UNIVERSITY OF
OXFORD

# A formalization of the $\lambda$-$Y$ calculus


Samuel Balco

GTC

University of Oxford

Supervised by Faris Abou-Saleh, Luke Ong and Steven Ramsay



Submitted in partial completion of the

*MSc in Computer Science*

Trinity 2016

# Acknowledgements

Say thanks to whoever listened to your rants for 2 months

# Abstract


Higher order model checking (HOMC), has been intensively studied in recent years (C.-H. L. Ong (2006), Kobayashi (2013), Ramsay, Neatherway, and Ong (2014), Tsukada and Ong (2014)). A common approach to studying HOMC is through higher order recursion schemes (HORS).

Recently, it was shown that $\lambda$-$Y$ calculus with intersection-types can be used as an alternative to HORS, when studying higher order model checking. Whilst the theory of HORS and $\lambda$-$Y$ has been formalized "on paper", there has been little done in mechanizing this theory in a fully formal setting of a theorem prover.

The aim of this project was to start such a formalization, by first mechanizing the $\lambda$-$Y$ calculus along with the proof of confluence for simple types. This served as a benchmark for choosing the implementation specifics, such as the binder representation strategy as well as the implementation language.

The formalization of the terms of the calculus along with the proof of confluence served as a benchmark for comparing the chosen mechanization approaches. The best mechanization was then extended with the formalization of intersection types for the $\lambda$-$Y$ calculus, along with the proofs of subject invariance.

# Contents

# 1. Introduction

## 1.1 Motivation

Formal verification of software is essential in a lot of safety critical systems in the industry and has been a field of active research in computer science. One of the main approaches to verification is model checking, wherein a system specification is checked against certain correctness properties, by generating a model of the system, encoding the desired correctness property as a logical formula and then exhaustively checking whether the given formula is satisfiable in the model of the system. Big advances in model checking of $1^{st}$ order (imperative) programs have been made, with techniques like abstraction refinement and SAT/SMT-solver use, allowing scalability.
Aspects of functional programming, such as anonymous/$\lambda$ functions have gained prominence in mainstream languages, such as C++ or JavaScript and functional languages like Scala, F# or Haskell have garnered wider interest. With growing interest in using functional programming, interest in verifying higher-order functional programs has also grown. Current approaches to formal verification of such programs usually involve the use of (automatic) theorem provers, which usually require a lot of user interaction and as a result have not managed to scale as well as model checking in the $1^{st}$ order setting.
Using type systems is another way to ensure program safety, but using expressive-enough types often requires explicit type annotations, since type checking/inference usually becomes undecidable, as is the case for dependent-type systems. Simpler type systems, where type inference is decidable, can instead prove too coarse, i.e. the required properties are difficult if not impossible to capture in such type systems.
In recent years, advances in higher order model checking (HOMC) have been made (C.-H. L. Ong (2006), Kobayashi (2013), Ramsay, Neatherway, and Ong (2014), Tsukada and Ong (2014)), but whilst a lot of theory has been developed for HOMC, there has been little done in implementing/mechanizing these results in a fully formal setting of a theorem prover.

## 1.2 Aims

The aim of this project is to make a start of mechanizing the proofs underpinning HOMC approaches using type-checking of higher-order recursion schemes, by formalizing and formally proving certain key properties about the $\lambda$-$Y$ calculus with an intersection-type system (Clairambault and Murawski (2013), Tsukada and Ong (2014)), which can be used to study HOMC as an alternative to higher order recursion schemes (HORS).

The first part of this work focuses on the mechanization aspect of the simply typed $\lambda$-$Y$ calculus in a theorem prover, in a fashion similar to the PoplMark challenge, by exploring different encodings of binders in a theorem prover and also the use of different theorem provers. The reason why we chose to do such a comparison was to evaluate and chose the best mechanization approach and implementation language for the $\lambda$-$Y$ calculus, as there is little information available concerning the merits and disadvantages of different implementation approaches of $\lambda$-$Y$ or indeed just the (simply typed) $\lambda$-calculus. The comparison of different mechanizations focuses on the engineering choices and formalization overheads which result from translating the informal definitions into a fully-formal setting of a theorem prover. The project is roughly split into two main parts, with the first part exploring and evaluating the different formalizations of the simply-typed $\lambda$-$Y$ calculus together with the proof of the Church Rosser Theorem. The reason why we chose to formalize the Church Rosser theorem was to to test the implementation of a non-trivial, but simple enough proof in a fully formal setting.

The second part focuses on implementing the intersection-type system for the $\lambda$-$Y$ calculus and formalizing the proof of subject invariance for this type system. The formalization and engineering choices made in the implementation of the intersection-type system reflect the survey and analysis of the different possible choices of mechanization, explored in the first part of the project.

All the code described in this project can be found in the git repository at: https://github.com/goodlyrottenapple/lamYcalc.

