along with identification of the unsolvable subterm by the presence of the quotation operator.

We can showcase this property by applying the term $\lambda x.\ \langle\ \rangle'\ x$ which will quote any term, to the K combinator/Boolean truth predicate $\lambda xy.\ x$. Calculating:

$$
\begin{aligned}
(\lambda x.\ \langle\ \rangle'\ x)(\lambda\ x\ y.\ x) &\rightsquigarrow \langle\rangle'\ (\lambda x\ y.\ x) \\
&\rightsquigarrow \texttt{lam}\langle x\rangle'\ (\lambda\ y.\ x) \\
&\rightsquigarrow \texttt{lam}(\texttt{lam}(\langle x, y\rangle' x)) \\
&\rightsquigarrow \texttt{lam}(\texttt{lam}(\texttt{var}(\texttt{numeral}1))
\end{aligned}
$$

We convert the variable x to its numeric representation 1 as it is the index of x in the list of quoted variables, and to better reflect the De Bruijn indices we will use for our formalisation.

Using the quotation operator, we admit a typed (with regards to the Scott encoding) form of reflection internal to λ', as we can define terms which inspect quoted functions. As an example, we can define a term counting all occurences of a variable in a term[2]:

$$
\begin{aligned}
\mathit{count} := \lambda t.t(&\lambda n\ m.\ (\texttt{equals}\ n\ m)1 0) \\
&(\lambda a\ a_r\ m.\ a_r\ (S\ m)) \\
&(\lambda\ a\ b\ a_r\ b_r\ m.\ \texttt{add}\ \ (a_r\ m)\ (b_r\ m)) \\
&(\lambda n\ a\ a_r m.\ a_r\ (\texttt{add}\ \ m\ n))
\end{aligned}
$$

Where $\texttt{equals} \equiv \lambda nm.and\ (n \leq m)\ (m \leq n)$ and $\texttt{add}\ \equiv \lambda mn.mn(\lambda tz.S\ z)$, with both the variable counting function and addition illustrating the utility of the recursive encoding, the addition substituting the second argument into the zero case and adding one in the successor case.

The counting function requires further explanation. We assume that the input replace each of the bound cases with a function, to which we also bind our target variable value m, which allows us to pass it down the recursive call tree. Intuitively, this allows us to "type" the variable counting function as $\Lambda \to \mathbb{N} \to \mathbb{N}$, where the first $\mathbb{N}$ is the de Bruijn index of the target variable and the final being the count, semantically overloading the naturals.

The variable case checks whether m is equal to the variable of the term. The abstraction case continues the recursion, incrementing the target variable to maintain the De Bruijn index (any λ-term will increment all variables under it). The application case adds the count of each side of the application, passing the target variable to each of the recursive cases. The quotation case is similar to the abstraction case, though since it binds more than one variable, the target is incremented by the bound variables.

[2]Using letters for variable names to improve readability

As an illustrative example, we apply `count` to the internally encoded term (0 (λ (0 1))) for the variable 0. The reduction is performed as follows:

$$
\begin{aligned}
&\mathit{count}\ (\texttt{app}\ (\texttt{var}\ 0)\ (\texttt{lam}\ (\texttt{app}\ (\texttt{var}\ 0)(\texttt{var}\ 1)))) \ 0 \\
&\rightsquigarrow^* \texttt{add}\ ((\lambda m.(\texttt{equals}\ 0\ m)\ 1\ 0)\ 0)((\lambda t_r m.\ t_r\ (S\ m)) \\
&\quad (\texttt{app}\ (\texttt{var}\ 1)\ (\texttt{var}\ 2))\ 1)(\ldots(\lambda a\ b\ a_r\ b_r\ m.\ \texttt{add}\ (a_r\ m)\ (b_r\ m))\ldots)1) \\
&\rightsquigarrow^* \texttt{add}\ 1((\lambda m.\ \texttt{add}\ ((\texttt{var}\ 1)(\lambda n m.\ (\texttt{equals}\ n\ m)1\ 0)\ldots) \\
&\quad ((\texttt{var}\ 2)((\lambda n\ m.\ (\texttt{equals}\ n\ m)1\ 0)\ldots)1) \\
&\rightsquigarrow^* \texttt{add}\ 1(\ \texttt{add}\ \ ((\lambda m.\ (\texttt{equals}\ 1\ m)1\ 0)\ 1)\ ((\lambda m.\ (\texttt{equals}\ 2\ m)1\ 0))1) \\
&\rightsquigarrow^* \texttt{add}\ 1\ 1 \\
&\rightsquigarrow^* 2
\end{aligned}
$$

We use the ... to signify all of the cases of the `count` function, along with contracting irrelevant redexes and some abuse of notation to bring into focus the cases being passed down the structure recursively. We note the successor case on the `lam` term, as 1 would refer to the same variable as 0 in the context of the outer `app`.

2.4 Confluence

Confluence (or the Church-Rosser property, to which it is equivalent) is a property of rewriting systems (which one can consider λ-calculus to be) defined as:

Definition 2.4.1 (Confluence)**.** *Let A, B, C be a terms in a rewriting system $(\Sigma, \rightsquigarrow)$, if $A \rightsquigarrow B$ and $A \rightsquigarrow C$, then there exists a term D such that $B \rightsquigarrow^* D$ and $C \rightsquigarrow^* D$.*

This is also known as the *diamond property*, named after the diagrammatic representation of the two reductions above. It can be intuitively understood as "however you reduce a term, you will eventually get the same result", with a simple example given by the natural numbers and the operations $(+, \times, ())$ e.g. we can reduce $(3 + 4) + (4 \times 3)$ as $(3 + 4) + (4 \times 3) \Rightarrow 7 + (4 \times 3) \Rightarrow 7 + 12 \Rightarrow 19$ and $(3 + 4) + (4 \times 3) \Rightarrow (3 + 4) + 12 \Rightarrow 7 + 12 \Rightarrow 19$, the terms $7 + 12$ and 19 are both D in the definition above.

We show this property by using the Tait–Martin-Löf–Takahashi method, for which we define a specific reduction relation, parallel reduction denoted $\rightsquigarrow^*$. The idea of parallel reduction is a reduction relation which reduces all, one or some of the available redexes, but only for a single reduction step, e.g. the parallel reduction of the term can be given as $(\lambda\ x.x)\ ((\lambda\ x.x)\ y)\ (\lambda x.z)\ w \rightsquigarrow^o \{((\lambda\ x.x)\ y)((\lambda x.z)\ w), (\lambda\ x.x)\ ((\lambda\ x.x)\ y)z, ((\lambda\ x.x)\ y)\ z\}$, but crucially not yz, as we do not perform any reductions beyond those available in the original term.

To prove confluence using $\rightsquigarrow^o$, we first show that $\rightsquigarrow$ is contained in $\rightsquigarrow^o$, which should hold trivially as $\rightsquigarrow^o$ is an extension of $\rightsquigarrow$. We proceed by defining a function $\_^* : \Lambda X \to \Lambda X$ (pronounced starred), which intuitively performs every reduction possible in a single step, which amounts to β-redexes for the regular λ-calculus (the term $(((\lambda\ x.x)\ y)\ z)$ in the example above) and the various quotation reductions. Using this function, we need to show a stronger result than the diamond property, namely *the triangle property*, which states that given two terms M, N and $M \rightsquigarrow^o N$, we can derive $N \rightsquigarrow^o M^*$, showing that the every parallel reduction "factors through" the $\_*$ function. Finally, we show that $\rightsquigarrow^o$ is contained in $\rightsquigarrow^*$, and conclude that since the regular reduction relation $\rightsquigarrow$ is contained in $\rightsquigarrow^o$ and $\rightsquigarrow^o$ is confluent, $\rightsquigarrow$ is confluent as well.

Chapter 3

Formalisation

In this chapter we present a formalisation of λ'-calculus, including a formal encoding of terms as a datatype, metatheoretical functions for mapping over and substituting into the free variables of the terms, an encoding of the reduction relation and its transitive reflexive closure, culminating in the proof of confluence using the Martin-Lof-Tait-Takahashi method, described informally in the previous chapter.

Following the formalised proof in Agda requires some knowledge its syntax. We liberally use the parameter inference mechanism Agda provides, denoted by _ for terms, allowing us to not name every parameter of a given function call, instead allowing the type inference mechanism of Agda to do the work for us.

In some parts, we will use λ-abstractions (instantiated in the metatheory , not in λ') as part of our definitions, denoted as λ x $\to$?, where ? is some term and x any number of bound variables. We start by introducing the imports from the `agda-unimath` [22] library which we will use:

```
{-# OPTIONS -without-K #-}

module Syntax where

open import univalent-combinatorics.standard-finite-types
  using (Fin; nat-Fin; raise-Fin)
open import elementary-number-theory.natural-numbers using
  (ℕ; zero-ℕ; succ-ℕ)
open import elementary-number-theory.addition-natural-numbers using
  (add-ℕ)
open import foundation.empty-types using
```

```
  (raise-empty; ex-falso)
open import foundation.raising-universe-levels using
  (map-raise)
open import foundation.unit-type using
  (star; unit; raise-unit ; raise-star)
open import foundation.identity-types
open import foundation.action-on-identifications-binary-functions
open import foundation.action-on-identifications-functions
open import foundation-core.function-types
open import foundation.negation using (¬)
open import foundation.dependent-pair-types
open import foundation.universe-levels
open import foundation.cartesian-product-types
open import foundation.coproduct-types using
  (_+_; inl; inr; is-injective-inl; neq-inr-inl)
```

The natural numbers $\mathbb{N}$ are the Peano numerals, the inductive type generated by a nullary constructor zero-$\mathbb{N}$ and a unary constructor succ-$\mathbb{N}$. We also import a number of the types discussed in the introduction, such as the unit type, the empty type (and associated negation), sums and product. We also include identity, and Π and *Sigma* types, along with some auxilliary functions relating to all of the above.

The finite type Fin is a type family indexed by the natural numbers, defined inductively as:

$$\texttt{Fin}\ 0 := \texttt{empty}$$
$$\texttt{Fin}\ (\texttt{n} + 1) := \texttt{Fin}\ \texttt{n} + \texttt{unit}$$

with $(+)$ being the the coproduct, serving as the canonical choice of finite types. We use it as a convenient way to encode De Bruijn indices(every bound unit representing a variable to be bound), as described in the previous chapter. Finally, we do not assume axiom K, a proper treatment of which is beyond the scope of this thesis, but we note that (1) assuming it will turn all types into mere sets and (2) we wish for the results to be as general as possible.

3.1 Syntax

Before giving our formalised proof of confluence, we present the syntax of λ'-calculus, encoded using De Bruijn indices. As discussed in the previous section, we define De Bruin indices

on a type level by coproducts of some type X and the unit type 1. We define some pattern bindings (renamings) to better suit our needs, along with a useful principle of contradiction which states that a bound variable cannot equal a free one:

```
pattern bound x = inr x
pattern free x = inl x

bound=free-absurd : ∀ {i j k} {X : UU i}{A : UU j}{B : UU k} {a : A} {b : B}
  → Id (bound a) (free b) → X
bound=free-absurd ()
```

The encoding of λ' is given by the datatype Λ, Λ being defined over some fixed type X in UU, the universe of small types, which all types we will use are elements of. We can see a nearly direct translation of the inference rules given for De Bruijn indices in the previous chapter, using λ as a standin for λ as λ is a keyword in Agda. The $\forall$ quantifier shadows the dependent function type and Agda's typechecker automatically infers it to be a natural number due to its use in $Fin\ n$.

```
infix 10 _'_

data Λ (X : UU) : UU where
  var  : X → Λ X
  л_   : Λ (X + unit) → Λ X
  app  : Λ X → Λ X → Λ X
  _'_ : ∀ n → Λ (X + Fin n) → Λ X
```

We also define a pattern for closed terms as a simple way of referring to the first bound varible in a term (for the definition of internal constructors and other examples) along with a type denoting closed terms.

```
pattern v = var (bound star)

Λ-Closed = ∀ {X} → Λ X
```

We define a map function over the Λ datatype, to lift functions on free variables to functions on terms, along with hinting at the functorial (and even monadic) structure we show explicitly later. To do so, we define a special case over the left side (free variables) of arbitrary coproducts, since both quotation and abstraction are constructed as iterated coproducts of units. We also define two syntaxes for later use in typesetting proofs.

```
map-free : ∀ {i i' j }{X : UU i}{X' : UU i'}{B : UU j}
         → (X → X')
         → X + B → X' + B
map-free f (free x) = free (f x)
map-free _ (bound b) = bound b

syntax map-free f = f ⁺

map-free' : ∀ {i i' j }{X : UU i}{X' : UU i'}
            (B : UU j)
          → (X → X')
          → X + B → X' + B
map-free' B f = map-free {B = B} f

syntax map-free' B f = f [ B ]⁺
```

The mapping function recursively applies the function $\phi : \mathsf{X} \to \mathsf{X'}$ down every constructor of the Λ type, using the shortened syntax for map-free for abstraction and quotation to only map the function over free variables (left side of the coproduct).

```
map : ∀ {X X'}
    → (X → X')
    → Λ X → Λ X'
map φ (var x) = var (φ x)
map φ (л f) = л (map (φ ⁺) f)
map φ (app f x) = app (map φ f) (map φ x)
map φ (Y ' f) = Y ' (map (φ ⁺) f)
```

The function shift-variable or $\__+$ allows us to expand the sum of variables under Λ, thereby encoding bound variables of a term, which is accomplished by mapping $free$ over it. The shortened syntax refers to $map - free$, used in our implementation of De Bruijn indices, along with the .

```
shift-variable : ∀ {X}{V : UU} → Λ X → Λ (X + V)
shift-variable = map free

syntax shift-variable x = x ₊
```

As examples of closed terms of Λ, we encode the identity function $\lambda\, x.x$ and the "quotation function" q, $\lambda x.\ \langle\ \rangle'\ x$. Note that the empty variable list is still part of the sum (as Fin 0 =

empty), so the variable bound by the л must be shifted. The function q may be applied to any other term, and will produce and internalised version of it.

```
i : Λ-Closed
i = л v

q : Λ-Closed
q = л (zero-ℕ ' v ₊)
```

As a substitution may change the type of free variables in a term, without changing the bound variables, $\__+$ is used in the definition of a similar function to $map-free$, $substitute-free-variables$, which applies a substitution to all free variables of a term. As above, we include a shorthand for readability, adding \ to reference a commonly used notation for substitution, $[x \setminus t]$.

```
substitute-free-variables : ∀ {X X' : UU}{B : UU}
  → (X → Λ X')
  → (X + B) → Λ (X' + B)
substitute-free-variables t (free x) = shift-variable (t x)
substitute-free-variables t (bound x) = var (bound x)

syntax substitute-free-variables f = f \⁺

substitute-free-variables' : ∀ {X X' : UU} (B : UU)
  → (X → Λ X')
  → (X + B) → Λ (X' + B)
substitute-free-variables' B f = substitute-free-variables {B = B} f

syntax substitute-free-variables' B f = f [ B ]\⁺
```

Substitution is then defined recursively down an expression, analogously to map above, with () used for the abstraction and quotation cases to handle free variables without changing the bound. Note that the signature of substitute is a flipped version of monadic composition, known as bind or (>>=) in Haskell.

```
substitute : ∀ {X X'}
           → (X → Λ X')
           → Λ X → Λ X'
substitute t (var x) = t x
substitute t (л f)
  = л (substitute (t \⁺) f)
substitute t (app f x)
```

```
  = app (substitute t f) (substitute t x)
substitute t (Y ' e)
  = Y ' substitute (t \⁺) e
```

Substitution of a single variable is denoted using the mixfix notation _[_], using the substitution function substitute-single below, and is defined to perform β-reduction.

```
substitute-single : ∀ {X : UU} → Λ X → (X + unit) → Λ X
substitute-single t (free x) = var x
substitute-single t (bound _) = t

_[_] : ∀ {X} → Λ (X + unit) → Λ X → Λ X
f [ x ] = substitute (substitute-single x) f

infix 100 _[_]
```

Vectors (lists indexed by a natural number) are defined for defining nested applications, used for the internal constructors of λ'.

```
Vec : ∀ {i} → ℕ → UU i → UU i
Vec zero-ℕ X = raise-unit _
Vec (succ-ℕ n) X = Vec n X × X

- map-Vec : ∀ {i j n} {X : UU i} {Y : UU j}
-          → (X → Y)
-          → Vec n X → Vec n Y
- map-Vec {n = zero-ℕ} f _ = raise-star
- map-Vec {n = succ-ℕ _} f (pair α x) = pair (map-Vec f α) (f x)

apps : ∀ {n X} → Λ X → Vec n (Λ X) → Λ X
apps {n = zero-ℕ} f _ = f
apps {n = succ-ℕ _} f (pair α x) = app (apps f α) x
```

To encode the internal constructors of λ' (as discussed in the previous chapter), we define functions in our metatheory to construct terms of "arities" 0, 1 and 2. Each construct requires a "template" (a term of Λ defined directly in the metatheory) which is substituted into by the construct functions. The template must match the arguments, and since Λ is defined over the finite type, we define them using finite types corresponding to the arities (Fin 0 for constant, Fin 1 for unary, Fin 2 on binary).

The unary constructor simply substitutes the argument into the template. The binary is case split on Fin 2 (which has the explicit type ((empty + unit) + unit)) using construct$_2$-substitution such that the arguments are substituted into the right sides.

```
raise-Fin' : ∀ k → ℕ → UU k
raise-Fin' k zero-ℕ = raise-empty k
raise-Fin' k (succ-ℕ n) = raise-Fin' k n + unit

construct₀ : Λ (Fin 0) → ∀ {X} → Λ X
construct₀ template = substitute (λ {()}) template

construct₁ : Λ (Fin 1) → ∀ {X} → Λ X → Λ X
construct₁ template argument = substitute (λ _ → argument) template

construct₂-substitution : ∀ {X} → Λ X → Λ X → Fin 2 → Λ X
construct₂-substitution argument₀ argument₁ (inr _) = argument₁
construct₂-substitution argument₀ argument₁ (inl _) = argument₀

construct₂ : Λ (Fin 2) → ∀ {X} → Λ X → Λ X → Λ X
construct₂ template argument₀ argument₁
  = substitute (construct₂-substitution argument₀ argument₁) template
```

The templates for each of the constructors, as mentioned in the previous chapter, bind "cases" for variables, application, abstraction and quotation (respectively) using the л from the Λ datatype. The recursive calls pass the cases down the expressions, with the var case terminating the recursion. Each template has at least one free variable, with application (left and right side of the application) and quotation (number of variables quoted and the term they belong to) having two. This can be seen from the "unit"s bound on the right of each of the templates. Since we are using De Bruijn indices, the innermost variable is bound first (denoted by v, as mentioned above), with following variable referenced by mapping free using shift-variable ($_+$).

As an example, the following terms:

```
var-template : {X : UU} → Λ (X + unit)
var-template = л л л л (app (v + + +) (v + + + +))

л-template : {X : UU} → Λ (X + unit)
л-template = л л л л (apps {n = 2} л-case ((raise-star , v + + + +) , recursive-call)) where
  var-case = v + + +
  л-case = v + +
  app-case = v +
  quote-case = v
  recursive-call = apps {n = 4} (v + + + +) ((((raise-star
                                              , var-case)
                                              , л-case)
```

```
                                                , app-case)
                                                , quote-case)
```

are the encodings of the internal constructors `var` and lam:

$$\texttt{var} := \lambda x\ c_v\ c_l\ c_a\ c_q.c_v x$$
$$\texttt{lam} := \lambda t\ c_v\ c_l\ c_a\ c_q.c_l\ t\ (t\ c_v\ c_l\ c_a\ c_q)$$

The outermost variable (corresponding to the quoted term) not being represented as an explicit л due to the use of $\mathsf{construct}_1$. Note how the recursive call $(t\ c_v\ c_l\ c_a\ c_q)$ is encoded using a where binding and the apps function.

The rest of the internal constructors are shown below, and the interested reader may compare them to those in the previous chapter.

```
app-template : {X : UU} → Λ ((X + unit) + unit)
app-template = л л л л (apps {n = 4} app-case ((((raise-star
                                                , (v +) + + + +)
                                                , v + + + +)
                                                , recursive-call-t)
                                                , recursive-call-u)) where
  var-case = v + + +
  л-case = v + +
  app-case = v +
  quote-case = v
  recursive-call-t = apps {n = 4} (v + + + + +) ((((raise-star
                                                , var-case)
                                                , л-case)
                                                , app-case)
                                                , quote-case)
  recursive-call-u = apps {n = 4} (v + + + +) ((((raise-star
                                                , var-case)
                                                , л-case)
                                                , app-case)
                                                , quote-case)


quote-template : {X : UU} → Λ ((X + unit) + unit)
quote-template = л л л л (apps {n = 3} quote-case
                              (((raise-star
                                , v + + + + +)
                                , v + + + +)
                                , recursive-call)) where
  var-case = v + + +
```

```
  л-case = v _+_+
  app-case = v _+_
  quote-case = v
  recursive-call = apps {n = 4} (v _+_+_+_+) ((((raise-star
                                                 , var-case)
                                                 , л-case)
                                                 , app-case)
                                                 , quote-case)

fix-template :  {X : UU} → Λ (X + unit)
fix-template = app (л (app (v _+_) (app v v )))
                   (л (app (v _+_) (app v v )))

s-template : {X : UU} → Λ (X + unit)
s-template = л л (app (app v (v _+_+)) (app (app (v _+_+) (v _+_)) v))
```

The internal constructors of λ'-calculus are defined below, using the $\mathsf{construct}_n$ along with a version of numerals encoded as an inductive datatype, as described in the previous chapter.

```
- The zero numeral
z-cons : Λ X
z-cons = л л (v _+_)

- Constructor for successor numerals
s-cons : Λ X → Λ X
s-cons = construct₁ s-template

- Encoding of standard numerals
numeral : ℕ → Λ X
numeral zero-ℕ = z-cons
numeral (succ-ℕ n) = s-cons (numeral n)

- The encoding of variables
var-cons : Λ X → Λ X
var-cons = construct₁ var-template

- The encoding of λ-abstraction
л-cons : Λ X → Λ X
л-cons = construct₁ л-template

- The encoding of function application
app-cons : Λ X → Λ X → Λ X
app-cons = construct₂ app-template
```

```
- The encoding of quoted terms
quote-cons : Λ X → Λ X → Λ X
quote-cons = construct₂ quote-template

- Encoding of fixed points
fix-cons : Λ X → Λ X
fix-cons = construct₁ fix-template
```

When reducing a quoted л-term n ' л t we have to add the bound variable to the list of variables represented by the finite type. The term t will be of the type ((X + Fin n) + unit), and to show it has been quoted we reassociate to the right to get $(X + (Finn + unit)) \equiv (X + (Fin(n + 1)))$, by mapping (in the metalanguage of Agda) the library function map-assoc-coprod : (A + B) + C → A + (B + C) over the quoted term t for

л to reassociate the bound variable and transform the л-term into its corresponding internal constructor while respecting the typing constraints.

```
quote-free-var : ∀ {X : UU} {Y : UU}
                 → Λ ((X + Y) + unit)
                 → Λ (X + (Y + unit))
quote-free-var = map map-assoc-coprod
```

Some more type-level yoga has to be done for the internal reduction of the double quoted term $n\ '\ (m\ '\ t)$, for which we wish to join the lists of quoted variables together, which amounts to joining the Finite types Fin n and Fin m. We do so inductively by defining a pair of helper functions for the binding the free variables m and bound variables n of the term, sending both of them to the bound/right side of the sum using the join-Fin function, and mapping join-Fin over the term.

```
free+ : ∀ m n → Fin m → Fin (add-ℕ m n)
free+ n zero-ℕ x = x
free+ n (succ-ℕ m) x = free (free+ n m x)

bound+ : ∀ m n → Fin n → Fin (add-ℕ m n)
bound+ n zero-ℕ ()
bound+ n (succ-ℕ m) (free x) = free (bound+ n m x)
bound+ n (succ-ℕ m) (bound x) = bound x


join-Fin : ∀ {j} {X : UU j} m n
                    → ((X + Fin m) + Fin n)
                    → (X + Fin (add-ℕ m n))
```

```
join-Fin m n (free (free x)) = free x
join-Fin m n (free (bound x)) = bound (free+ m n x)
join-Fin m n (bound x) = bound (bound+ m n x)

- Apply join-Fin to a term
join-quotes : ∀ {X} m n
            → Λ ((X + Fin m) + Fin n)
            → Λ (X + Fin (add-ℕ m n))
join-quotes m n = map (join-Fin m n)
```

3.2 Properties of Λ

We show a number of properties of mapping and substitution over λ' (using the identity type Id of Martin-Lof Type Theory as an internal notion of equality) for use in our proof of confluence. We first prove extensionality of mapping, introducing the general pattern for many of the proofs that follow: each of the properties is defined recursively over the constructors of Λ, along with an extra lemma showing the property for an arbitrary coproduct to account for the abstraction and quotation cases. The term refl is of the identity type, and is used when the claim is trivial i.e. all identities of the claim are definitionally equal.

Extensional equality of functions is pointwise equality, or that two functions are equal if they are equal for all of their inputs. This is idiomatically encoded as ∀ x → Id (f x) (g x). Extensionality of map is then the claim that if two functions are pointwise equal, then mapping those functions over any term is also pointwise equal.

Each of the cases first applies their respective constructs, using the functorial action of the identity type ap : (f : A → B) → {x y : A} → (p : Id x y) → (Id (f x) (f y)), ap-binary being the equivalent for binary functions, then either recursively applies the inductive claim or the pointwise equality.

```
- ap : (f : A → B) {x y : A} (p : Id x y) → (Id (f x) (f y))

map-free-pointwise : ∀ {X : UU} {A B : UU}
                   → (f : A → B)
                   → (g : A → B)
                   → (∀ x → Id (f x) (g x))
                   → ∀ (t : (A + X)) → Id ((f [ X ]⁺) t) ((g [ X ]⁺) t)
map-free-pointwise f g pw (free t) = ap free (pw t)
map-free-pointwise f g pw (bound t) = refl
```

```
map-pointwise : ∀ {A B}
              → (f : A → B)
              → (g : A → B)
              → (∀ x → Id (f x) (g x))
              → ∀ t → Id (map f t) (map g t)
map-pointwise f g pw (var t) = ap var (pw t)
map-pointwise f g pw (л t)   = ap л_
                                 (map-pointwise (f ⁺) (g ⁺)
                                 (map-free-pointwise _ _ pw ) t)
map-pointwise f g pw (app s t) = ap-binary app
                                   (map-pointwise f g pw s)
                                   (map-pointwise f g pw t)
map-pointwise f g pw (n ' t) = ap (n '_)
                                 (map-pointwise (f ⁺) (g ⁺)
                                   (map-free-pointwise _ _ pw ) t)
```

Proceeding with the same methodology as above, we show the compositionality of map, which claims that for all terms of Λ A, mapping two compatible functions (f : A → B) and (g : B → C) sequentially is identical to mapping their composition $(g \circ f)$. The variable case is definitionally equal, as $(g \circ f)\ x$ is defined as $g\ (f\ x)$.

```
map-free-∘ : ∀ {A B C : UU} {X : UU}
           → (f : A → B)
           → (g : B → C)
           → ∀ x → Id ((g [ X ]⁺) ((f ⁺) x))
                      (((g ∘ f) ⁺) x)
map-free-∘ f g (free x) = refl
map-free-∘ f g (bound x) = refl

map-composition : ∀ {A B C}
                → (f : A → B)
                → (g : B → C)
                → ∀ x → Id (map g (map f x))
                           (map (g ∘ f) x)
map-composition f g (var x) = refl
map-composition f g (л x) = ap л_
                              (map-composition (f ⁺)
                                               (g ⁺)
                                               x
                              · map-pointwise (λ x → (g [ unit ]⁺) ((f ⁺) x))
                                              ((g ∘ f) ⁺)
```

```
                                                    (map-free-∘ f g) x)
map-composition f g (app t u) = ap-binary app
                                                  (map-composition f g t)
                                                  (map-composition f g u)
map-composition f g (n ' x) = ap (n '_)
                                        (map-composition (f ⁺)
                                                         (g ⁺)
                                                           x
                                      · map-pointwise (λ x → (g [ Fin n ]⁺) ((f ⁺) x))
                                                      ((g ∘ f) ⁺)
                                                      (map-free-∘ f g) x)
```

Together, the proceeding lemmas illustrate key properties of map, the first fulfilling extensionality (as pointwise identical functions in the metatheory are reflected under map for terms of ΛX) and the second compositionality (composition of functions is equal to compositions of map), showing one part of the functoriality of our construction. The other, identity, can be shown as follows:

```
map-free-id : ∀ {X B : UU}
            → (t : (X + B))
            → Id ((map-free {B = B} id) t) t
map-free-id (free x) = refl
map-free-id (bound x) = refl

two-map-frees : ∀ {X B C : UU}
              → (x : (X + B) + C)
              → Id (map-free (map-free id) x) (map-free id x)
two-map-frees (free t) = ap free (map-free-id t)
two-map-frees (bound x) = refl

map-free-pw : ∀ {X B}
            → (t : Λ (X + B))
            → Id (map (map-free {B = B} id) t) (map id t)
map-free-pw (var x) = ap var (map-free-id x)
map-free-pw (л t) = ap л_ (map-pointwise _ _ two-map-frees t)
map-free-pw (app t u) = ap-binary app (map-free-pw t)
                                      (map-free-pw u)
map-free-pw (n ' t) = ap (n '_)
                         (map-pointwise _ _ two-map-frees t)

map-identity : ∀ {A}
             → (t : Λ A)
             → Id (map id t) (t)
map-identity (var x) = refl
```

```
map-identity (л t) = ap л_ (map-free-pw t · map-identity t)
map-identity (app t u) = ap-binary app (map-identity t)
                                       (map-identity u)
map-identity (n ' t) = ap (n '_) (map-free-pw t · map-identity t)
```

An extensionality property also holds for substitution, though we identify the "substitutions" f g : X → Λ X' in the metatheory. The proof follows much in the same way as above the exception of the case for free variables for the coproduct case, where the pointwise equality must be shifted from X to X + B, following the definition of \+.

```
\+-pointwise : ∀ {X X' : UU} {B : UU}
               → (f g : X → Λ X')
               → (∀ x → Id (f x) (g x))
               → ∀ t → Id ((f [ B ]\+) t)
                          ((g [ B ]\+) t)
\+-pointwise f g pw (free x) = ap shift-variable (pw x)
\+-pointwise f g pw (bound x) = refl

substitute-pointwise : ∀ {X X'}
                       → (f g : X → Λ X')
                       → (t : Λ X)
                       → (∀ x → Id (f x) (g x))
                       → Id (substitute f t) (substitute g t)
substitute-pointwise f g (var x) pw = pw x
substitute-pointwise f g (л t) pw
  = ap л_
      (substitute-pointwise (f \+)
                            (g \+)
                              t
                            (\+-pointwise f g pw))
substitute-pointwise f g (app t u) pw
  = ap-binary app
              (substitute-pointwise _ _ t pw)
              (substitute-pointwise _ _ u pw)
substitute-pointwise f g (n ' t) pw
  = ap (n '_)
       (substitute-pointwise (f \+)
                             (g \+)
                               t
                             (\+-pointwise f g pw))
```

To define the associativity of substitute, we need to prove that map and substitute will commute both from the left (substitution then map) and the right (map then substition), necessitated by the map function in our definition of substitution.

Starting with the left side of the identity and proceeding as before with the application and variable cases, the substitution and quotation necessitates the use of the composition of identities. Using the inductive call, we only have the substitution of the extended substitutions composed $(\phi\backslash^+ \circ f^+)$, and so we show its pointwise equality of substitution with the extended composition $(\phi \circ f)\backslash^+$ using the trivial (as both cases are definitionally equal after a case split) lemma substitute-map-coprod.

```
- _·_ : {x y z : A} → Id (x y) → Id (y z) → Id (x z)

substitute-map-coprod : ∀ {A B C : UU} {X : UU}
  → (f : A → B)
  → (ϕ : B → Λ C)
  → (x : A + X) → Id (((ϕ \+) ∘ (f +)) x)
                     (((ϕ ∘ f) \+) x)
substitute-map-coprod f ϕ (free x) = refl
substitute-map-coprod f ϕ (bound x) = refl

- Substitution of a mapped term is the subsitutition with the precomposition
substitute-map-commute-∘ : ∀ {A B C : UU}
                           → (x : Λ A)
                           → (f : A → B)
                           → (ϕ : B → Λ C)
                           → Id (substitute ϕ (map f x))
                                (substitute (ϕ ∘ f) x)
substitute-map-commute-∘ (var x) f ϕ = refl
substitute-map-commute-∘ (л x) f ϕ
  = ap л_
      ( substitute-map-commute-∘ x
                                 (f +)
                                 (ϕ \+)
      · substitute-pointwise     ((ϕ \+) ∘ (f +))
                                 ((ϕ ∘ f) \+)
                                 x
                                 (substitute-map-coprod _ _))
substitute-map-commute-∘ (app x x₁) f ϕ
  = ap-binary app
              (substitute-map-commute-∘ x _ _)
              (substitute-map-commute-∘ x₁ _ _)
substitute-map-commute-∘ (n ' x) f ϕ
  = ap (n '_)
       (substitute-map-commute-∘ x
                                 (f +)
                                 (ϕ \+)
```

```
      · substitute-pointwise ((ϕ \+) ∘ (f +))
                             ((ϕ ∘ f) \+)
                             x
                             (substitute-map-coprod _ _))
```

The other direction, namely mapping a function over a substitution, is the same as substituting the composition of a map with a substitution is proven along similar lines, though stated in the inverse direction from what we described. The pointwise extension is also less trivial.