## 1.3 Main Achievements

**TODO: Expand on the points eventually....leaving for the end**

- Formalization of the simply typed $\lambda$-$Y$ calculus and proofs of confluence in Isabelle, using both Nominal sets and locally nameless encoding of binders.
- Formalization of the simply typed $\lambda$-$Y$ calculus and proofs of confluence in Agda, using a locally nameless encoding of binders
- Analysis and comparison of binder encodings
- Comparison of Agda and Isabelle
- Formalization of an intersection-type system for the $\lambda$-$Y$ calculus and proof of subject invariance for intersection-types

## 1.4 Dissertation Structure

The dissertation has 7 chapters and is split into three conceptual parts. The first part (chapters 1-3) describes the domain and the goals of this project, the second part (chapters 4 and 5) is a comparison of several mechanizations of the simply typed $\lambda$-$Y$ calculus and the third part (chapter 6) discusses the intersection typing and associated proofs.

Chapter 1 is an overview of the aims and achievements of this project.

Chapter 2 gives an introduction to the $\lambda$-$Y$ calculus, together with an overview of the proof of confluence (Church Rosser). The chapter also introduces intersection types and discusses an important aspect of a fully formal mechanization of a $\lambda$-calculus, namely the treatment of binders

in a fully formal setting of a theorem prover.
Chapter 3 introduces the methodology used for comparing the different mechanizations discussed in later chapters.
Chapter 4 compares two mechanizations of the $\lambda$-$Y$ calculus (nominal and locally nameless), focused around the treatment of binders. The comparison looks at the overall length and structure of the two formalizations, as well as using specific instances of the same definitions/lemmas across the two mechanizations, to illustrate the advantages and disadvantages of both mechanizations.
Chapter 5 details the differences between using Isabelle and Agda for formalizing the $\lambda$-$Y$ calculus.
Chapter 6 discusses the implementation details of intersection types for the $\lambda$-$Y$ calculus and the various engineering choices that were made in order to simplify the ensuing proof of subject invariance.
**??** summarizes the outcomes of the project and details possible further work.

# 2. Background

## 2.1 Binders

When describing the (untyped) $\lambda$-calculus on paper, the terms of the $\lambda$-calculus are usually inductively defined in the following way:

$$t ::= x \mid tt \mid \lambda x.t \text{ where } x \in Var$$

This definition of terms yields an induction/recursion principle, which can be used to define functions over the $\lambda$-terms by structural recursion and prove properties about the $\lambda$-terms using structural induction (recursion and induction being two sides of the same coin).
However, whilst the definition above describes valid terms of the $\lambda$-calculus, there are implicit assumptions one makes about the terms, namely, the $x$ in the $\lambda x.t$ case appears bound in $t$. This means that while $x$ and $y$ might be distinct terms of the $\lambda$-calculus (i.e. $x \neq y$), $\lambda x.x$ and $\lambda y.y$ represent the same term, as $x$ and $y$ are bound by the $\lambda$. Without the notion of $\alpha$-equivalence of terms, one cannot prove any properties of terms involving bound variables, such as saying that $\lambda x.x \equiv \lambda y.y$.

In an informal setting, reasoning with $\alpha$-equivalence of terms is often very implicit, however in a formal setting of theorem provers, having an inductive definition of "raw" *lambda*-terms, which are not *alpha*-equivalent, yet reasoning about $\alpha$-equivalent $\lambda$-terms poses certain challenges.
One of the main problems is the fact that the inductive/recursive definition does not easily lift to *alpha*-equivalent terms. Take a trivial example of a function on raw terms, which checks whether a variable appears bound in a given $\lambda$-term. Clearly, such function is well formed for "raw" terms, but does not work (or even make sense) for $\alpha$-equivalent terms.
Conversely, there are informal definitions over $\alpha$-equivalent terms, which are not straight-forward to define over raw terms. Take the usual definition of substitution, defined over $\alpha$-equivalent terms, which actually relies on this fact in the following case:

$$(\lambda y'.s')[t/x] \equiv \lambda y'.(s'[t/x]) \text{ assuming } y' \not\equiv x \text{ and } y' \notin FV(t)$$

Here in the $\lambda$ case, it is assumed that a given $\lambda$-term $\lambda y.s$ can always be swapped out for an alpha equivalent term $\lambda y'.s'$, such that $y'$ satisfies the side condition. The assumption that a bound variable can be swapped out for a "fresh" one to avoid name clashes is often referred to as the Barendregt Variable Convention.

The direct approach of defining "raw" terms and an additional notion of $\alpha$-equivalence introduces a lot of overhead when defining functions, as one either has to use the recursive principles for "raw" terms and then show that the function lifts to the $\alpha$-equivalent terms or define functions

on *alpha*-equivalence classes and prove that it is well-founded, without being able to rely on the structurally inductive principles that one gets "for free" with the "raw" terms.
Because of this, the usual informal representation of the $\lambda$-calculus is rarely used in a fully formal setting.

To mitigate the overheads of a fully formal definition of the $\lambda$-calculus, we want to have an encoding of the $\lambda$-terms, which includes the notion of $\alpha$-equivalence whilst being inductively defined, giving us the inductive/recursive principles for *alpha*-equivalent terms directly. This can be achieved in several different ways. In general, there are two main approaches taken in a rigorous formalization of the terms of the lambda calculus, namely the concrete approaches and the higher-order approaches, both described in some detail below.