In the case of free variables, we have to show that the extension of a map composed with a substitution is the same as mapping an extended function over an extended substitution. As the left side of $\backslash^+$ is simply shifting the variable (mapping inl to free) we first establish an identity of the composition of the function f with free, the substitution ϕ applied directly to a value (x : X). We then show that this composition is symmetric ($inl \circ f = f^+ \circ inl$) using pointwise equality and a trivial proof and finally reverse the composition, which maps free over ϕ t, giving us the definition for the free case of $\backslash^+$ = (map free (ϕ x)).

```
substitute-map-comm-pointwise : ∀ {X X' X" : UU} {B : UU}
                                → (ϕ : X → Λ X')
                                → (f : X' → X")
                                → (∀ t → Id (((map f ∘ ϕ) [ B ]\+ ) t)
                                            (map (f +) ((ϕ \+) t)))
substitute-map-comm-pointwise ϕ f (free x)
  =     map-composition _ _ _
    · ( map-pointwise     _ _ (\_ → refl) _
    · inv (map-composition _ _ _))
substitute-map-comm-pointwise ϕ f (bound x) = refl

substitute-map-commute : ∀ {X X' X" : UU}
                         → (ϕ : X → Λ X')
                         → (f : X' → X")
                         → (t : Λ X)
                         → Id (substitute (map f ∘ ϕ) t)
                              (map f (substitute ϕ t))
substitute-map-commute ϕ f (var x)          = refl
substitute-map-commute {X = X} ϕ f (л t)
  = ap л_ (substitute-pointwise _ _ t (substitute-map-comm-pointwise ϕ f)
          · substitute-map-commute (ϕ [ unit ]\+ ) (f +) t)
substitute-map-commute ϕ f (app t t₁)
  = ap-binary app
              (substitute-map-commute ϕ f t )
```

```
                (substitute-map-commute φ f t₁ )
substitute-map-commute φ f (n ' t)
  = ap (n '_)
       (substitute-pointwise _ _ t (substitute-map-comm-pointwise φ f)
       · substitute-map-commute (φ [ Fin n ]\⁺) (f ⁺) t)
```

Finally, we can prove that composition of two substitutions is equal to substituting consecutively, with the pointwise coproduct lemma using the previous precomposition and postcomposition lemmas. Just as before, we case split, have a trivial case for variables recurse on the application, and use a pointwise equality for the abstraction and quotation cases.

The pointwise coproduct lemma (with the desugared notation (x : A + D) to represent universal quantification of the coproduct) uses both of the previously proven identities to show that an expanded substitution (substitute (g \⁺)) into an expanded term under the function f ((f \⁺) x) is identical to the expanded substitution of their compositions (substitute g ∘f)\⁺. We first reassociate the internal expanded term into the substitution using the map precomposition identity (as \⁺ is defined as mapping inl for free variables), followed by the postcomposition identity moving free/inl outside as a map, concluding the proof.

```
substitute-composition-pointwise : ∀ {A B C : UU} {D : UU}
  → (f : A → Λ B)
  → (g : B → Λ C)
  → (x : A + D)
  → Id (substitute (g \⁺) ((f \⁺) x)) (((substitute g ∘ f) \⁺) x)
substitute-composition-pointwise f g (free x)
  = substitute-map-commute-∘ (f x) inl (substitute-free-variables g)
  · substitute-map-commute g inl (f x)
substitute-composition-pointwise f g (bound x) = refl

substitute-composition : ∀ {A B C}
                       → (f : A → Λ B)
                       → (g : B → Λ C)
                       → ∀ x → Id (substitute g (substitute f x))
                                  (substitute (substitute g ∘ f) x)
substitute-composition f g (var x) = refl
substitute-composition f g (л x)
  = ap л_ ( substitute-composition (f \⁺) (g \⁺) x
          · substitute-pointwise (λ x → substitute (g \⁺) ((f \⁺) x))
                                 ((substitute g ∘ f) \⁺)
                                 x
                                 (substitute-composition-pointwise _ _))
substitute-composition f g (app t u)
  = ap-binary app
```

```
        (substitute-composition f g t)
        (substitute-composition f g u)
substitute-composition f g (n ' x)
  = ap (n '_)
       ( substitute-composition (f \+) (g \+) x
       · substitute-pointwise (substitute (g \+) ∘ (f \+))
                              ((substitute g ∘ f) \+)
                              x
                              (substitute-composition-pointwise _ _))
```

Using the commutativity of map and substitution (as a conversion mechanism between terms of map and substitute) and the pointwise equality of substitution, the setup of term substitution into the internal constructors allows us to prove the commutativity of map with the internal constructors.

```
construct₀-map : ∀ {A B} t
               → (f : A → B)
               → Id (map f (construct₀ t))
                    (construct₀ t)
construct₀-map t f = inv (substitute-map-commute _ f t)
                   · (substitute-pointwise _ _ t (λ{(())}))

construct₁-map : ∀ {A B} t
               → (f : A → B)
               → ∀ u
               → Id (map f (construct₁ t u))
                    (construct₁ t (map f u))
construct₁-map t f u = inv (substitute-map-commute _ f t)
                     · (substitute-pointwise _ _ t (λ _ → refl))

construct₂-map : ∀ {A B} t
               → (f : A → B)
               → ∀ u r
               → Id (map f (construct₂ t u r))
                    (construct₂ t (map f u) (map f r))
construct₂-map t f u r = inv (substitute-map-commute _ f t)
                       · (substitute-pointwise _ _ t construct₂-substitution-map) where
  construct₂-substitution-map : ∀ x → Id ((map f ∘ construct₂-substitution u r) x)
                                        (construct₂-substitution (map f u) (map f r) x)
  construct₂-substitution-map (inl _) = refl
  construct₂-substitution-map (inr _) = refl
```

As mentioned above, the internal constructors are special cases of substitute, so the composition for substitution suffices to prove the commutativity of substitution with the internal constructors.

```
construct₀-substitute : ∀ {A B} t
                      → (ϕ : A → Λ B)
                      → Id (substitute ϕ (construct₀ t))
                           (construct₀ t)
construct₀-substitute t ϕ = substitute-composition _ ϕ t
                          · substitute-pointwise _ _ t (λ{()})

construct₁-substitute : ∀ {A B} t
                      → (ϕ : A → Λ B)
                      → (u : Λ A)
                      → Id (substitute ϕ (construct₁ t u))
                           (construct₁ t (substitute ϕ u))
construct₁-substitute t ϕ u = substitute-composition _ ϕ t
                            · substitute-pointwise _ _ t (\_ → refl)


construct₂-substitute : ∀ {A B} t
  → (ϕ : A → Λ B)
  → (u r : Λ A)
  → Id (substitute ϕ (construct₂ t u r))
                     (construct₂ t (substitute ϕ u) (substitute ϕ r))
construct₂-substitute t ϕ u r = substitute-composition _ ϕ t
                              · substitute-pointwise _ _ t construct₂-substitution-substitution where
  construct₂-substitution-substitution : ∀ x
                                       → Id ((substitute ϕ ∘ construct₂-substitution u r) x)
                                            (construct₂-substitution (substitute ϕ u) (substitute ϕ r) x)
  construct₂-substitution-substitution (inl _) = refl
  construct₂-substitution-substitution (inr _) = refl
```

Identities for mapping and substituting for each of the specific internal constructors are included in the appendix, as they follow definitionally from applications of the above identities.

3.2.1 Monadic properties of Λ

We show monadic properties we've been hinting at above. A notion of monad (as an programming idiom) is considering as a functor/type constructor with two operations, unit : a → M a, a lifting of values "into" the monad, and bind : M a → (a → M b) → M b, which combines monadic functions. In our case, the constructor var : X → Λ X is unit and substitute : (X → Λ X') → Λ X → Λ X' is bind with the first two arguments flipped, which we will call bind'.

There are multiple equivalent methods of defining and proving that an object is a monad, we chose to follow the Haskell methodology and show that the aforementioned unit ($\mathsf{u} : \mathsf{A} \to \mathsf{T\ A}$) and composition ($\_ \otimes \_ : \to (\mathsf{A} \to \mathsf{T\ B}) \to (\mathsf{B} \to \mathsf{T\ C}) \to \mathsf{T\ C}$) of a Kleisli category[16] fulfill the following axioms for all $\phi : \mathsf{A} \to \mathsf{T\ B}$, $\beta : \mathsf{B} \to \mathsf{T\ C}$ and $\alpha : \mathsf{C} \to \mathsf{T\ D}$, giving the corresponding expression using bind':

$$
\begin{aligned}
\mathsf{u} \otimes \phi &= \phi && \textit{Left Identity} \\
&= \mathsf{bind}'\ u \circ \phi \\
\phi \otimes \mathsf{u} &= \phi && \textit{Right Identity} \\
&= \mathsf{bind}'\ \phi \circ \mathsf{u} \\
(\phi \otimes \beta) \otimes \alpha &= \phi \otimes (\beta \otimes \alpha) && \textit{Associativity} \\
\mathsf{bind}'\ \phi \circ (\mathsf{bind}'\ \beta \circ \alpha) &= \mathsf{bind}'(\mathsf{bind}'\ \phi \circ \beta) \circ \alpha
\end{aligned}
$$

We note that in our case, these properties hold exensionally, in the sense that quantification over a term $\mathsf{t} : \mathsf{A}$ is required for all of the identities below and above to hold, so all This makes Λ a monad in some E-Category, a category enriched over setoids[20].

The constructor var is shown to be the identity of the Kleisli category by a similar argument to many of the lemmas above, with no outstanding issues:

```
var-identity-pointwise : ∀ {X B}
                       → (x : X + B)
                       → Id ((var \⁺) x) (var x)
var-identity-pointwise (free x) = refl
var-identity-pointwise (bound x) = refl

var-identity : ∀ {X}
             → ∀ (t : Λ X)
             → Id (substitute var t) t
var-identity (var x) = refl
var-identity (л x)
  = ap л_ (substitute-pointwise (var \⁺)
        var
        x
        var-identity-pointwise
        · var-identity x)
var-identity (app t u)
  = ap-binary app
               (var-identity t)
               (var-identity u)
var-identity (n ' x)
```

```
  = ap (n '_) (substitute-pointwise (var \^+)
                                    var
                                    x
                 var-identity-pointwise
               · var-identity x)
```

The properties of var as a monadic identity for Λ are the first and second laws above, and are respectively definitionally true or immediate form the identity above.

```
right-var-substitute : ∀ {X X' : UU}
                       → (ϕ : X → Λ X')
                       → ∀ x → Id (substitute ϕ (var x))
                                  (ϕ x)
right-var-substitute ϕ x = refl

left-var-substitute : ∀ {X X' : UU}
                      → (ϕ : X → Λ X')
                      → ∀ x → Id (substitute var (ϕ x))
                                 (ϕ x)
left-var-substitute ϕ x = var-identity (ϕ x)
```

Finally, we give an identity related to the Applicative typeclass of Haskell. It states that mapping a function f (map f x) is identical to the subsititution of f postcomposed with the variable constructor (which we interpret as our unit), which can be intutively seen from the type information of (substitute (var ∘ f)), is typed as var : X → Λ X and (f : X → X'), so postcomposing gives (Λ X → Λ X'), which corresponds to the type of a function mapping map f.

The actual proof is along the same lines as the previous proofs, again showing the pointwise equality for coproducts and composing it with a recursive call on the induction hypothesis.

```
map-subst-law-pointwise : ∀ {X X'} {B}
                          → (f : X → X')
                          → (x : X + B )
                          → Id (((var ∘ f) \^+) x)
                               ((var ∘ (f ^+)) x)
map-subst-law-pointwise f (free x) = refl
map-subst-law-pointwise f (bound x) = refl

map-subst-law : ∀ {X X'}
```

```
                  → (x : Λ X )
                  → (f : X → X')
                  → Id (substitute (var ∘ f) x) (map f x)
map-subst-law (var x) f = refl
map-subst-law {X = X} (л x) f
  = ap л_ (substitute-pointwise _ _ x
              (map-subst-law-pointwise f)
         · (map-subst-law x (f ⁺)))
map-subst-law (app t u) f
  = ap-binary app (map-subst-law t f) (map-subst-law u f)
map-subst-law (n ' x) f
  = ap (n '_)
       (substitute-pointwise _ _ x (map-subst-law-pointwise f)
      · map-subst-law x (f ⁺))
```

Finally, we show the associative property of substitute, proving the third law stated above and establishing Λ as a monad, the proof relying on two applications of the substitute-composition.

```
substitute-assoc : ∀ {A B C D : UU}
                  → (ϕ : A → Λ B)
                  → (ψ : B → Λ C)
                  → (α : C → Λ D)
                  → ∀ x → Id (substitute (substitute α ∘ ψ) (ϕ x))
                             (substitute α (substitute ψ (ϕ x)))
substitute-assoc ϕ ψ α x = inv (substitute-composition ψ α (ϕ x))
```

This concludes all the syntactical notions we need, and we can move our discussion to the reduction relation before finally finishing with the proof of confluence.

3.3 Reduction

We continue our journey towards the formalised proof of confluence by giving the formalisation of the reduction relation outline in the chapter on λ'-calculus, giving us a way to computationally reason it in Agda. We also include formalisations of our notion of head normal form and the transitive reflexive closure of the reduction relation.

3.3.1 Single step reduction

Reductions for the each of the constructors (lambda, applications on the left and right, quotation) "pass through" towards the subterms, building terms until a potential β-reduction, an application of a λ-term, is reached, at which point a reduction is performed. Note how a quotation term is defined to have the same behaviour, allowing β-reduction to be performed under the quote, as advertised in the introduction.

```
infix 8 _⇝_

data _⇝_ {X} : (Λ X) → (Λ X) → UU₁ where
  β : ∀ f x → (app (л f) x) ⇝ f [ x ]
  л⇝       : ∀ f f' → f ⇝ f' → л f ⇝ л f'
  app⇝left : ∀ f f' x → f ⇝ f' → app f x ⇝ app f' x
  app⇝right : ∀ f x x' → x ⇝ x' → app f x ⇝ app f x'
  '⇝       : ∀ n t t' → t ⇝ t' → (n ' t) ⇝ (n ' t')
```

The variable term encodes a De Bruijn index, and so we use the internal numeral constructor to represent which variable is being quoted, syntactically corresponding to the index of a variable in the list. nat-Fin is a library function taking elements of the finite type to the natural numbers, returning the number corresponding to the element of the finite type.

```
- Quoting variables
'var⇝ : ∀ n j → n ' (var (bound j))
              ⇝ var-cons (numeral (nat-Fin n j))
```

The reduction to the internal abstraction term moves the bound variable of the λ into the list of quoted variables, incrementing the number of quoted variablese, and applying the function quote-free-var to reassociate on the type level.

```
- Quoting λ-abstractions
'л⇝ : ∀ n f → n ' (л f) ⇝ л-cons (succ-ℕ n ' quote-free-var f)
```

Before discussing the next two cases of the reduction relation, we take a detour to introduce our formalisation of head normal form, as discussed in the previous chapter.

3.3.2 Head Normal Form

Our definition of Head Normal Form (also known as β-normal form) is non-standard, as it does not permit any λ-terms to be in head normal form the "head" (leftmost term of a sequence of applications), which is possible in the literature, where it is usually defined as the leftmost term before any bound variables i.e. not in the form $\lambda x_1 \lambda x_2 \ldots \lambda x_n.\ ((\lambda t.N)\ M_1\ \ldots M_m)$, or equivalently, in the form $\lambda x_1 \lambda x_2 \ldots \lambda x_n.\ y\ M_1\ \ldots M_m$. We lack a constructor for λ-terms as we handle them separately, we cannot have any abstractions binding applications or quotations in this version of head normal form, leaving us with terms of the form(s) $(\langle x_1,\ x_2 \ldots\ x_n \rangle\ '\ )\ y\ M_1\ M_2 \ldots M_m$. Using this definition, we can guarantee that redexes reduce under the quote before being reduced to the internal constructors. (λ-terms might still be in head normal form with the canonical definition).

The head normal form is defined as a family over an arbitrary type X (the type of free variables) Any variable term will trivially be in head-normal form, whilst the application case requires the first (leftmost) term to be in head normal form, moving leftwards through the term inductively.

```
data head-normal {X} (x : X) : Λ X → UU₁ where
  var-hn : head-normal x (var x)
  app-hn : ∀ t u → head-normal x t → head-normal x (app t u)
  '-hn : ∀ n t → head-normal (free x) t → head-normal x (n ' t)
```

Identity (and therefore uniqueness) of two normal forms is as always given recursively down each of the cases and distinguished by the head variable, and so the same term will have the same variable in the head (leftmost position) h1 and h2 being the head normal forms of the first term of an application (for applications):

```
head-normal-Id : ∀ {X} (t : Λ X) (x : X) (y : X)
                 → head-normal x t → head-normal y t → Id x y
head-normal-Id t x y var-hn var-hn = refl
head-normal-Id _ _ _ (app-hn t u h1) (app-hn t' u' h2) = head-normal-Id _ _ _ h1 h2
head-normal-Id _ _ _ ('-hn n t h1) ('-hn .n .t h2) = is-injective-inl
                                                      (head-normal-Id _ _ _ h1 h2)
```

The function is-injective-inl is a result from the Unimath library and gives (eponymously) the injectivity of the the left side of a sum. Applying it to the recursion in the '-hn case, effectively only considering the free variables of the quoted term when determining if its head variable identical to another.

Returning to the reduction relation, to convert an application into its corresponding internal constructor while preserving confluence, the quoted application must be in head normal form

for an already quoted variable. This prevents a term such as $\langle\rangle\ '\ ((\lambda x.x)\ y)$ from yielding two different reductions, either by first quoting the lambda and then the variable, or by performing β-reduction and quoting the result.

Stipulating the requirement of the application being in head normal form ensures the "quotability" of the left side of the application (as it has a variable furthest to the left), while still leaving the option for terms within the application to be unquotable.

$$\begin{aligned}
&\text{'app}\rightsquigarrow : \forall\ n\ t\ u\ i \\
&\qquad \to \text{head-normal}\ (\text{bound}\ i)\ t \\
&\qquad \to n\ '\ (\text{app}\ t\ u) \\
&\qquad \rightsquigarrow \text{app-cons}\ (n\ '\ t)\ (n\ '\ u)
\end{aligned}$$

Reducing to the internal quotation constructor, or quoting a quote, is similar to the application term, as we require a head normal form for some bound variable (present in the list of quoted variables), but with the stipulation that it is not present in the inner (m-sized) list. The result is the internal constructor quoting the list of variables of the internal constructor, and proceeding with the reduction by also adding it to the rest of the term and joining the quotes (on the type level, adding the finite types together).

$$\begin{aligned}
&\text{''}\rightsquigarrow : \forall\ m\ n\ i\ t \\
&\qquad \to \text{head-normal}\ (\text{free}\ (\text{bound}\ i))\ t \\
&\qquad \to n\ '\ (m\ '\ t) \\
&\qquad \rightsquigarrow \text{quote-cons}\ (\text{numeral}\ m)\ ((\text{add-}\mathbb{N}\ n\ m)\ '\ \text{join-quotes}\ n\ m\ t)
\end{aligned}$$

The the final term of iterated reduction will only contain terms with the constructors of the λ-calculus (or diverge as with other divergent terms). We can reason about iterated reduction by encoding a notion of the transitive reflexive closure of a term.

3.3.3 Decidability of head normal form

We define a decision procedure for checking whether or not a term is in head normal form, encoding it as a sum where inl witnessing the normal form and inr withnessing the lack of one. Variables are trivially in head normal form, and so are returned in the left case of the coproduct, var-hn witnessing the head normal form. Lambda terms are definitionally never in head normal form, as shown below, and we use a metalanguage λ-function to encode the universal quantification of the lemma λ-hn-is-empty. Application and quotation are defined using lemmas discussed below, passing an extra call with the decision procedure for the the right side of the application and the quoted term respectively.

```
decide-head-normal : ∀ {X}
                   → (t : Λ _)
                   → (Σ X ( λ x → head-normal x t))
                        + ((x : X) → ¬ (head-normal x t))
decide-head-normal (var x) = inl (x , var-hn)
decide-head-normal (л t)   = inr (λ x → л-hn-is-empty t x)
decide-head-normal (app t u) = decide-app t u ((decide-head-normal t ))
decide-head-normal (n ' t) = decide-' n t (decide-head-normal {_ + Fin n} t)
```

We vacuously observe that abstraction terms are by definition never in (abstraction-free) head normal form, as there is no constructor in the datatype head-normal including a lambda.

```
л-hn-is-empty : ∀ {X}
              → (t : Λ (X + unit))
              → (x : X) → ¬ (head-normal x (л t))
л-hn-is-empty t x ()
```

For the quotation case, we need to further recurse on whether or not the head variable is bound (part of the list of quoted variables) or free (the term being quoted) and so use a seperate decision procedure.

If the quoted term is in head normal form for a free variable x, the quotation will also be in head normal form, so we encode it using the inl constructor over a dependent pair along the head normal form term using the quotation constructor.

When a term is not in head normal form, the quotation will not have a normal form, which we show through a *pattern matching lambda*, a feature of Agda allowing us to define a top-level helper function inline and letting us give the non-existence of a head normal form for a quoted term using the non-existence of the quotation.

```
decide-' : ∀ {X} (n : ℕ) (t : Λ (X + Fin n))
         → (Σ (X + Fin n) (λ x → head-normal x t))
              + ((i : (X + Fin n)) → ¬ (head-normal i t))
         → (Σ X ( λ x → head-normal x (n ' t)))
              + ((x : X) → ¬ (head-normal x (n ' t)) )
decide-' n t (inl (free x , hnf)) = inl (x , ('-hn n t hnf))
decide-' n t (inl (bound x , hnf)) = inr (λ x₁ t₁ → ¬-decides-'-head-normal _ _ t hnf t₁)
decide-' n t (inr n-hnf)           = inr (λ{ x ('-hn .n .t hnf) → n-hnf (free x) hnf})
```

A term cannot be in a head normal form for a bound variable and we show it by the following lemma, which uses the identity of head normal form along with the absurdity of equality between the two sides of a coproduct:

```
¬-decides-'-head-normal : ∀ {X : UU} {n : ℕ} → (x : X) → (i : Fin n)
                        → (t : Λ (X + Fin n))
                        → (head-normal {X = (X + Fin n)} (bound i) t)
                        → ¬ (head-normal x (n ' t))
¬-decides-'-head-normal x i₁ t x₁ ('-hn _ .t x2) = bound=free-absurd
                                                    (head-normal-Id _ _ _ x₁ x2 )
```

showing that for all lists of quoted variables (terms of the finite type $\mathsf{x_1}$: Fin n) and all terms with n free variables $\mathsf{t_1}$: Λ (X + Fin n), there are no head normal forms of the quoted term given the (non-existant) normal form of the quoted variables.

The application case is simpler, as it only depends on the head normal form of the first term t in the application. If the term is in head normal form, we extract the witness of this fact using pr1, the first projection of the head normal form, and use the inl constructor to encode the continued decision procedure.

```
decide-app : ∀ {X} (t u : Λ X)
           → (Σ X ( λ x → head-normal x t))
                + ((x : X) → ¬ (head-normal x t))
           → (Σ X ( λ x → head-normal x (app t u)))
                + ((x : X) → ¬ (head-normal x (app t u)))
decide-app t u (inl x) = inl ((pr1 x) , (app-hn t u (pr2 x)))
decide-app t u (inr x) = inr (λ y → ¬-decides-app-head-normal y t u (x y))
```

Finally if the term is not in head normal form, we use the lemma below to show that if the left side of an application is not in head normal form (¬ (head-normal x t)), neither is the rest of the application (¬ (head-normal x (app t u))).

```
¬-decides-app-head-normal : ∀ {X}
                          → (x : X)
                          → (t u : Λ X)
                          → ¬ (head-normal x t)
                          → ¬ (head-normal x (app t u))
¬-decides-app-head-normal x t u n-hnf (app-hn .t .u x2) = n-hnf x2
```

3.3.4 Transitive Reflexive Closure

Encoding the transitive reflexive closure of the reduction relation ($\rightsquigarrow^*$) can be done in a fairly minimal way, defining it for a Λ-term t. Reflexivity is simply encoded as a relation of

t with itself, the reflexivity of $\rightsquigarrow$. The encoding of transitivity requires a reduction of the term in the context $t \rightsquigarrow u$ and the next step/termination of the reduction $u \rightsquigarrow *v$, giving the transitive composition $t \rightsquigarrow *v$.

```
infix 25 _⇝*_
data _⇝*_ {X} (t : Λ X) : (Λ X) → UU₁ where
  ⇝*-refl : t ⇝* t
  ⇝*-trans : ∀ {u v} → t ⇝ u → u ⇝* v → t ⇝* v
```

The transitivity of $\rightsquigarrow *$ is used to show the inclusion of the parallel reduction relation in the reflexive transitive closure, part of the proof of confluence. It's shown through induction on both arguments, with a recursion on the t Note how the lemma below includes two terms of the trans. refl. closure in its context, (x $\rightsquigarrow$* y) and (y $\rightsquigarrow$* z), in contrast to the transitivity constructor for the datatype, which requires a single step reduction $\rightsquigarrow$.

```
⇝*-is-transitive : ∀ {X} {x : Λ X} {y : Λ X} {z : Λ X}
                   → x ⇝* y
                   → y ⇝* z
                   → x ⇝* z
⇝*-is-transitive s ⇝*-refl                = s
⇝*-is-transitive ⇝*-refl (⇝*-trans x t) = ⇝*-trans x t
⇝*-is-transitive (⇝*-trans x s) (⇝*-trans y t) = ⇝*-trans x
                                                   (⇝*-is-transitive s
                                                     (⇝*-trans y t))
```

Each of the constructors in the reduction relation allows for a pass-through of a term into the corresponding term of the transitive reflexive closure, allowing us to build terms from sub-terms similarly to the reduction relation. The complete proofs for each of these are included in the appendix, as each contains the same cases:

```
app-⇝*-left : ∀ {X}
  → {f f' x : Λ X}
  → f ⇝* f'
  → app f x ⇝* app f' x
app-⇝*-left ⇝*-refl                = ⇝*-refl
app-⇝*-left (⇝*-trans x ⇝*-refl) = ⇝*-trans (app⇝left _ _ _ x)
                                             ⇝*-refl
app-⇝*-left (⇝*-trans x (⇝*-trans x₁ red)) = ⇝*-trans (app⇝left _ _ _ x)
                                                      (app-⇝*-left
                                                        (⇝*-trans x₁ red))
```

The first encodes for no reductions (⇝*-refl), and reflexivity suffices, the next for a single step of the reduction, using the corresponding constructor of the original reduction and the final for any amount of steps (⇝*-trans x (⇝*-trans x_1 red)), applying the constructor recursively by applying the lemma to the transitive constructor red.

Each of the internal constructors also pass through the reflexive transitive closure, and we show them for fixed terms substituted into the constructors (and whether or not they are in head normal form), e.g. the terms of the application and the head normal form of the first subterm. This allows the proofs to be particularly simple, as we do not recurse over any reduction. We can also construct a more general formulation of this pass-through which directly takes a term $\rightsquigarrow$ and gives a term of $\rightsquigarrow *$.

```
cons-↝* : ∀ {X} {n : ℕ} {a b : Fin n → Λ X}
            → (j : Fin n)
            → a j ↝ b j
            → a j ↝* b j
cons-↝* j red = ↝*-trans red ↝*-refl

var-cons-↝* : ∀ {X} {n : ℕ}
            → (j : Fin n)
            → (n ' var (inr {A = X} j)) ↝* var-cons (numeral (nat-Fin n j))
var-cons-↝* {n = n} j = cons-↝* {n = n}
                                {a = (λ t → n ' var (inr j))}
                                {b = (λ t → var-cons (numeral (nat-Fin n j)))}
                                j ('var↝ n j)

л-cons-↝* : ∀ {X} {n : ℕ}
          → (f : Λ ((X + Fin n) + unit))
          → (n ' (л f)) ↝* (л-cons (succ-ℕ n ' quote-free-var f))
л-cons-↝* f =  ↝*-trans ('л↝ _ f) ↝*-refl

app-cons-↝* : ∀ {X} {n : ℕ}
            → (i : Fin n)
            → (t u : Λ (X + Fin n) )
            → head-normal (bound i) t
            → (n ' app t u) ↝* app-cons (n ' t) (n ' u)
app-cons-↝* i t u hnf = ↝*-trans ('app↝ _ t u i hnf) ↝*-refl

quote-cons-↝* : ∀ {X} (m n : ℕ)
              → (i : Fin n)
              → (t : Λ ((X + Fin n) + Fin m))
              → (head-normal (free (bound i)) t)
              → (n ' (m ' t)) ↝* quote-cons (numeral m) ((add-ℕ n m) ' join-quotes n m t)
quote-cons-↝* n m i t hnf = ↝*-trans ('"↝ n m i _ hnf) ↝*-refl
```

3.4 Confluence

We have finally reached the point in this thesis where we can show the confluence of λ'. First, we define the parallel reduction relation as an inductive relation over some type X,

similar to the regular reduction relation:

```
infix 9 _⇝∘_
data _⇝∘_ {X} : Λ X → Λ X → UU₁ where
```

The parallel reduction has the same signature as the regular reduction relation as one might expect, but the differences start already with the first few constructors:

```
β∘   : ∀ f f' x x' → f ⇝∘ f'
                   → x ⇝∘ x' → (app (л f) x) ⇝∘ f' [ x' ]
var∘ : ∀ x         → (var x) ⇝∘ (var x)
app∘ : ∀ f f' x x' → f ⇝∘ f' → x ⇝∘ x' → app f x ⇝∘ app f' x'
л∘   : ∀ f f'      → f ⇝∘ f' → л f ⇝∘ л f'
```

The parallel reduction quantifies over more terms (ΛX's) than the original and contains more terms in the context for application and β-reduction, as we need to encode all possible single-step reductions, which the added $\rightsquigarrow\circ$-constraints encode. We can (and will below) recover the regular reduction relation by considering the marked variables (f' and x') as f and x.

```
'∘      : ∀ n t t'       → t ⇝∘ t' → n ' t ⇝∘ n ' t'
'-var∘  : ∀ n j          → n ' (var (bound j)) ⇝∘ var-cons (numeral (nat-Fin n j))
'л∘     : ∀ n f f'       → f ⇝∘ f'
                         → n ' (л f) ⇝∘ л-cons (succ-ℕ n ' quote-free-var f')
'-app∘  : ∀ n t t' u u' i → head-normal (bound i) t
                         → t ⇝∘ t'
                         → u ⇝∘ u'
                         → n ' (app t u) ⇝∘ app-cons (n ' t') (n ' u')
''∘     : ∀ n m t t' i   → head-normal (free (bound i)) t
                         → t ⇝∘ t'
                         → n ' (m ' t) ⇝∘ quote-cons (numeral m)
                                                    ((add-ℕ n m) ' join-quotes n m t')
```

The story with internal constructors is much the same as the reduction relation, with additional constraints on the reduction which allow for the transfer of a parallel reduction (a single-step redex) under the internal constructors.