### 2.1.1 Concrete approaches

The concrete or first-order approaches usually encode variables using names (like strings or natural numbers). Encoding of terms and capture-avoiding substitution must be encoded explicitly. A survey by B. Aydemir et al. (2008) details three main groups of concrete approaches, found in formalizations of the $\lambda$-calculus in the literature:

#### 2.1.1.1 Named

This approach generally defines terms in much the same way as the informal inductive definition given above. Using a functional language, such as Haskell or ML, such a definition might look like this:

```
datatype trm =
  Var name
| App trm trm
| Lam name trm
```

As was mentioned before, defining "raw" terms and the notion of $\alpha$-equivalence of "raw" terms separately carries a lot of overhead in a theorem prover and is therefore not favored.

To obtain an inductive definition of $\lambda$-terms with a built in notion of $\alpha$-equivalence, one can instead use nominal sets. The theory of nominal sets captures the notion of bound variables and freshness, as it is based around the notion of having properties invariant in name permutation.
The nominal package in Isabelle provides tools to automatically define terms with binders, which generate inductive definitions of $\alpha$-equivalent terms. Using nominal sets in Isabelle results in a definition of terms which looks very similar to the informal presentation of the lambda calculus:

```
nominal_datatype trm =
  Var name
| App trm trm
| Lam x::name l::trm  binds x in l
```

Most importantly, this definition allows one to define functions over $\alpha$-equivalent terms using structural induction. The nominal package also provides freshness lemmas and a strengthened induction principle with name freshness for terms involving binders.

#### 2.1.1.2 Nameless/de Bruijn

Using a named representation of the lambda calculus in a fully formal setting can be inconvenient when dealing with bound variables. For example, substitution, as described in the introduction, with its side-condition of freshness of $y$ in $x$ and $t$ is not structurally recursive on “raw” terms, but rather requires well-founded recursion over $\alpha$-equivalence classes of terms. To avoid this problem in the definition of substitution, the terms of the lambda calculus can be encoded using de Bruijn indices:

```
datatype trm =
  Var nat
| App trm trm
| Lam trm
```

This representation of terms uses indices instead of named variables. The indices are natural numbers, which encode an occurrence of a variable in a $\lambda$-term. For bound variables, the index indicates which $\lambda$ it refers to, by encoding the number of $\lambda$-binders that are in the scope between the index and the $\lambda$-binder the variable corresponds to.

**Example 2.1.** The term $\lambda x.\lambda y.yx$ will be represented as $\lambda\ \lambda\ 0\ 1$. Here, 0 stands for $y$, as there are no binders in scope between itself and the $\lambda$ it corresponds to, and 1 corresponds to $x$, as there is one $\lambda$-binder in scope. To encode free variables, one simply choses an index greater than the number of $\lambda$'s currently in scope, for example, $\lambda\ 4$.

To see that this representation of $\lambda$-terms is isomorphic to the usual named definition, we can define two function $f$ and $g$, which translate the named representation to de Bruijn notation and vice versa. More precisely, since we are dealing with $\alpha$-equivalence classes, its is an isomorphism between these that we can formalize.

To make things easier, we consider a representation of named terms, where we map named variables, $x, y, z, \dots$ to indexed variables $x_1, x_2, x_3, \dots$. Then, the mapping from named terms to de Bruijn term is given by $f$, which we define in terms of an auxiliary function $e$:

$$
\begin{aligned}
e_k^m(x_n) &= \begin{cases} k - m(x_n) - 1 & x_n \in \text{dom } m \\ k + n & \textit{otherwise} \end{cases} \\
e_k^m(uv) &= e_k^m(u)\ e_k^m(v) \\
e_k^m(\lambda x_n.u) &= \lambda\ e_{k+1}^{m \oplus (x_n, k)}(u)
\end{aligned}
$$

Then $f(t) \equiv e_0^{\varnothing}(t)$

The function $e$ takes two additional parameters, $k$ and $m$. $k$ keeps track of the scope from the root of the term and $m$ is a map from bound variables to the levels they were bound at. In the variable case, if $x_n$ appears in $m$, it is a bound variable, and it's index can be calculated by taking the difference between the current index and the index $m(x_k)$, at which the variable was bound. If $x_n$ is not in $m$, then the variable is encoded by adding the current level $k$ to $n$.
In the abstraction case, $x_n$ is added to $m$ with the current level $k$, possibly overshadowing a previous

binding of the same variable at a different level (like in $\lambda x_1.(\lambda x_1.x_1)$) and $k$ is incremented, going into the body of the abstraction.

The function $g$, taking de Bruijn terms to named terms is a little more tricky. We need to replace indices encoding free variables (those that have a value greater than or equal to $k$, where $k$ is the number of binders in scope) with named variables, such that for every index $n$, we substitute $x_m$, where $m = n - k$, without capturing these free variables.