We define the metatheoretical function $*$ or starred, which as mentioned above reduces every single step reduction available, but crucially does not do so for multiple reduction steps. To do so, we define a few helper functions for the internal constructors, which will mutually

recurse with the main function note the split on the application term to force β-redexes to reduce and the calls to the helper functions, where the last argument is a call to starred t:

```
mutual
  starred : ∀ {X} → Λ X → Λ X
  starred (var x)                = var x
  starred (л x)                  = л starred x
  starred (app (л x) y)          = (starred x) [ (starred y) ]
  starred (app (var x) y)        = app (starred (var x)) (starred y )
  starred (app (app x x₁) y)     = app (starred (app x x₁) ) (starred y)
  starred (app (n ' x) y)        = app (starred (n ' x)) (starred y)
  starred (n ' var (free x))     = n ' (starred (var (free x)))
  starred (n ' var (bound x))    = var-cons ((numeral (nat-Fin n x)))
  starred (n ' л x)              = л-cons (succ-ℕ n ' quote-free-var (starred x))
  starred {X} (n ' app t u)      = star-app {X} {n} t u
                                            (decide-head-normal t)
                                            (starred (app t u))
  starred {X} (n ' (m ' x))      = star-'' {X} {n} {m}
                                            (decide-head-normal x)
                                            (starred (m ' x))
```

The application term is dependent on whether the left term is has a variable in the quoted list as its head variable. In the case of a free variable in the head position, we cannot reduce to the internal constructor of application, so instead we use the quote constructor and recurse on both sides of the application, while if the term is in head normal form for a bound variable, we fulfill the criteria for reduction to internal application and so we recurse on both sides of the application. Finally, if the term is not in head normal form we quote a starred call, represented by r, where the reduction of quotation is blocked and the term is stuck under the quote.

```
star-app : ∀ {X} {n : ℕ} (t : Λ (X + Fin n)) → (u : Λ (X + Fin n))
         → (Σ (X + Fin n) ( λ x → head-normal x t))
              + ((x : (X + Fin n)) → ¬ (head-normal x t) )
         → (r : Λ (X + Fin n)) - spot in the context for a starred call
         → Λ X
star-app {n = n} t u (inl (free x , pr4)) r = n ' (app (starred t) (starred u))
star-app {n = n} t u (inl (bound x , pr4)) r = (app-cons (n ' starred t) (n ' starred u))
star-app {n = n} t u (inr x) r              = n ' r
```

A similar breakdown is true for the double quotation helper, with the corresponding free applied to to the decision procedure, to see if the head variable is not in the bound list of the inner quotation while being bound in the list of the outer quotation:

```
star-'' : ∀ {X} {n m : ℕ} {t : Λ ((X + Fin n) + Fin m)}
        → (Σ ((X + Fin n) + Fin m) ( λ x → head-normal x t))
             + ((x : (X + Fin n) + Fin m ) → ¬ (head-normal x t))
        → (s : Λ (X + Fin n)) - spot in the context for a starred call
        → Λ X
star-'' {X} {n} {m} {t} (inl (free (free x) , pr4)) s
  = n ' (m ' starred t)
star-'' {X} {n} {m} {t} (inl (free (bound x) , pr4)) s
  = quote-cons (numeral m)
       ((add-ℕ n m) ' join-quotes {X} n m (starred t))
star-'' {n = n} (inl (bound x , pr4)) s = n ' s
star-'' {n = n} (inr x) s               = n ' s
```

We continue by showing that the reduction relation $\rightsquigarrow$ is included in the parallel relation $\rightsquigarrow\circ$. Before doing so, we show the reflexivity of $\rightsquigarrow\circ$ for any term of λ' as a recursion down each of the constructors of ΛX:

```
- The parallel reduction relation is reflexive
refl-⇝∘ : ∀ {X} → (x : Λ X) → x ⇝∘ x
refl-⇝∘ (var x) = var∘ x
refl-⇝∘ (л x)   = л∘ x x (refl-⇝∘ x)
refl-⇝∘ (app t u) = app∘ t t u u (refl-⇝∘ t) (refl-⇝∘ u)
refl-⇝∘ (n ' x) = '∘ n x x (refl-⇝∘ x)
```

The reflexivity of $\rightsquigarrow\circ$ is used to provide the parallel reductions stipulated by the constructors of $\rightsquigarrow\circ$ translating the applications (one side will stay constant) and internal constructors (as they do not parallel reduce under regular reduction), otherwise we recurse down each of the constructors using the induction hypothesis.

```
⇝-implies-⇝∘ : ∀ {X} (x y : Λ X)
                → x ⇝ y
                → x ⇝∘ y
⇝-implies-⇝∘ .(app (л f) x) .(f [ x ]) (β f x)
  = β∘ f f x x (refl-⇝∘ f) (refl-⇝∘ x)
⇝-implies-⇝∘ .(л f) .(л f') (л⇝ f f' s)
  = л∘ f f' (⇝-implies-⇝∘ f f' s)
⇝-implies-⇝∘ .(app f x) .(app f' x) (app⇝left f f' x s)
  = app∘ f f' x x (⇝-implies-⇝∘ f f' s ) (refl-⇝∘ x)
⇝-implies-⇝∘ .(app f x) .(app f x') (app⇝right f x x' s)
  = app∘ f f x x' (refl-⇝∘ f) (⇝-implies-⇝∘ x x' s)
⇝-implies-⇝∘ .(n ' t) .(n ' t') ('⇝ n t t' s)
  = '∘ n t t' (⇝-implies-⇝∘ _ _ s)
```

```
⇝-implies-⇝∘ .(n ' (л f)) .(л-cons (succ-ℕ n ' quote-free-var f)) ('л⇝ n f)
  = 'л∘ n f f (refl-⇝∘ f)
⇝-implies-⇝∘ .(n ' var (bound j)) .(var-cons (numeral (nat-Fin n j))) ('var⇝ n j)
  = '-var∘ n j
⇝-implies-⇝∘ .(n ' app t u) .(app-cons (n ' t) (n ' u))
  ('app⇝ n t u i x)
  = '-app∘ n t t u u i x (refl-⇝∘ t) (refl-⇝∘ u)
⇝-implies-⇝∘ .(n ' (m ' t)) .(quote-cons (numeral m) (add-ℕ n m ' join-quotes n m t))
  ("⇝ m n i t x)
  = "∘ n m t t i x (refl-⇝∘ t)
```

We can now show our first non-trivial result, namely that any term (x : Λ X) will parallel reduce to the term starred x with all of the redexes being reduced. The proof requires another mutual block to define, due to the helper functions necessitated by the starred function calling the main function in certain cases.

The final proof is again a case split recursing over the constructors of the Λ datatype, though with an additional split on application terms, as starred reduces all available β-reductions, using helper lemmas for the internal application and quotation constructors to encode the internal reductions depending on head normal form.

```
mutual
  refl-star-⇝∘ : ∀ {X} (x : Λ X) → x ⇝∘ (starred x)
  refl-star-⇝∘ (var x)
    = var∘ x
  refl-star-⇝∘ (л s)
    = л∘ s (starred s) (refl-star-⇝∘ s)
  refl-star-⇝∘ (app (var x) s)
    = app∘ (var x) (var x)
           s (starred s)
        (refl-star-⇝∘ (var x))
        (refl-star-⇝∘ s)
  refl-star-⇝∘(app (л f) x)
    = β∘ f (starred f)
         x (starred x)
         (refl-star-⇝∘ f)
           (refl-star-⇝∘ x)
  refl-star-⇝∘ (app (app f x) x')
    = app∘ (app f x) (starred (app f x))
           x' (starred x')
        (refl-star-⇝∘ _)
        (refl-star-⇝∘ _)
```

```
refl-star-⇝∘ (app (n ' t) t')
  = app∘ (n ' t) _ t' _ (refl-star-⇝∘ (n ' t))
                        (refl-star-⇝∘ t')
refl-star-⇝∘ (n ' var (free x))
  = '∘ n (var (free x))
         (var (free x))
         (var∘ (free x))
refl-star-⇝∘ (n ' var (bound x))
  = '-var∘ n x
refl-star-⇝∘ (n ' (л s))
  = 'л∘ n s
       (starred s)
       (refl-star-⇝∘ s)
refl-star-⇝∘ (n ' app f x)
  = refl-star-app (decide-head-normal f)
refl-star-⇝∘ (n ' (m ' s))
  = refl-star-quote (decide-head-normal s)
```

For quoted application, case split on the decision procedure for evaluating the head normal form outlined in the previous section. In the case that the term is in head normal form for some free variable, we continue the recursive call of the application under the quotation, same as if the term is not in head normal form (inr). In the case of no normal form we case split further on the left side of the quoted application to allow for β-reductions, which are not an issue for the free case, as no л-terms are in (our notion of) head normal form. The only case in which we apply and recurse under the internal constructor is if the left side has a head variable in the quoted list:

```
refl-star-app : ∀ {X} {n : ℕ} {f x : Λ (X + Fin n)}
              → (s : ((Σ (X + Fin n) ( λ x → head-normal x f))
                      + ((x : (X + Fin n)) → ¬ (head-normal x f))))
              → n ' app f x ⇝∘ star-app {X} {n} f x s (starred (app f x))
refl-star-app (inl (free x₁ , pr4))
  = '∘ _ _ _ (app∘ _ _ _ _ (refl-star-⇝∘ _) (refl-star-⇝∘ _))
refl-star-app (inl (bound x₁ , pr4))
  = '-app∘ _ _ _ _ _ x₁ pr4
           (refl-star-⇝∘ _)
           (refl-star-⇝∘ _)
refl-star-app {f = var x₁} {x} (inr ._)
  = '∘ _ _ _ (app∘ _ _ _ _ (refl-⇝∘ _)
                           (refl-star-⇝∘ _))
refl-star-app {f = л f} {x} (inr ._)
  = '∘ _ _ _ (β∘ _ _ _ _ (refl-star-⇝∘ _)
```

```
                                    (refl-star-⇝∘ _))
refl-star-app {f = app f x'} {x} (inr ._)
  = '∘ _ _ _ (app∘ _ _ _ _ (refl-star-⇝∘ _)
                           (refl-star-⇝∘ _))
refl-star-app {f = m ' f} {x = x} (inr ._)
  = '∘ _ _ _ (app∘ _ _ _ _ (refl-star-⇝∘ _)
                           (refl-star-⇝∘ _))
```

The induction for the double quotation case is similarly case split, reducing only to the internal constructor ($''\circ$) in the case of a the term being in head normal form for a variable free in the inner list and bound in outer quoted list. Otherwise we recurse down the term with the relevant terms of the parallel reduction and inductive hypothesis (using the helper functions directly in some of the cases).

```
refl-star-quote : ∀ {X} {n m : ℕ} {t : Λ ((X + Fin n) + Fin m)}
                → (s : ((Σ ((X + Fin n) + Fin m) ( λ x → head-normal x t))
                         + ((x : ((X + Fin n) + Fin m)) → ¬ (head-normal x t))))
                → n ' (m ' t) ⇝∘ star-'' {n = n} {m = m} s (starred (m ' t))
refl-star-quote {t = t}
  (inl (free (free x) , hnf))
  = '∘ _ _ _ ('∘ _ _ _ (refl-star-⇝∘ t))
refl-star-quote {t = t}
  (inl (free (bound x) , hnf))
  = ''∘ _ _ _ _ x hnf (refl-star-⇝∘ t)
refl-star-quote {t = var (bound x₁)}
  (inl (bound x , hnf))
  = '∘ _ _ _ ('-var∘ _ x₁)
refl-star-quote {t = app t t₁}
  (inl (bound x , hnf))
  = '∘ _ _ _ (refl-star-app (decide-head-normal t))
refl-star-quote {t = n₁ ' t}
  (inl (bound x , hnf))
  = '∘ _ _ _ (refl-star-quote {m = n₁} (decide-head-normal t))
refl-star-quote {t = var (free y)} (inr x)
  = '∘ _ _ _ (refl-⇝∘ (_ ' var (free y)))
refl-star-quote {t = var (bound t)} (inr x)
  = '∘ _ _ _ ('-var∘ _ t)
refl-star-quote {t = л t} (inr x)
  = '∘ _ _ _ ('л∘ _ _ _ (refl-star-⇝∘ t))
refl-star-quote {t = app t u} (inr x)
  = '∘ _ _ _ (refl-star-app (decide-head-normal t))
refl-star-quote {t = n₁ ' t} (inr x)
  = '∘ _ _ _ (refl-star-quote (decide-head-normal t))
```

We recall that the next step outlined in the previous chapter is showing the triangle property (For all terms x, y x $\rightsquigarrow\circ$ y implies y $\rightsquigarrow\circ$ (starred x)) for the parallel reduction relation. A specific case (β-reduction) of this lemma, requires us to show that mapping and substitution distribute over the parallel reduction.

3.4.1 map over parallel reduction

Before we can show that maps distribute over parallel reductions, we need some preliminary lemmas. First, we show that we can map a function over a head normal form by recursing over the constructors of head normal form, required for the internal quotation and application.

```
map-head-normal : ∀ {X X'}
                  → (f : X → X')
                  → (s : X)
                  → (t : Λ X )
                  → head-normal s t
                  → head-normal (f s) (map f t)
map-head-normal f s .(var s) var-hn
  = var-hn
map-head-normal f s .(app t u) (app-hn t u x)
  = app-hn _ _
           (map-head-normal _ _ _ x)
map-head-normal f s .(n ' t) ('-hn n t x)
  = '-hn   _ _
           (map-head-normal _ _ _ x)
```

Many of the cases for the mapping lemma require us to transport over identities which witness the commutativity of map with single variable substitution, sum reassociation (for internal abstraction), and joining quotations. Transport (tr : (B : A → UU j) → (x y : A) → (p : Id x y) → B x → B y) is a property of the identity type which allows us to transport two terms of the same type (x y : A) as sections of a type family (B : A → UU j) given an identity between them (p : Id x y).

The first lemma can be read as "mapping f over a term t with a free variable and then substituting a single variable with the mapping of f over a different term x is equal to mapping f over the single variable substitution of x into t.

Commutativity of the single variable substitution differs significantly from the other two, as it is a special case of substitute for a single free variable. We therefore use the identity for map and substitution from the left (substitute-map-commute-$\circ$) and show that it is pointwise equal to the identity of the map and substitution from the left (substitute-map-comute).

```
sub-single-map-pointwise : ∀ {X X' B : UU}
      → (f : X → X')
      → (x : Λ X)
      → (t : X + unit)
      → Id ((substitute-single (map f x) ∘ (f ⁺)) t)
           ((map f ∘ substitute-single x) t)
sub-single-map-pointwise f x (free t) = refl
sub-single-map-pointwise f x (bound star) = refl

sub-single-map-commutes : ∀ {X X'}
  → (f : X → X')
  → (t : Λ (X + unit))
  → (x : Λ X)
  → Id ((map (f ⁺) t) [ map f x ])
       (map f (t [ x ]))
sub-single-map-commutes f t x
  = ((substitute-map-commute-∘ t (f ⁺)
                                 (substitute-single (map f x))
     · substitute-pointwise (substitute-single (map f x) ∘ (f ⁺))
                            (map f ∘ substitute-single x)
                            t
                            (sub-single-map-pointwise {B = unit} f x)))
     · substitute-map-commute (substitute-single x) f t
```

The commutativity of reassociating the coproduct, named quote-map-commute as it's used in the quotation of Λ-terms. We apply the composition lemma of map on both sides (inverted on the right) and show the pointwise equality of the compositions.

```
assoc-map-free-commute : ∀ {X X' B C : UU}
                       → (f : X → X')
                       → (x : (X + B) + C)
                       → Id (map-assoc-coprod ((f ⁺ ⁺) x))
                            ((f ⁺) (map-assoc-coprod x))
assoc-map-free-commute f (free (free x)) = refl
assoc-map-free-commute f (free (bound x)) = refl
assoc-map-free-commute f (bound x) = refl

assoc-map-commute : ∀ {X X' B C : UU}
                  → (f : X → X')
                  → (t : Λ ((X + B) + C))
                  → Id (map map-assoc-coprod (map ((f [ B ]⁺) [ C ]⁺) t))
                       (map (f [ B + C ]⁺) (map map-assoc-coprod t))
```

```
assoc-map-commute f t = (map-composition (f ⁺ ⁺) map-assoc-coprod t
                        · map-pointwise _ _ lem t)
                        · inv (map-composition map-assoc-coprod (f ⁺) t) where
  lem : ∀ x → Id (map-assoc-coprod ((f ⁺ ⁺) x)) ((f ⁺) (map-assoc-coprod x))
  lem = assoc-map-free-commute f
```

The second identity, used to show that join-quote commutes (recall that join-quote is defined using a map) As both of these are lifts (maps) of functions over two types of free variables into λ' calculus, they do not depend on any substitutions and can be shown nearly identically through reassociation and (trivial) pointwise equality.

```
join-Fin-map-free-commute : ∀ {X X' : UU} → (n m : ℕ)
                          → (f : X → X')
                          → (x : _)
                          → Id (join-Fin n m (((f [ Fin n ]⁺) [ Fin m ]⁺ ) x))
                               ((f ⁺) (join-Fin n m x))
join-Fin-map-free-commute n m f (free (free x)) = refl
join-Fin-map-free-commute n m f (free (bound x)) = refl
join-Fin-map-free-commute n m f (bound x) = refl

join-quote-map-commute : ∀ {X X'} {n m : ℕ}
  → (f : X → X')
  → (t : Λ _)
  → Id (map (join-Fin n m) (map (f ⁺ ⁺) t))
                           (map (f ⁺) (map (join-Fin n m) t))
join-quote-map-commute {n = n} {m = m} f t
  = map-composition (f ⁺ ⁺) (join-Fin n m) t
  · (map-pointwise _ _ lem t
  · inv (map-composition (join-Fin n m) (f ⁺) t))
  where
    lem : ∀ x → Id (join-Fin n m ((f ⁺ ⁺) x)) ((f ⁺) (join-Fin n m x))
    lem x = join-Fin-map-free-commute _ _ f x
```

Mapping over the terms of the basic constructors of the λ-calculus amounts to mapping over each of the subterms of the reduction with termination in the case of variables, where we apply the function to the variable.

```
map-↝∘ : ∀ {X X'} {x x' : Λ X}
         → (f : X → X')
         → x ↝∘ x'
         → map f x ↝∘ map f x'
map-↝∘ f (var∘ x) = var∘ (f x)
```

```
map-⇝∘ f (app∘ g h x x' r₁ r₂) = app∘ (map f g) (map f h)
                                   (map f x) (map f x')
                                   (map-⇝∘ f r₁)
                                   (map-⇝∘ f r₂)
map-⇝∘ f (л∘ t t' r) = л∘ (map (f ⁺) t) (map (f ⁺) t')
                          (map-⇝∘ (f ⁺) r)
map-⇝∘ f ('∘ n t t' r) = '∘ n (map (f ⁺) t) (map (f ⁺) t')
                              (map-⇝∘ (f ⁺) r)
```

Using the first of the above identities, we transport the parallel reduction of mapping over a β-reduction to recursing map down the components of the β case of parallel reduction.

```
map-⇝∘ f (β∘ g g' x x' r₁ r₂)
  = tr (λ t → map f (app (л g) x) ⇝∘ t )
       (sub-single-map-commutes _ g' x')
       (β∘ _ _ _ _ (map-⇝∘ (f ⁺) r₁)
                   (map-⇝∘ f r₂))
```

The internal variable constructor assumes free variables to map over, as indicated by the bound j term, referring to the index of the variable in the list. Using map-var, we show that mapping over the constructor is the same as applying the constructor to the mapping over its parameters (the internal representation of De Bruijn indices), which is encoded as an ap of var-cons and the commutativity of the numerals, reflecting the right hand (result) of the parallel reduction.

```
map-⇝∘ f ('-var∘ n j)
  = tr (λ x → n ' var (bound j) ⇝∘ x )
       (inv (map-var f (numeral (nat-Fin n j))
         · ap var-cons (map-numeral-fixed f (nat-Fin n j))))
       (('-var∘ n j))
```

We continue to transport over the reduction of the mapping using the same idea as above with the commutativity of map and coproduct reassociation to show that mapping over a quoted л-term will reduce to mapping over the continued reduction under the internal constructor, by applying л-cons to the witness of commutativity and rebuilding the term.

```
map-⇝∘ f ('л∘ n g g' x)
  = tr (λ x → map f (n ' (л g)) ⇝∘ x)
       (ap (λ x → л-cons (succ-ℕ n ' x))
           (assoc-map-commute f g')
```

```
    · inv (map-л f ((succ-ℕ n ' quote-free-var g')))
  ('л∘ n (map (f ⁺ ⁺) g) (map (f ⁺ ⁺) g')
         (map-⇝∘ (f ⁺ ⁺) x))
```

The quotation case is similar to the л-case, with the added complication of the internal quotation operator taking both a numeral representing the list of quoted variables, hence we use ap-binary to apply the quotation constructor to both the identity showing that mapping over numeral is "fixed" (idempotent) and the commutativity of join-quote and map. We also map over the proof of head normal form of the induction hypothesis to provide the required head normal form.

```
map-⇝∘ f ('∘ n m t t' j x x₁)
  = tr (λ x → map f (n ' (m ' t)) ⇝∘ x)
       (ap-binary quote-cons (inv (map-numeral-fixed f m))
         (ap (λ t → add-ℕ n m ' t) (join-quote-map-commute {n = n} {m = m} f t'))
           · (inv (map-' f (numeral m) _ )))
       ('∘ n m (map (f ⁺ ⁺) t) (map (f ⁺ ⁺) t') j
          (map-head-normal (f ⁺ ⁺) _ _ x)
          (map-⇝∘ (f ⁺ ⁺) x₁ ))
```

The internal application case is easier to prove, as it simply recurses the reduction under the internal quotation case. The transport uses map-app, a special case of the commutativity of map and substitution construct$_2$-substitute, along with mapping f over the witness of head normal form of the left term in the application to give a witness for the term being transported over.

```
map-⇝∘ f ('-app∘ n t t' u u' i hnf s1 s2)
  = tr (λ x → map f (n ' (app t u)) ⇝∘ x )
       (inv (map-app f (n ' t') (n ' u')))
       ('-app∘ n _ _ _ _ i (map-head-normal (f ⁺) _ _ hnf)
                           (map-⇝∘ (f ⁺) s1)
                           (map-⇝∘ (f ⁺) s2))
```

3.4.2 substitute over parallel reduction

Having proven that one can map over the parallel reduction, we define shift-variable over it, which is defined as mapping free and adding some amount of fresh free variables.

```
shift-variables-parallel : ∀ {X  V} {x x' : Λ X}
                         → x ⇝∘ x'
```

```
                                → shift-variable {V = V} x ⇝∘ shift-variable {V = V} x'
shift-variables-parallel = map-⇝∘ free
```

Substitution of free variables follows quite naturally from this, given a term encoding the pointwise reduction (pw), a term witnessing the parallel reduction of a substitution ($\phi : X \to \Lambda X'$) to ($\phi' : X \to \Lambda X'$) for all free variables, we keep the bound variables using the variable constructor and shift the free varibles.

```
\⁺-over-⇝∘ : ∀ {X X' B} {ϕ ϕ' : X → Λ X'}
           → (∀ i → ϕ i ⇝∘ ϕ' i)
           → (t : (X + B))
           → (substitute-free-variables ϕ t) ⇝∘ (substitute-free-variables ϕ' t)
\⁺-over-⇝∘ pw (free f)    = shift-variables-parallel (pw f)
\⁺-over-⇝∘ pw (bound f) = var∘ (bound f)
```

We will use the same strategy to prove the distributivity of pointwise reducible substitutions as for map, so we need to prove identities to transport the parallel reductions. The first of these is the commutativity of single variable substitution (used for defining β substitution, hence the name) and full substitution, accounting for free variables by using $\backslash^+$.

We start by composing each of the substitutions, leaving us with a pointwise identity to show. The bound variables are trivially idenitical as they are not changed under substitution. As the single variable substitution is defined using a map for free variables, we use the lemma for precomposition of substitution with map, giving a term with only substitutions, and as the single variable substitution uses the var constructor we use the witness for var being the Kleisli identity, and use a trivial witness to show their pointwise definitional equality.

```
single-substitution-commute-pointwise : ∀ {X X'}
  → (ϕ : X → Λ X')
  → (x : Λ X)
  → ∀ f
  → Id ((substitute
          (substitute-single (substitute ϕ x)) ∘ (ϕ \⁺)) f)
       ((substitute ϕ ∘ substitute-single x) f)
single-substitution-commute-pointwise ϕ x (free f) = substitute-map-commute-∘ (ϕ f) _ _
                                                   · (substitute-pointwise _ _ (ϕ f) (λ _ → refl)
                                                   · var-identity (ϕ f))
single-substitution-commute-pointwise ϕ x (bound star) = refl

single-substitution-commute : ∀ {X X'}
  → (ϕ : X → Λ X')
  → (f : Λ (X + unit))
```

```
  → (x : Λ X)
  → Id (substitute (ϕ \+) f [ substitute ϕ x ])
       (substitute ϕ (f [ x ]))
single-substitution-commute ϕ f x
  = substitute-composition (ϕ \+) _ f
    · (substitute-pointwise _ _ f (single-substitution-commute-pointwise _ _)
     · inv (substitute-composition _ ϕ f))
```

Proceeding with commutativity of substitution and reassociation of the coproduct (with the obvious expansion of substitutions). The expansions require the post- and precompositions of substitution with map (similar to above), with the pointwise lemma being trivial for all cases but the leftmost (free (free x)), for which we decompose each of the maps (using the inverse of math-composition).

```
assoc-coprod-\+ : ∀ {X X' B C : UU}
                  → (t : X → Λ X')
                  → (x : (X + B) + C )
                  → Id (((t [ (B + C) ]\+) ∘ map-assoc-coprod) x)
                       ((map map-assoc-coprod ∘ (t \+ \+)) x)
assoc-coprod-\+ t (free (free x))
  = inv (map-composition _ _ _ )
  · (inv (map-composition _ _ _))
assoc-coprod-\+ t (free (bound x)) = refl
assoc-coprod-\+ t (bound x) = refl

assoc-coprod-substitute-commute : ∀ {X X' B C : UU}
                                  → (t : X → Λ X')
                                  → (x : Λ ((X + B) + C))
                                  → Id (substitute (t [(B + C)]\+ ) (map map-assoc-coprod x))
                                       (map map-assoc-coprod (substitute (t \+ \+) x))
assoc-coprod-substitute-commute t s = (substitute-map-commute-∘ s map-assoc-coprod (t \+)
                                       · substitute-pointwise _ _ s
                                          (assoc-coprod-\+ t))
                                       · (substitute-map-commute (t \+ \+) map-assoc-coprod s)
```

The commutativity of joining quotes and substitution is proven similarly to the lemma directly above (inverted due to its use in the proof of parallel substitution). The pointwise lemma again simply composes the maps in the same way as above, though inverted, reflecting the swapped identity of the main lemma.

```
join-Fin-subt-pointwise : ∀ {X X'} {n m : ℕ}
  → (ϕ : X → Λ X')
```

```
  → (x : (X + Fin n) + Fin m)
  → Id ((map (join-Fin n m) ∘ (φ \⁺ \⁺)) x)
       (((φ \⁺) ∘ join-Fin n m) x)
join-Fin-subt-pointwise {n = n} {m = m} φ (free (free x))
  = map-composition _ _ _
  · map-composition _ _ _
join-Fin-subt-pointwise {n = n} {m = m} φ (free (bound x)) = refl
join-Fin-subt-pointwise {n = n} {m = m} φ (bound x) = refl

join-quote-substitute-commute : ∀ {X X'} {n m : ℕ}
                                → (φ : X → Λ X')
                                → (x : Λ ((X + Fin n) + Fin m))
                                → Id (map (join-Fin n m) (substitute (φ \⁺ \⁺) x))
                                     (substitute (φ \⁺) (map (join-Fin n m) x))
join-quote-substitute-commute {n = n} {m = m} φ x
  = (inv (substitute-map-commute _ _ x))
    · (substitute-pointwise _ _ x
         (join-Fin-subt-pointwise {n = n} {m = m} φ)
       · inv (substitute-map-commute-∘ x _ _))
```

As above, we must show that terms are in head normal form for the parallel reductions of the internal constructors, showing that a variable can be substituted/replaced in a head normal form. We do this by assuming that the substitution (φ : X → Λ X') is identical/equivalent to the variable constructor, the unit for the Kleisli category generated by Λ. We transport the constructor for the variable case across the identity and recurse for the others, applying the free constructor to the quotation case.

```
head-normal-replace-variable : ∀ {X X'} (x : X) (y : X') (t : Λ X)
                               → (φ : X → Λ X')
                               → head-normal x t
                               → Id (var y) (φ x)
                               → head-normal y (substitute φ t)
head-normal-replace-variable x y (var .x) φ var-hn s
  = tr (head-normal y) s var-hn
head-normal-replace-variable x y (app t t₁) φ (app-hn .t .t₁ hnf) s
  = app-hn _ _
      (head-normal-replace-variable x y t φ hnf s )
head-normal-replace-variable x y (n ' t) φ ('-hn .n .t hnf) s
  = '-hn n (substitute (φ \⁺) t)
      (head-normal-replace-variable (free x) (free y) t (φ \⁺) hnf
        (ap (map free) s))
```

To derive the substitution of a parallel reduction, we assume a pointwise parallel reduction of two substitutions for all terms (as with the lemma for \⁺) and given any other parallel

reduction, we can substitute with ϕ and ϕ' on the left and right sides of the parallel reductions. The proof method is nearly identical to the proof of mapping over a parallel reduction, β-reduction and the internal constructors requiring transport over the identities proven above. The basic constructors for the λ'-calculus are shown by reductions, using $\backslash^+$-over-$\rightsquigarrow\circ$ for quotation and abstraction.

```
substitute-⇝∘ : ∀ {X X'}
                  {ϕ ϕ' : X → Λ X'}
                  {x x' : Λ X}
                → x ⇝∘ x'
                → (∀ i → ϕ i ⇝∘ ϕ' i)
                → substitute ϕ x ⇝∘ substitute ϕ' x'
substitute-⇝∘ (var∘ x) s = s x
substitute-⇝∘ {ϕ = ϕ} {ϕ' = ϕ'} (app∘ f f' x x' x₁ x₂) s
  = app∘ (substitute ϕ f) (substitute ϕ' f')
         (substitute ϕ x) (substitute ϕ' x')
         (substitute-⇝∘ x₁ s )
         (substitute-⇝∘ x₂ s )
substitute-⇝∘ {ϕ = ϕ} {ϕ' = ϕ'} (л∘ f f' x) s
  = л∘ (substitute (ϕ \⁺) f) (substitute (ϕ' \⁺) f')
       (substitute-⇝∘ x (\⁺-over-⇝∘ {B = unit} s))
substitute-⇝∘ {ϕ = ϕ} {ϕ' = ϕ'} ('∘ n t t' x) s
  = '∘ n (substitute (ϕ \⁺) t)
         (substitute (ϕ' \⁺) t')
         (substitute-⇝∘ x (\⁺-over-⇝∘ {B = Fin n} s))
```

We transport over the right hand side of the parallel reduction, using the first of the commutativity lemmas, similarly to the mapping lemma.

```
substitute-⇝∘ {ϕ = ϕ} {ϕ' = ϕ'} (β∘ f f' x x' x₁ x₂) s
  = tr (λ s → substitute ϕ (app (л f) x) ⇝∘ s)
       (single-substitution-commute ϕ' f' x')
       (β∘ (substitute (ϕ \⁺) f) (substitute (ϕ' \⁺) f')
         (substitute ϕ x) (substitute ϕ' x')
         (substitute-⇝∘ x₁ (\⁺-over-⇝∘ s))
         (substitute-⇝∘ x₂ s))
```

The parallel reduction of substitutions into the internal variable and application constructors are both shown through another transport across an identity constructed by the commutativity of substitution and the internal constructors, shown in the section on syntax, composed with ap of the constructors themselves.

```
substitute-⇝∘ {ϕ = ϕ} {ϕ' = ϕ'} ('-var∘ n j) s
  = tr (λ x → substitute ϕ (n ' var (bound j)) ⇝∘ x)
       (inv (substitute-var ϕ' (numeral (nat-Fin n j))
         · ap var-cons (substitute-numeral ϕ' ((nat-Fin n j)))))
       ('-var∘ n j)


substitute-⇝∘ {ϕ = ϕ} {ϕ' = ϕ'} ('-app∘ n a a' b b' i hnf x x0) s
  = tr (λ x → substitute ϕ (n ' app a b ) ⇝∘ x)
       (ap-binary app-cons refl refl
         · inv (substitute-app ϕ' (n ' a') (n ' b')))
       ('-app∘ n (substitute (ϕ \+) a) (substitute (ϕ' \+) a')
                 (substitute (ϕ \+) b) (substitute (ϕ' \+) b') i
                 (head-normal-replace-variable _ _ _ _ hnf refl)
                 (substitute-⇝∘ x (\+-over-⇝∘ s))
                 (substitute-⇝∘ x0 (\+-over-⇝∘ s)))
```

The internal abstraction constructor is also transported, using the commutativity of substitution and reassociation of the product and commutativity of the internal abstraction constructor and substitution for the identity.

```
substitute-⇝∘ {ϕ = ϕ} {ϕ' = ϕ'} ('л∘ n f f' x) s
  = tr (λ x → substitute ϕ (n ' л f) ⇝∘ x)
       (ap л-cons (ap (succ-ℕ n '_)
           (inv (assoc-coprod-substitute-commute ϕ' f')))
       · inv (substitute-л ϕ' (succ-ℕ n ' quote-free-var f')))
       ('л∘ n
            (substitute (ϕ \+ \+) f)
            (substitute (ϕ' \+ \+) f')
            (substitute-⇝∘ x (\+-over-⇝∘ (\+-over-⇝∘ s))))
```

Finally, the substitution over the parallel reduction of the internal quotation constructor is shown by transporting over an identity constructed by using the commutativities of substitution and the internal quotation, internal natural numbers and joining lists of quoted variables. We also generate a head normal term in the same way as for the application operator above.

```
substitute-⇝∘ {ϕ = ϕ} {ϕ' = ϕ'} ('"∘ n m t t' i₁ hnf x0) s
  = tr (λ x → substitute ϕ (n ' (m ' t)) ⇝∘ x)
       (ap-binary quote-cons (inv (substitute-numeral ϕ' m))
         (ap (add-ℕ n m '_ ) (join-quote-substitute-commute ϕ' t'))
       · inv (substitute-quote _ (numeral m) (((add-ℕ n m) ' join-quotes n m t'))))
```

```
  ("∘ n m _ _ i₁
    (head-normal-replace-variable _ _ _ _ hnf refl)
    (substitute-⇝∘ x0 (\⁺-over-⇝∘ ( \⁺-over-⇝∘ s))))
```

To simplify the proof of the triangle property, we prove a few specialised lemmas about parallel reduction, namely that for any two parallel reductions, one with a free variable y ⇝∘ y', the single variable substitutions will reduce correspondingly. This corresponds to β-reduction, and is the main motivation for the introduction of substitute⇝∘ and map⇝∘, though the latter also allows us to prove that quoting free variables distributes over a parallel reduction in a similar way.

```
substitute-single-⇝∘ : ∀ {X} {y y' : Λ X}
                     → (y ⇝∘ y')
                     → (∀ i → substitute-single y i ⇝∘ substitute-single y' i)
substitute-single-⇝∘ s (free x) = refl-⇝∘ _
substitute-single-⇝∘ s (bound star) = s

β-substitute-⇝∘ : ∀ {X} {x x' : Λ (X + unit)} {y y' : Λ X}
                → x ⇝∘ x'
                → y ⇝∘ y'
                → x [ y ] ⇝∘ x' [ y' ]
β-substitute-⇝∘ s t = substitute-⇝∘ s (substitute-single-⇝∘ t)

quote-free-variables⇝∘ : ∀ {X} {n : ℕ} {x x' : Λ ((X + Fin n) + unit)}
  → x ⇝∘ x'
  → quote-free-var x ⇝∘ quote-free-var x'
quote-free-variables⇝∘ x = map-⇝∘ (map-assoc-coprod) x
```