We need two auxiliary functions to define $g$:

$$\begin{aligned}
h_k^b(n) &= \begin{cases} x_{n-k} & n \geq k \\ x_{k+b-n-1} & otherwise \end{cases} \\
h_k^b(uv) &= h_k^b(u)\ h_k^b(v) \\
h_k^b(\lambda u) &= \lambda x_{k+b}.\ h_{k+1}^b(u)
\end{aligned}$$

$$\begin{aligned}
\Diamond_k(n) &= \begin{cases} n-k & n \geq k \\ 0 & otherwise \end{cases} \\
\Diamond_k(uv) &= \mathbf{max}\ (\Diamond_k(u),\ \Diamond_k(v)) \\
\Diamond_k(\lambda u) &= \Diamond_{k+1}(u)
\end{aligned}$$

The function $g$ is then defined as $g(t) \equiv h_0^{\Diamond_0(t)+1}(t)$. As mentioned above, the complicated definition has to do with avoiding free variable capture. A term like $\lambda(\lambda\ 2)$ intuitively represents a named $\lambda$-term with two bound variables and a free variable $x_0$ according to the definition above. If we started giving the bound variables names in a naive way, starting from $x_0$, we would end up with a term $\lambda x_0.(\lambda x_1.x_0)$, which is obviously not the term we had in mind, as $x_0$ is no longer a free variable. To ensure we start naming the bound variables in such a way as to avoid this situation, we use $\Diamond$ to compute the maximal value of any free variable in the given term, and then start naming bound variables with an index one higher than the value returned by $\Diamond$.

As one quickly notices, a term like $\lambda x.x$ and $\lambda y.y$ have a single unique representation as a de Bruijn term $\lambda\ 0$. Indeed, since there are no named variables in a de Bruijn term, there is only one way to represent any $\lambda$-term, and the notion of $\alpha$-equivalence is no longer relevant. We thus get around our problem of having an inductive principle and $\alpha$-equivalent terms, by having a representation of $\lambda$-terms where every $\alpha$-equivalence class of $\lambda$-terms has a single representative term in the de Bruijn notation.

In their comparison between named vs. nameless/de Bruijn representations of $\lambda$-terms, Berghofer and Urban (2006) give details about the definition of substitution, which no longer needs the variable convention and can therefore be defined using primitive structural recursion.
The main disadvantage of using de Bruijn indices is the relative unreadability of both the terms and the formulation of properties about these terms. For instance, take the substitution lemma, which in the named setting would be stated as:

$$\text{If } x \neq y \text{ and } x \notin FV(L)\text{, then } M[N/x][L/y] \equiv M[L/y][N[L/y]/x].$$

In de Bruijn notation, the statement of this lemma becomes:

$$\text{For all indices } i, j \text{ with } i \leq j,\ M[N/i][L/j] = M[L/j+1][N[L/j-i]/i]$$

Clearly, the first version of this lemma is much more intuitive.

#### 2.1.1.3 Locally Nameless

The locally nameless approach to binders is a mix of the two previous approaches. Whilst a named representation uses variables for both free and bound variables and the nameless encoding uses de Bruijn indices in both cases as well, a locally nameless encoding distinguishes between the two types of variables.
Free variables are represented by names, much like in the named version, and bound variables are encoded using de Bruijn indices. By using de Bruijn indices for bound variables, we again obtain an inductive definition of terms which are already *alpha*-equivalent.

While closed terms, like $\lambda x.x$ and $\lambda y.y$ are represented as de Bruijn terms, the term $\lambda x.xz$ and $\lambda x.xz$ are encoded as $\lambda\ 0z$. The following definition captures the syntax of the locally nameless terms:

```
datatype ptrm =
  Fvar name
  BVar nat
| App trm trm
| Lam trm
```

Note however, that this definition doesn't quite fit the notion of $\lambda$-terms, since a `pterm` like `(BVar 1)` does not represent a $\lambda$-term, since bound variables can only appear in the context of a lambda, such as in `(Lam (BVar 1))`.
The advantage of using a locally nameless definition of $\lambda$-terms is a better readability of such terms, compared to equivalent de Bruijn terms. Another advantage is the fact that definitions of functions and reasoning about properties of these terms is much closer to the informal setting.

### 2.1.2 Higher-Order approaches

Unlike concrete approaches to formalizing the lambda calculus, where the notion of binding and substitution is defined explicitly in the host language, higher-order formalizations use the function space of the implementation language, which handles binding. HOAS, or higher-order abstract syntax (F. Pfenning and Elliott 1988, Harper, Honsell, and Plotkin (1993)), is a framework for defining logics based on the simply typed lambda calculus. A form of HOAS, introduced by Harper, Honsell, and Plotkin (1993), called the Logical Framework (LF) has been implemented as Twelf by Frank Pfenning and Schürmann (1999), which has been previously used to encode the $\lambda$-calculus.
Using HOAS for encoding the $\lambda$-calculus comes down to encoding binders using the meta-language binders. This way, the definitions of capture avoiding substitution or notion of $\alpha$-equivalence are offloaded onto the meta-language. As an example, take the following definition of terms of the $\lambda$-calculus in Haskell:

```
data Term where
  Var :: Int -> Term
  App :: Term -> Term -> Term
  Lam :: (Term -> Term) -> Term
```

This definition avoids the need for explicitly defining substitution, because it encodes a $\lambda$-term as a Haskell function `(Term -> Term)`, relying on Haskell's internal substitution and notion of $\alpha$-equivalence. As with the de Bruijn and locally nameless representations, this encoding gives us inductively defined terms with a built in notion of $\alpha$-equivalence.
However, using HOAS only works if the notion of $\alpha$-equivalence and substitution of the meta-language coincide with these notions in the object-language.