Due to the construction of the internal constructors as substitutions, we can use the substitution over parallel reduction to give some very short proofs of their distributivity over any parallel reduction, using the templates discussed in the section on syntax along with the reflexivity of parallel reduction.

```
var-cons-⇝∘ : ∀ {X} {t t' : Λ X}
            → t ⇝∘ t'
            → var-cons t ⇝∘ var-cons t'
var-cons-⇝∘ {X = X} red = substitute-⇝∘ {x = var-template {X = X}}
                                       (refl-⇝∘ _ )
                                       (λ _ → red)

л-cons-⇝∘ : ∀ {X} {f f' : Λ X}
```

```
                  → (f ⇝∘ f')
                  → л-cons f ⇝∘ л-cons f'
л-cons-⇝∘ {X = X} red = substitute-⇝∘ {x = л-template {X = X}}
                                      (refl-⇝∘ _)
                                      (λ _ → red)
```

The internal constructors for application and quotation are both encoded using the binary constructor $\mathsf{construct}_2$, but differ their use. For the application term, we reduce both of the terms, using a lambda record to split on the coproduct used to define the substitution internal constructors, while the quotation term includes a numeral which is conserved over the parallel reduction, and so the lambda record stays constant in the left term, using the reflexivity of parallel reduction.

```
app-cons-⇝∘ : ∀ {X} {t t' u u' : Λ X}
              → t ⇝∘ t'
              → u ⇝∘ u'
              → app-cons t u ⇝∘ app-cons t' u'
app-cons-⇝∘ {X = X} {t = t} {t' = t'} {u = u} {u' = u'} red red₁
  = substitute-⇝∘ {ϕ = λ{(inl _) → t; (inr _) → u}}
                  {ϕ' = λ{(inl _) → t'; (inr _) → u'}}
                  {x = app-template {X = X}}
                  (refl-⇝∘ _)
                  λ{(inl _) → red; (inr _) → red₁}

quote-cons-⇝∘ : ∀ {X} {n : ℕ} {t t' : Λ X}
                → t ⇝∘ t'
  → quote-cons (numeral n) t ⇝∘ quote-cons (numeral n) t'
quote-cons-⇝∘ {X = X} {n = n} {t = t} {t' = t'} red
  = substitute-⇝∘ {ϕ = λ{(inl _) → (numeral n); (inr _) → t }}
                  {ϕ' = λ{(inl _) → (numeral n); (inr _) → t'}}
                  {x = quote-template {X = X}}
                  (refl-⇝∘ _)
                  λ{(inl _) → (refl-⇝∘ _); (inr _) → red}
```

Finally, we show that parallel reduction conserves the head normal form of a term, case-splitting on the reduction relation. On non-internal application and quotation we recurse using the induction hypothesis. As none of the internal constructors have an abstractionless head normal form, we use bound=free-absurd to derive absurdity from the identity of head normal form.

```
head-normal-⇝∘ : ∀ {X} {t t' : Λ X}
                 → (i : X)
```

```
                  → t ⇝∘ t'
                  → head-normal i t
                  → head-normal i t'
head-normal-⇝∘ i (var∘ .i) var-hn = var-hn
head-normal-⇝∘ i (app∘ .t f' .u x' red red₁) (app-hn t u hnf)
  = app-hn _ _ (head-normal-⇝∘ i red hnf )
head-normal-⇝∘ i ('∘ .n .t t' s) ('-hn n t hnf)
  = '-hn _ _ (head-normal-⇝∘ _ s hnf )
head-normal-⇝∘ i ('-app∘ .n t t' u u' i₁ hnf s s₁) ('-hn n .(app t u) (app-hn .t .u hnf₁))
  = bound=free-absurd
      (inv (head-normal-Id t _ _ hnf₁ hnf))
head-normal-⇝∘ i ('∘ .n m t t' i₁ hnf₁ s) ('-hn n .(m ' t) ('-hn .m .t hnf))
  = bound=free-absurd
      (is-injective-inl (head-normal-Id t _ _ hnf₁ hnf))
```

3.4.3 The Triangle Property

The main lemma of the confluence result is the triangle property, which (as discussed in the chapter on λ-calculus) shows that given any (parallel) reduction x ⇝∘ y, the left hand side will reduce to the "fully reduced" (with regards to any possible single step of the parallel reduction relation) term starred x.

We start by case splitting on the reduction x ⇝∘ y, and show that simplest constructors also have the simplest proofs (with the exception of β-reduction, which most of this section has been dedicated to) by recursing down the constructors of variables, application and abstraction:

```
triangle-⇝∘ : ∀ {X} {x y : Λ X}
              → x ⇝∘ y
              → y ⇝∘ (starred x)
triangle-⇝∘ (β∘ x x' f f' x₁ x₂)
  = β-substitute-⇝∘ (triangle-⇝∘ x₁)
                    (triangle-⇝∘ x₂)
triangle-⇝∘ (var∘ x) = var∘ x
triangle-⇝∘ (app∘ (var f) f' x x' s1 s2)
  = app∘ f' (var f) _ _
            (triangle-⇝∘ s1)
            (triangle-⇝∘ s2)
triangle-⇝∘ (app∘ (л f) (л f') y x' (л∘ .f .f' s1) s2)
  = β∘ f' (starred f)
```

```
        x' (starred y)
        (triangle-↝∘ s1)
        (triangle-↝∘ s2)
triangle-↝∘ (app∘ (app f f₁) f' x y s1 s2)
  = app∘ f' (starred (app f f₁)) y (starred x)
            (triangle-↝∘ s1)
            (triangle-↝∘ s2)
triangle-↝∘ (app∘ (n ' f) f' x y s1 s2)
  = app∘ f' (starred (n ' f)) y (starred x)
            (triangle-↝∘ s1)
            (triangle-↝∘ s2)
triangle-↝∘ (л∘ f f' x₁)
  = л∘ f' (starred f) (triangle-↝∘ x₁)
```

Moving on to the quotation cases, a quoted variable reduces to the internal constructor for variables if the variable term is in the list of quoted variables (bound) and otherwise recurses under the quotation constructor. Quotation of an abstraction term always reduces to the internal abstraction constructor. The final case encodes β-reduction under the quote, and uses the induction hypothesis on a β constructor for the parallel reduction.

```
triangle-↝∘ ('∘ n (var (free x)) t' red)
  = '∘ n t' (var (free x)) (triangle-↝∘ red)
triangle-↝∘ ('∘ n (var (bound x)) .(var (bound x)) (var∘ .(bound x)))
  = '-var∘ n x
triangle-↝∘ ('∘ n (л f) .(л f') (л∘ .f f' x₁))
  = 'л∘ n f' (starred f) (triangle-↝∘ x₁)
triangle-↝∘ ('∘ n (app .(л f) x) .(f' [ x' ]) (β∘ f f' .x x' s₁ s₂) )
  = ('∘ n (f' [ x' ]) ((starred f) [ (starred x) ])
        (triangle-↝∘ (β∘ f f' x x' s₁ s₂)))
```

We use another piece of Agda syntax, the with abstraction [18], which allows us to perform a case split on some term not in the context, whether or not the term is in head normal form in our case. Splitting for application under quotation, we have the case of a head normal form for free variables, meaning they are not in the list of quoted variables, so we continue to compute the application term under quotation without reducing to the the internal constructor, and conversely for a head normal form of a bound variable.

```
triangle-↝∘ ('∘ n (app f x) .(app f' x') (app∘ .f f' .x x' x₁ x₂))
  with decide-head-normal f
triangle-↝∘ ('∘ n (app f x) (app f' x') (app∘ f f' x x' s₁ s₂))
  | inl (free t , hnf)
    = '∘ n (app f' x')
```

```
                (app (starred f) (starred x))
                (app∘ f' (starred f)
                      x' (starred x)
                      (triangle-⇝∘ s₁)
                      (triangle-⇝∘ s₂))
triangle-⇝∘ ('∘ n (app f x) (app f' x') (app∘ f f' x x' s₁ s₂))
  | inl (bound t , hnf)
    = '-app∘ n f' (starred f)
             x' (starred x)
             t (head-normal-⇝∘ (bound t) s₁ hnf)
          (triangle-⇝∘ s₁) (triangle-⇝∘ s₂)
```

In the case of the application not being in head normal form, we split on the reduction of the first term of the application so as match on β-reduction, which is unnecessary if the term is in head normal form.

```
triangle-⇝∘ ('∘ n (app (var t) x) .(app f' x') (app∘ .(var t) f' x x' s₁ s₂))
  | inr not-hnf
    = '∘ n _ _ (app∘ _ _ _ _ (triangle-⇝∘ s₁) (triangle-⇝∘ s₂))
triangle-⇝∘ ('∘ n (app (л f) x) (app (л f') x')
                  (app∘ .(л f) .(л f') x x' (л∘ .f f' s₁) s₂))
  | inr not-hnf
    = '∘ n (app (л f') x')
           (starred (app (л f) x))
           (β∘ f' (starred f)
               x' (starred x)
             (triangle-⇝∘ s₁) (triangle-⇝∘ s₂))
triangle-⇝∘ ('∘ n (app (app t u) x) (app f' x')
           (app∘ .(app t u) f' x x' s₁ s₂))
  | inr not-hnf
    = '∘ n _ _ (app∘ _ _ _ _ (triangle-⇝∘ s₁) (triangle-⇝∘ s₂))
triangle-⇝∘ {_} {m ' app (n ' t) _}
           ('∘ m (app (n ' t) x) (app f' x') (app∘ .(n ' t) f' x x' s₁ s₂))
  | inr not-hnf
    = '∘ m (app f' x') (starred (app (n ' t) x))
           (app∘ _ _ _ _ (triangle-⇝∘ s₁) (triangle-⇝∘ s₂))
```

We again use the with abstraction for double quotation, splitting on whether or not the term is in normal form. Having a head normal form for variables not in the list of quoted variables (denoted by free (free x)) implies that a double quotation term cannot safely reduce to any of the internal constructors, and so both of the reductions containing them will be absurd, which we prove using the head normal forms provided by their induction hypotheses.

```
triangle-⇝∘ ('∘ n (m ' t) t' s₁)
  with decide-head-normal t
triangle-⇝∘ ('∘ n (m ' t) (m ' t') ('∘ m t t' s₁))
  | inl (free (free x) , pr4)
  = '∘ n (m ' t') (m ' starred t) ('∘ m _ _ (triangle-⇝∘ s₁))
triangle-⇝∘ ('∘ n (m ' (app t u)) _ ('-app∘ m t t' u u' i hnf s₁ s₂))
  | inl (free (free x) , app-hn .t .u pr4)
  = bound=free-absurd (head-normal-ld t (bound i) (free (free x)) hnf pr4 )
triangle-⇝∘ ('∘ n (m₁ ' (m₂ ' t)) _ ("∘ m₁ m₂ t t' i s₁ hnf))
  | inl (free (free x) , '-hn .m₂ .t pr4)
  = bound=free-absurd
      (is-injective-inl (head-normal-ld t (free (bound i)) _ s₁ pr4 ))
```

Given the witness of quoted quotation term being in head normal form for some variable in the list of quoted variables, we can reduce the to the constructor for internal quotation, but cannot do anything in the cases for quoted internal encodings, as they are encoded as abstraction terms, and definitionally do not have a λ-less head normal form, the absurdity being expressed with bound=free-absurd.

```
triangle-⇝∘ ('∘ n (m ' t) (m ' t') ('∘ m t t' x₁))
    | inl (free (bound x) , pr4)
    = "∘ n m t' (starred t) x ((head-normal-⇝∘ _ x₁ pr4)) (triangle-⇝∘ x₁)
triangle-⇝∘ ('∘ n (m ' (app t u)) _ ('-app∘ m t t' u u' i hnf x₁ x₂))
  | inl (free (bound x) , app-hn .t .u pr4)
    = bound=free-absurd (head-normal-ld t _ _ hnf pr4 )
triangle-⇝∘ ('∘ n (m₁ ' (m₂ ' t)) _ ("∘ m₁ m₂ t t' i hnf s₁))
    | inl (free (bound x) , '-hn .m₂ .t pr4)
    = bound=free-absurd ( is-injective-inl (head-normal-ld t _ _ hnf pr4 ))
triangle-⇝∘ ('∘ n (m ' t) t' s₁)
    | inl (bound x , pr4)
    = '∘ n t' (starred (m ' t)) (triangle-⇝∘ s₁)
```

In the case of no head normal form for a double quotation reduction, the calculation of starred continues to recurse under the constructor for quotation for each of the terms, terminating at the internal variable constructor (hence the refl-⇝∘), continuing under the internal abstraction constructor, not existing for the internal application (as there is no head normal form), and continuing under the internal quotation constructor.

```
triangle-⇝∘ ('∘ n (m ' t) _ ('∘ m t t' x₁))
  | inr no-hnf
  = '∘ n (m ' t') (starred (m ' t)) (triangle-⇝∘ ('∘ m _ _ x₁ ))
triangle-⇝∘ ('∘ n .(m ' (var (bound j))) _ ('-var∘ m j))
```

```
  | inr no-hnf
  = refl-↝∘ (n ' (var-cons (numeral (nat-Fin m j))))
triangle-↝∘ ('∘ m (n ' (л f)) _ ('л∘ n f f' x₁))
  | inr no-hnf
  = '∘ m _ _ (л-cons-↝∘ ('∘ (succ-ℕ n)
                            (quote-free-var f')
                            (quote-free-var (starred f))
                            (quote-free-variables↝∘ {n = n}
                            (triangle-↝∘ x₁))))
triangle-↝∘ ('∘ n (m ' (app t u)) _ ('-app∘ m t t' u u' i₁ hnf x₁ x₂))
  | inr no-hnf
  = ex-falso (no-hnf _ (app-hn _ _ hnf))
triangle-↝∘ ('∘ n (m ' (m₁ ' t)) _ ("∘ m m₁ t t' i₁ hnf x₁))
  | inr no-hnf
  = '∘ n (quote-cons (numeral m₁) ((add-ℕ m m₁) ' join-quotes m m₁ t'))
         (starred (m ' (m₁ ' t)))
         (triangle-↝∘ ("∘ m m₁ t t' i₁ hnf x₁))
```

The internal variable constructor is invariant under the starred function, and so the reflexivity of parallel reduction suffices, and we use the distributivity of quoting free variables along with a recursive to the triangle property to construct the continued reduction under the internal л-constructor.

```
triangle-↝∘ ('-var∘ n j) = refl-↝∘ (var-cons (numeral (nat-Fin n j)))
triangle-↝∘ ('л∘ n f f' red1) = л-cons-↝∘ ('∘ (succ-ℕ n)
                                              (quote-free-var f')
                                              (quote-free-var (starred f))
                                              (quote-free-variables↝∘ {n = n}
                                              (triangle-↝∘ red1)))
```

The starred function for internal application and quotation will continue the reduction beneath themselves, the proofs requiring two explicitly calculated identities witnessing the reduction of starred for the aforementioned terms. Both proofs split on the decidability of the term, for which all but the bound cases (the bound head normal of the quoted term) are absurd, and shown using either the inequality of bound and free variables or the non-existence of a head normal form.

```
head-normal-starred-app : ∀ {X} {n : ℕ} (t u : Λ (X + Fin n))
  → (s : _)
  → (x : Fin n)
  → (head-normal (bound x) t)
  → Id (star-app {n = n} t u s (starred (app t u))) (app-cons (n ' starred t) (n ' starred u))
```

```
head-normal-starred-app t u (inl (free x₁ , pr4)) x hnf
  = bound=free-absurd
      (head-normal-Id _ _ _ hnf pr4)
head-normal-starred-app t u (inl (bound x₁ , pr4)) x hnf = refl
head-normal-starred-app t u (inr x₁) x hnf = ex-falso (x₁ _ hnf)

head-normal-starred-quote : ∀ {X} {n m : ℕ} (t : Λ ((X + Fin n) + Fin m))
  → (s : _ )
  → (x : Fin n)
  → (head-normal (free (bound x)) t)
  → Id (star-'' {n = n} {m = m} {t = t} s (starred (m ' t)))
       (quote-cons (numeral m)
                   (add-ℕ n m ' map (join-Fin n m) (starred t)))
head-normal-starred-quote t (inl (free (free x₁) , pr4)) x hnf
  = bound=free-absurd
    (is-injective-inl
      (head-normal-Id _ _ _ hnf pr4))
head-normal-starred-quote t (inl (free (bound x₁) , pr4)) x hnf = refl
head-normal-starred-quote t (inl (bound x₁ , pr4)) x hnf
  = bound=free-absurd
      (head-normal-Id _ _ _ pr4 hnf)
head-normal-starred-quote t (inr y) x hnf
  = ex-falso (y (free (bound x)) hnf)
```

Which we proceed to transport over this identity, using the right hand sides of the reduction relation as the point and inverting the identities above to account for this, yielding the starred helpers.

```
triangle-↝∘ ('-app∘ n t t' u u' i₁ hnf red1 red2)
  = tr (λ y → (app-cons (n ' t') (n ' u')) ↝∘ y)
       (inv (head-normal-starred-app _ _
            (decide-head-normal t ) _ hnf ))
       (app-cons-↝∘ ('∘ n t' (starred t) (triangle-↝∘ red1))
                    ('∘ n u' (starred u) (triangle-↝∘ red2)))
triangle-↝∘ ('' n m t t' i₁ hnf red)
  = tr (λ y → (quote-cons (numeral m)
          (add-ℕ n m ' join-quotes n m t')) ↝∘ y)
       (inv (head-normal-starred-quote t (decide-head-normal t) i₁ hnf))
       (quote-cons-↝∘ {n = m}
       ('∘ (add-ℕ n m)
           (join-quotes n m t')
           (join-quotes n m (starred t))
           (map-↝∘
```

```
      (join-Fin n m)
      (triangle-↝∘ red))))
```

3.4.4 Confluence

Before we can show the confluence of λ', we must show the inclusion of parallel reduction in the reflexive, transitive closure of the reduction relation. To do so, we use the explicit transitivity of ↝* and the inclusion of parallel reduction, along with the pass-through of the relevant constructors under the reflexive transitive closure (discussed in the previous section) and use of the induction hypothesis for recursion.

```
↝∘-implies-↝* : ∀ {X} {x y : Λ X} → x ↝∘ y → x ↝* y
↝∘-implies-↝* (β∘ f f' x x' s s₁)
  = ↝*-is-transitive (app-↝*-right
                       (↝∘-implies-↝* s₁))
                       (β-↝* (↝∘-implies-↝* s))
↝∘-implies-↝* (var∘ x) = ↝*-refl
↝∘-implies-↝* (app∘ f f' x x' s s₁)
  = ↝*-is-transitive (app-↝*-left
                       (↝∘-implies-↝* s))
                     (app-↝*-right
                       (↝∘-implies-↝* s₁))
↝∘-implies-↝* (л∘ f f' s)
  = л-↝* (↝∘-implies-↝* s)
↝∘-implies-↝* ('∘ n t t' s)
  = '-↝* (↝∘-implies-↝* s)
↝∘-implies-↝* ('-var∘ n j)
  = var-cons-↝* j
↝∘-implies-↝* ('л∘ n f f' s)
  = ↝*-is-transitive ('-↝* {n = n}
                       (л-↝* (↝∘-implies-↝* s)))
                           (л-cons-↝* f')
↝∘-implies-↝* ('-app∘ n t t' u u' i x s s₁)
  = ↝*-is-transitive
      ('-↝* {n = n}
        (↝*-is-transitive
          (app-↝*-left
            (↝∘-implies-↝* s))
              (app-↝*-right
                (↝∘-implies-↝* s₁))))
```

```
          (app-cons-↝* i _ _ (head-normal-↝∘ _ s x))
↝∘-implies-↝* ("∘ n m t t' i x s)
  = ↝*-is-transitive ('-↝* ('-↝* (↝∘-implies-↝* s)))
                                (quote-cons-↝* m n i t'
                                  (head-normal-↝∘ (free (bound i)) s x))
```

We can finally show confluence, encoding it as the diamond property, which states that for a term and two reductions of that term (t ↝ u and t ↝ u'), there exists (Σ) a term for which v to which u and u' reduce to in zero or more reduction steps (u ↝* v × u' ↝* v).

The term is given explicitly as starred t, and we construct the terms of ↝* by first using the inclusion of regular reduction in parallel reduction ↝-implies-↝∘, use the triangle property to construct the term u ↝∘ starred t, and finally using the inclusion of the parallel reduction in the reflexive transitive closure to promote the term to u ↝* starred t.

```
'Λ-is-confluent : ∀ {X} {u u' : Λ X}
  → (t : Λ X)
  → t ↝ u
  → t ↝ u'
  → Σ _ (λ v → u ↝* v × u' ↝* v)
'Λ-is-confluent t red₀ red₁ = starred t , ↝∘-implies-↝*
                                            (triangle-↝∘
                                              (↝-implies-↝∘ _ _ red₀)),
                                          ↝∘-implies-↝*
                                            (triangle-↝∘
                                            (↝-implies-↝∘ _ _ red₁))
```

Chapter 4

Conclusion

4.1 Discussion

Confluence, as with Calculating a possible reduction, using named variables for clarity:

$$\begin{aligned}(\lambda x.x)(\langle\rangle\ '\ ((\lambda y.y)\ (\lambda z.z))) &\rightsquigarrow (\langle\rangle\ '\ (\lambda y.y)\ (\lambda z.z)) \\ &\rightsquigarrow (\langle\rangle\ '\ (\lambda z.z)) \\ &\rightsquigarrow (\langle z\rangle\ '\ \texttt{lam}(z)) \\ &\rightsquigarrow (\langle z\rangle\ '\ \texttt{lam}(\texttt{numeral}(0))\end{aligned}$$

We may also have performed this reduction by

$$\begin{aligned}(\lambda x.x)(\langle\rangle\ '\ ((\lambda y.y)\ (\lambda z.z))) &\rightsquigarrow (\lambda x.x)(\langle\rangle\ '\ (\lambda z.z)) \\ &\rightsquigarrow (\langle\rangle\ '\ (\lambda z.z)) \\ &\rightsquigarrow \ldots\end{aligned}$$

A more interesting property of confluence is the ability to uniquely define β-equivalence, which is usually given as the smallest equivalence relation including β-reduction (and internal constructors, in our case). If we instead define a relation $a \equiv b$ by a and b having a common reduct[1], which is trivially reflexive and symmetric, but not obviously transitive.

Transitivity is guaranteed by confluence of the reduction relation, as for any $a \equiv b$ and $b \equiv c$, we have common reducts s and t respectively, confluence giving a term u which s

[1] A common left-hand term in the transitive, reflexive closure

and t reduce to, so a and c reduce to as well. As an example of this, $(\lambda y.y)(\langle\rangle\ '\ (\lambda z.z))$ is β-equivalent to $(\lambda x.x)(\langle\rangle\ '\ ((\lambda y.y)\ (\lambda z.z)))$, with the common reduct $(\langle z\rangle\ '\ \texttt{lam}(z))$.

Confluence is a key property for an implementation of λ' as a programming language, as it guarantees any reduction strategy will produce the same result, meaning that the implementer is free to chose the most efficient strategy/-ies for their purpose, knowing that they will all produce the same term. The use of De Bruijn indices for our formalisation was partly motivated by such practical considerations, as they are often used to implement substitution, as per the intention of their invention[8].

The implementation of such a language would require terms (programs) acting on the internal representation of λ', similar to the variable counter from Section 2.2. as an example, a term which many authors refer to as a "self-reducer", which is usually defined as a term $T : \Lambda$ such that for any fully quoted term $Q : \Lambda X$, the application TQ normalises Q while preserving the "type" defined by the internal constructors. An internalisation of substitution and the procedure for identifying head normal forms would be required as prerequisites.

We also lack a treatment of η-conversion, the rule $\lambda x.\ f\ x \equiv f$ where f does not contain a free occurence of x, often present in functional programming languages. The issue appears to be with the notion of head normal form we use for β-reduction, as an η-reduced term $\lambda x.f \rightsquigarrow f$ is not necessarily equivalent when converted into the internal constructors, eg. $\texttt{lam app}\ (\texttt{var}(f))\ (\texttt{var}(x)) \not\equiv \texttt{var}(f))$. One either needs further refinement of the notion of head normal form, or some other normal form ensuring the confluence of η-reduction.

4.2 Further Work

By giving terms which fulfill the formation, introduction, elimination and computation rules of various types (such as $\lambda x.x$ for the unit type), thereby typing λ', we may extend the hypothetical programming language to a typed quotation calculus, similar to [5] and [11]. For this to be useful, normalisation (all terms have a normal form) must be proven.

Another aim of further development for λ'-calculus would be to give it a denotational semantic interpretation. This may come in the form of some domain-theoretic construction in the same vein as the influencial model of the untyped λ-calculus given by Dana Scott [3] which initiated the field. It may also come in the form of a Category with Families[7], one of the categorical models of Martin-Löf Type Theory which may also be applied to weaker type theories, which the untyped λ-calculus could be considered (all terms being of a single type).

The confluence of λ', with its particular requirement of head normal form, raises interesting questions about which classes of functions can be self-represented. A number of terms, such

as those which do not reduce to a normal form, are trivially excluded from this. Bauer[4], commenting on Brown and Palsberg, shows that if the representation is injective and β-normal, a self-interpreter will exist, while also showing how they possess more properties such as the quotation commuting with substitution. He calls for a rigorous and structural definition of self-interpretation, which we have not found during our literature review for this thesis, and so might be an open problem.

We also wish to give an appropriate notion of equivalence for the quoted terms, for which simple β-equivalence will not suffice. As a simple example, we would wish for the expansion of a fixed point combinator to be identified, so $F\ fix\ F \equiv F\ (F\ fix\ F)$ in some sense. Böhm trees are being investigated as a possible solution for this.

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. MIT Electrical Engineering and Computer Science. MIT Press, London, England, 2 edition, July 1996.

[2] Marius Bancila. *Template Metaprogramming with C++*. Packt Publishing, Birmingham, England, October 2022.

[3] Hendrik P Barendregt. *The Lambda Calculus — Its Syntax and Semantics*, volume 103. North-Holland Amsterdam, 1984.

[4] A. Bauer. On self-interpreters for system t and other typed λ-calculi. In *On Self-Interpreters for System T and Other Typed λ-Calculi*, 2017.

[5] Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for f-omega. *SIGPLAN Not.*, 51(1):5–17, jan 2016. ISSN 0362-1340. doi: 10.1145/2914770.2837623.
URL: `https://doi.org/10.1145/2914770.2837623`.

[6] Altenkirch T. ; Ghani N. ; Hancock P. ; Mcbride C. and Morris P. Indexed containers. *Journal of Functional Programming*, 25:e5, 2015. doi: 10.1017/S095679681500009X.

[7] Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. In *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*, pages 135–180. Springer International Publishing, Cham, 2021.

[8] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. ISSN 1385-7258. doi: https://doi.org/10.1016/1385-7258(72)90034-0.

[9] Agda development team. Agda, 2021.
URL: `https://wiki.portal.chalmers.se/agda/pmwiki.php`.

[10] Håkon Gylterud. Quote operations.
URL: `https://hakon.gylterud.net/research/quote/`.

[11] Barry Jay. Self-quotation in a typed, intensional lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 336:207–222, 2018. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2018.03.024. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

[12] Brian W Kernighan and Dennis M Ritchie. *C Programming Language, 2nd Edition*. Prentice-Hall software series. Prentice Hall, Old Tappan, NJ, September 1988.

[13] S. C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340 – 353, 1936. doi: 10.1215/S0012-7094-36-00227-2.
URL: https://doi.org/10.1215/S0012-7094-36-00227-2.

[14] András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022. doi: 10.1145/3547641.
URL: https://doi.org/10.1145/3547641.

[15] Yannis Lilis and Anthony Savidis. A survey of metaprogramming languages. *ACM Comput. Surv.*, 52(6), oct 2019. ISSN 0360-0300. doi: 10.1145/3354584.
URL: https://doi.org/10.1145/3354584.

[16] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer Science+Business Media, 1998.

[17] Simon Marlow et al. Haskell 2010 language report. 2010.

[18] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. doi: 10.1017/S0956796803004829.

[19] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1992. doi: 10.1017/S0956796800000423.

[20] Erik Palmgren. On Equality of Objects in Categories in Constructive Type Theory. In Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, editors, *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*, volume 104 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:7, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-071-2. doi: 10.4230/LIPIcs.TYPES.2017.7.
URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TYPES.2017.7.

[21] E. Rijke. *Introduction to Homotopy Type Theory*. 2022.
URL: https://arxiv.org/abs/2212.11082.

[22] Egbert Rijke, Elisabeth Bonnevier, Jonathan Prieto-Cubides, Fredrik Bakke, and others. The agda-unimath library, 2023.
URL: https://github.com/UniMath/agda-unimath/.

[23] Terese. *Term rewriting systems*. Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge, England, March 2003.

[24] A. M. Turing. Computability and λ-definability. *J. Symb. Log.*, 2(4):153–163, December 1937.

Appendix A

Omitted Proofs

Identities Relating to Internal Constructors

From Section 3.2

```
map-s : ∀ {A B}
      → (f : A → B)
      → (t : Λ A)
      → Id (map f (s-cons t)) (s-cons (map f t))
map-s {A} = construct₁-map s-template

map-numeral-fixed : ∀ {X X'}
                  → (f : X → X')
                  → (n : ℕ)
                  → Id (map f (numeral n))
                       (numeral n)
map-numeral-fixed f zero-ℕ = refl
map-numeral-fixed f (succ-ℕ n) = map-s f _ · ap s-cons (map-numeral-fixed f n)


map-var : ∀ {X X'}
        → (f : X → X')
        → (t : Λ X)
        → Id (map f (var-cons t))
             (var-cons (map f t))
map-var = construct₁-map var-template

map-л : ∀ {X X'}
      → (f : X → X')
      → ∀ g
```

```
        → Id (map f (л-cons g))
             (л-cons (map f g))
map-л = construct₁-map л-template

map-app : ∀ {X X'}
        → (f : X → X')
        → ∀ g x
        → Id (map f (app-cons g x))
             (app-cons (map f g) (map f x))
map-app = construct₂-map app-template

map-' : ∀ {X X'}
      → (f : X → X')
      → ∀ n t
      → Id (map f (quote-cons n t ))
           (quote-cons (map f n) (map f t))
map-' = construct₂-map quote-template
```

Pass-through of constructors under ⇝*

From Section 3.3.4

```
app-⇝*-right : ∀ {X}
  → {f x x' : Λ X}
  → x ⇝* x'
  → app f x ⇝* app f x'
app-⇝*-right ⇝*-refl                = ⇝*-refl
app-⇝*-right (⇝*-trans x ⇝*-refl)   = ⇝*-trans (app⇝right _ _ _ x)
                                               ⇝*-refl
app-⇝*-right (⇝*-trans x (⇝*-trans x₁ red)) = ⇝*-trans (app⇝right _ _ _ x)
                                                       (app-⇝*-right
                                                         (⇝*-trans x₁ red))

л-⇝* : ∀ {X}
     → {x x' : Λ (X + unit)}
     → x ⇝* x'
     → (л x) ⇝* (л x')
л-⇝* ⇝*-refl                = ⇝*-refl
л-⇝* (⇝*-trans x ⇝*-refl) = ⇝*-trans (л⇝ _ _ x)
                                      ⇝*-refl
л-⇝* (⇝*-trans x (⇝*-trans x₁ s)) = ⇝*-trans (л⇝ _ _ x)
```

```
                                                (л-⇝*
                                                  (⇝*-trans x₁ s))

'-⇝* : ∀ {X n}
       → {t t' : Λ (X + Fin n)}
       → (t ⇝* t')
       → (n ' t) ⇝* (n ' t')
'-⇝* ⇝*-refl                                = ⇝*-refl
'-⇝* {n = n} (⇝*-trans x ⇝*-refl) = ⇝*-trans ('⇝ n _ _ x) ⇝*-refl
'-⇝* {n = n} (⇝*-trans x (⇝*-trans x₁ s)) = ⇝*-trans ('⇝ n _ _ x)
                                                     ('-⇝* (⇝*-trans x₁ s))

β-⇝* : ∀ {X} {f f' : Λ (X + unit)} {x : Λ X}
       → (f ⇝* f')
       → (app (л f) x) ⇝* (f' [ x ])
β-⇝* ⇝*-refl                                = ⇝*-trans (β _ _) ⇝*-refl
β-⇝* {f = f} {f' = f'} (⇝*-trans x ⇝*-refl) = ⇝*-trans (app⇝left _ _ _
                                                          (л⇝ _ _ x))
                                                        (β-⇝* ⇝*-refl)
β-⇝* (⇝*-trans x (⇝*-trans x₁ s)) = ⇝*-trans (app⇝left _ _ _
                                                (л⇝ _ _ x))
                                              (β-⇝* (⇝*-trans x₁ s))
```