## 2.2 Simple types

The simple types presented throughout this work (except for Chapter 6) are often referred to as simple types *a la Curry*, where a simply typed $\lambda$-term is a triple $(\Gamma, M, \sigma)$ s.t. $\Gamma \vdash M : \sigma$, where $\Gamma$ is the typing context, $M$ is a term of the untyped $\lambda$-calculus and $\sigma$ is a simple type.

**Definition 2.1.** [Simple-type assignment]

$$(var)\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad (app)\ \frac{\Gamma \vdash u : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash uv : B}$$

$$(abs)\ \frac{x : A, \Gamma \vdash m : B}{\Gamma \vdash \lambda.m : A \to B} \qquad (Y)\ \frac{}{\Gamma \vdash Y_A : (A \to A) \to A}$$

Such a term is deemed valid, if one can construct a typing tree from the given type and typing context.

**Example 2.2.** Take the following simply typed term $\{y : \tau\} \vdash \lambda x.xy : (\tau \to \varphi) \to \varphi$. To show that this is a well-typed $\lambda$-term, we construct the following typing tree:

$$(abs)\ \dfrac{(app)\ \dfrac{(var)\ \dfrac{}{\{x : \tau \to \varphi,\ y : \tau\} \vdash x : \tau \to \varphi} \qquad (var)\ \dfrac{}{\{x : \tau \to \varphi,\ y : \tau\} \vdash y : \tau}}{\{x : \tau \to \varphi,\ y : \tau\} \vdash xy : \varphi}}{\{y : \tau\} \vdash \lambda x.xy : (\tau \to \varphi) \to \varphi}$$

In the untyped $\lambda$-calculus, simple types and $\lambda$-terms are completely separate, brought together only through the typing relation $\vdash$ in the case of simple types *a la Curry*. The definition of $\lambda$-$Y$ terms, however, is dependent on the simple types in the case of the $Y$ constants, which are indexed by simple types. When talking about the $\lambda$-$Y$ calculus, we tend to conflate the "untyped" $\lambda$-$Y$ terms, which are just the terms defined in Definition 2.2, with the "typed" $\lambda$-$Y$ terms, which are simply-typed terms *a la Curry* of the form $\Gamma \vdash M : \sigma$, where $M$ is an "untyped" $\lambda$-$Y$ term. Thus, results about the $\lambda$-$Y$ calculus in this work are in fact results about the "typed" $\lambda$-$Y$ calculus.
However, the proofs of the Church Rosser theorem, as presented in the next section, use the untyped definition of $\beta$-reduction. Whilst it is possible to define a typed version of $\beta$-reduction, it turned out to be much easier to first prove the Church Rosser theorem for the so called "untyped" $\lambda$-$Y$ calculus and the additionally restrict this result to only well-types $\lambda$-$Y$ terms.
Thus, the definition of the Church Rosser Theorem, formulated for the $\lambda$-$Y$ calculus, is the following one:

**Theorem 2.1.** [Church Rosser]

$$\Gamma \vdash M : \sigma \land M \Rightarrow_Y^* M' \land M \Rightarrow_Y^* M'' \implies \exists M'''.\ M' \Rightarrow_Y^* M''' \land M'' \Rightarrow_Y^* M''' \land \Gamma \vdash M''' : \sigma$$

In order to prove this typed version of the Church Rosser Theorem, we need to prove an additional result of subject reduction for $\lambda$-$Y$ calculus, namely:

**Theorem 2.2.** [Subject reduction for $\Rightarrow_Y^*$]

$$\Gamma \vdash M : \sigma \land M \Rightarrow_Y^* M' \implies \Gamma \vdash M' : \sigma$$

## 2.3 $\lambda$-$Y$ calculus

Originally, the field of higher order model checking mainly involved studying higher order recursion schemes (HORS), but more recently, exploring the $\lambda$-$Y$ calculus, which is an extension of the simply typed $\lambda$-calculus, in the context of HOMC has gained traction (Clairambault and Murawski (2013)). We therefore present the $\lambda$-$Y$ calculus, along with the proofs of the Church Rosser theorem and the formalization of intersection types for the $\lambda$-$Y$ calculus, as the basis for formalizing the theory of HOMC.

### 2.3.1 Definitions

The first part of this project focuses on formalizing the simply typed $\lambda$-$Y$ calculus and the proof of confluence for this calculus (proof of the Church Rosser Theorem is sometimes also referred to as proof of confluence). The usual/informal definition of the $\lambda$-$Y$ terms and the simple types are given below:

> **Definition 2.2.** [$\lambda$-$Y$ types and terms]
>
> The set of simple types $\sigma$ is built up inductively form the $\mathsf{o}$ constant and the arrow type $\rightarrow$. Let *Var* be a countably infinite set of atoms in the definition of the set of $\lambda$-terms $M$:
>
> $$\begin{aligned} \sigma &::= \mathsf{o} \mid \sigma \rightarrow \sigma \\ M &::= x \mid MM \mid \lambda x.M \mid Y_\sigma \text{ where } x \in Var \end{aligned}$$

The $\lambda$-$Y$ calculus differs from the simply typed $\lambda$-calculus only in the addition of the $Y$ constant family, indexed at every simple type $\sigma$, where the (simple) type of a $Y_A$ constant (indexed with the type $A$) is $(A \rightarrow A) \rightarrow A$. The usual definition of $\beta$-reduction is then augmented with the $(Y)$ rule (this is the typed version of the rule):

$$(Y)\ \frac{\Gamma \vdash M : \sigma \rightarrow \sigma}{\Gamma \vdash Y_\sigma M \Rightarrow_Y M(Y_\sigma M) : \sigma}$$

In essence, the $Y$ rule allows (some) well-typed recursive definitions over simply typed $\lambda$-terms.

> **Example 2.3.** Take for example the term $\lambda x.x$, commonly referred to as the *identity*. The *identity* term can be given a type $\sigma \rightarrow \sigma$ for any simple type $\sigma$. We can therefore perform the following (well-typed) reduction in the $\lambda$-$Y$ calculus:
>
> $$Y_\sigma(\lambda x.x) \Rightarrow_Y (\lambda x.x)(Y_\sigma(\lambda x.x))$$

The typed version of the rule illustrates the restricted version of recursion clearly, since a recursive "$Y$-reduction" will only occur if the term $M$ in $Y_\sigma M$ has the matching type $\sigma \rightarrow \sigma$ (to $Y_\sigma$'s type $(\sigma \rightarrow \sigma) \rightarrow \sigma$), as in the example above.

### 2.3.2 Church-Rosser Theorem

The Church-Rosser Theorem states that the $\beta$-reduction of the $\lambda$-calculus is confluent, that is, the reflexive-transitive closure of the $\beta$-reduction has the *diamond property*, i.e. $\mathbf{dp}(\Rightarrow_Y^*)$, where:

> **Definition 2.3.** $[\mathbf{dp}(R)]$
>
> A binary relation $R$ has the *diamond property*, i.e. $\mathbf{dp}(R)$, iff
>
> $$\forall a, b, c.\ aRb \wedge aRc \implies \exists d.\ bRd \wedge cRd$$

The proof of confluence of $\Rightarrow_Y$, the $\beta Y$-reduction defined as the standard $\beta$-reduction with the addition of the aforementioned $(Y)$ rule, formalized in this project, follows a variation of the Tait-Martin-Löf Proof originally described in Takahashi (1995) (specifically using the notes by R. Pollack (1995)). To show why following this proof over the traditional proof is beneficial, we first give a high level overview of how the usual proof proceeds.

#### 2.3.2.1 Overview

In the traditional proof of the Church Rosser theorem, we define a new reduction relation, called the *parallel* $\beta$-reduction ($\gg$), which, unlike the "plain" $\beta$-reduction satisfies the *diamond property* (note that we are talking about the "single step" $\beta$-reduction and not the reflexive transitive closure). Once we prove the *diamond property* for $\gg$, the proof of $\mathbf{dp}(\gg^*)$ follows easily. The reason why we prove $\mathbf{dp}(\gg)$ in the first place is because the reflexive-transitive closure of $\gg$ coincides with the reflexive transitive closure of $\Rightarrow_Y$ and it is much easier to prove $\mathbf{dp}(\gg)$ than trying to prove $\mathbf{dp}(\Rightarrow_Y^*)$ directly. The usual proof of the *diamond property* for $\gg$ involves a double induction on the shape of the two parallel $\beta$-reductions from $M$ to $P$ and $Q$, where we try to show that the following diamond always exists, that is, given any reductions $M \gg P$ and $M \gg Q$, there is some $M'$ s.t. $P \gg M'$ and $Q \gg M'$:

$$\begin{tikzcd}
 & M \arrow[dl, "\gg"'] \arrow[dr, "\gg"] & \\
P \arrow[dr, dashed, "\gg"'] & & Q \arrow[dl, dashed, "\gg"] \\
 & M' &
\end{tikzcd}$$

Figure 2.1: The diamond property of $\gg$, visualized

The Takahashi (1995) proof simplifies this proof by eliminating the need to do simultaneous induction on the $M \gg P$ and $M \gg Q$ reductions. This is done by introducing another reduction, referred to as the *maximal parallel* $\beta$-reduction ($\ggg$). The idea of using $\ggg$ is to show that for every term $M$ there is a reduct term $M_{max}$ s.t. $M \ggg M_{max}$ and that any $M'$, s.t. $M \gg M'$, also reduces to $M_{max}$. We can then separate the "diamond" diagram above into two instances of the following triangle, where $M'$ from the previous diagram is $M_{max}$:

$$
\begin{tikzcd}
 & M \arrow[dl, "\gg"'] \arrow[dd, "\ggg"] \\
M' \arrow[dr, dashed, "\gg"'] & \\
 & M_{max}
\end{tikzcd}
$$

Figure 2.2: The proof of dp$(\gg)$ is split into two instances of this triangle

#### 2.3.2.2 Parallel $\beta Y$-reduction

Having described the high-level overview of the classical proof and the reason for following the Takahashi (1995) proof, we now present some of the major lemmas in more detail.
Firstly, we give the definition of *parallel $\beta Y$-reduction* $\gg$ formulated for the terms of the $\lambda$-$Y$ calculus, which allows simultaneous reduction of multiple parts of a term:

**Definition 2.4.** $[\gg]$

$$
(refl)\ \frac{}{x \gg x} \qquad (refl_Y)\ \frac{}{Y_\sigma \gg Y_\sigma} \qquad (app)\ \frac{M \gg M' \qquad N \gg N'}{MN \gg M'N'}
$$

$$
(abs)\ \frac{M \gg M'}{\lambda x.M \gg \lambda x.M'} \qquad (\beta)\ \frac{M \gg M' \qquad N \gg N'}{(\lambda x.M)N \gg M'[N'/x]} \qquad (Y)\ \frac{M \gg M'}{Y_\sigma M \gg M'(Y_\sigma M')}
$$

The most basic difference between the normal $\beta$-reduction and *parallel* $\beta Y$-reduction is the $(refl)/(refl_Y)$ rule, where $x \gg x$, for example, is a valid reduction, but we have $x \not\Rightarrow_Y x$ for the normal $\beta Y$-reduction ($x \Rightarrow_Y^* x$ is valid, since $\Rightarrow_Y^*$ is the reflexive transitive closure of $\Rightarrow_Y$). In the example below, $(refl^*)$ is a derived rule $\forall M.\ M \gg M$ (see Lemma 4.1):

**Example 2.4.** Another example where the two reductions differ is the simultaneous reduction of multiple sub-terms. *Parallel* reduction, unlike $\Rightarrow_Y$, allows the reduction of the term $((\lambda xy.x)z)(\lambda x.x)y$ to $(\lambda y.z)y$, by simultaneously reducing the two sub-terms $(\lambda xy.x)z$ and $(\lambda x.x)y$ to $\lambda y.z$ and $y$ respectively:

$$
(app)\ \frac{(\beta)\ \dfrac{(refl^*)\ \dfrac{}{\lambda xy.x \gg \lambda xy.x} \qquad (refl)\ \dfrac{}{z \gg z}}{(\lambda xy.x)z \gg \lambda y.z} \qquad (\beta)\ \dfrac{(refl^*)\ \dfrac{}{\lambda x.x \gg \lambda x.x} \qquad (refl)\ \dfrac{}{y \gg y}}{(\lambda x.x)y \gg y}}{((\lambda xy.x)z)(\lambda x.x)y \gg (\lambda y.z)y}
$$

When we try to construct a similar tree for $\beta$-reduction, we can clearly see that the only two rules we can use are $(red_L)$ or $(red_R)$. We can thus only perform the right-side or the left side reduction of the two sub-terms, but not both.

Having described the intuition behind the *parallel* $\beta$-reduction, we proceed to define the *maximum* *parallel reduction* $\ggg$, which contracts all redexes in a given term with a single step:

**Definition 2.5.** [$\ggg$]

$$(refl)\ \frac{}{x \ggg x} \qquad (refl_Y)\ \frac{}{Y_\sigma \ggg Y_\sigma} \qquad (app)\ \frac{M \ggg M' \qquad N \ggg N'}{MN \ggg M'N'}\ (M \text{ is not a } \lambda \text{ or } Y)$$

$$(abs)\ \frac{M \ggg M'}{\lambda x.M \ggg \lambda x.M'} \qquad (\beta)\ \frac{M \ggg M' \qquad N \ggg N'}{(\lambda x.M)N \ggg M'[N'/x]} \qquad (Y)\ \frac{M \ggg M'}{Y_\sigma M \ggg M'(Y_\sigma M')}$$

This relation only differs from $\gg$ in the $(app)$ rule, which can only be applied if $M$ is not a $\lambda$ or $Y$ term.

**Example 2.5.** To demonstrate the difference between $\gg$ and $\ggg$, we take a look at the term $(\lambda xy.x)((\lambda x.x)z)$.
Whilst $(\lambda xy.x)((\lambda x.x)z) \gg (\lambda xy.x)z$ or $(\lambda xy.x)((\lambda x.x)z) \gg \lambda y.z$ (amongst others) are valid reductions, the reduction $(\lambda xy.x)((\lambda x.x)z) \ggg (\lambda xy.x)z$ is not valid.
To see why this is the case, we observe that the last rule applied in the derivation tree must have been the $(app)$ rule, since we see that a reduction on the sub-term $(\lambda x.x)z \ggg z$ occurs:

$$(app)\ \frac{\dfrac{\vdots}{\lambda xy.x \ggg \lambda xy.x} \qquad \dfrac{\vdots}{(\lambda x.x)z \ggg z}}{(\lambda xy.x)(\lambda x.x)z \ggg (\lambda xy.x)z}\ (\lambda xy.x \text{ is not a } \lambda \text{ or } Y)$$

However, this clearly could not happen, because $\lambda xy.x$ is in fact a $\lambda$-term.

To prove $\mathbf{dp}(\gg)$, we first show that there always exists a term $M_{max}$ for every term $M$, where $M \ggg M_{max}$ is the maximal parallel reduction which contracts all redexes in $M$:

**Lemma 2.1.** $\forall M. \exists M_{max}. M \ggg M_{max}$

*Proof.* By induction on M. □

Finally, we show that any parallel reduction $M \gg M'$ can be "closed" by reducing to the term $M_{max}$ where all redexes have been contracted (as seen in Figure 2.2):

**Lemma 2.2.** $\forall M, M', M_{max}. M \ggg M_{max} \wedge M \gg M' \implies M' \gg M_{max}$

*Proof.* Omitted. Can be found on p. 8 of the R. Pollack (1995) notes. □

**Lemma 2.3.** $\mathbf{dp}(\gg)$

*Proof.* We can now prove $\mathbf{dp}(\gg)$ by simply applying Lemma 2.2 twice, namely for any term $M$ there is an $M_{max}$ s.t. $M \ggg M_{max}$ (by Lemma 2.1) and for any $M', M''$ where $M \gg M'$ and $M \gg M''$, it follows by two applications of Lemma 2.2 that $M' \gg M_{max}$ and $M'' \gg M_{max}$. □

## 2.4 Intersection types

For the formalization of intersection types, we initially chose a strict intersection-type system, presented in the Bakel (2003) notes. Intersection types, as classically presented in Barendregt, Dekkers, and Statman (2013) as $\lambda_\cap^{BCD}$, extend simple types by adding a conjunction to the definition of types:

**Definition 2.6.** [$\lambda_\cap^{BCD}$ types]

$$\mathcal{T} ::= \varphi \mid \mathcal{T} \rightsquigarrow \mathcal{T} \mid \mathcal{T} \cap \mathcal{T}$$

We restrict ourselves to a version of intersection types often called *strict intersection types*. *Strict intersection types* are a restriction on $\lambda_\cap^{BCD}$ types, where an intersection of types can only appear on the left side of an "arrow" type:

**Definition 2.7.** [Strict intersection types]

In the definition below, $\varphi$ is a constant (analogous to the constant **o**, introduced for the simple types in Definition 2.2).

$$\begin{aligned} \mathcal{T}_s &::= \varphi \mid \mathcal{T} \rightsquigarrow \mathcal{T}_s \\ \mathcal{T} &::= (\mathcal{T}_s \cap \ldots \cap \mathcal{T}_s) \end{aligned}$$

The following conventions for intersection types are adopted throughout this section; $\omega$ stands for the empty intersection and we write $\bigcap_{\underline{n}} \tau_i$ for the type $\tau_1 \cap \ldots \cap \tau_n$. We also define a subtype relation $\subseteq$ for intersection types, which intuitively capture the idea of one intersection of types being a subset of another, where we think of $\tau_1 \cap \ldots \cap \tau_i$ as a finite set $\{\tau_1, \ldots, \tau_i\}$, wherein $\subseteq$ for intersection types corresponds to subset inclusion e.g. $\tau \subseteq \tau \cap \psi$ because $\{\tau\} \subseteq \{\tau, \psi\}$.

*Remark.* The reason for defining the subset relation in this way, rather than taking the usual view of $\tau \cap \varphi \leq \tau$, was due the implementation of intersection types in Agda. Since intersection types $\mathcal{T}$ ended up being defined as lists of strict types $\mathcal{T}_s$ (the definition of lists in Agda included the notion of list inclusion $\in$ and by extension the $\subseteq$ relation), the above convention seemed more natural.

The formal definition of this relation is given below:

**Definition 2.8.** [$\subseteq$]

This relation is the least pre-order on intersection types s.t.:

$$\begin{gathered} \forall\, i \in \underline{n}.\ \ \tau_i \subseteq \bigcap\nolimits_{\underline{n}} \tau_i \\ \forall\, i \in \underline{n}.\ \ \tau_i \subseteq \tau \implies \bigcap\nolimits_{\underline{n}} \tau_i \subseteq \tau \\ \rho \subseteq \psi \wedge \tau \subseteq \mu \implies \psi \rightsquigarrow \tau \subseteq \rho \rightsquigarrow \mu \end{gathered}$$

(This relation is equivalent the $\leq$ relation, defined in R. Pollack (1995) notes, i.e. $\tau \leq \psi \equiv \psi \subseteq \tau$.)

In this presentation, $\lambda$-$Y$ terms are typed with the strict types $\mathcal{T}_s$ only. Much like the simple types, presented in the previous sections, an intersection-typing judgment is a triple $\Gamma, M, \tau$, written as

$\Gamma \Vdash M : \tau$, where $\Gamma$ is the intersection-type context, similar in construction to the simple typing context, $M$ is a $\lambda$-$Y$ term and $\tau$ is a strict intersection type $\mathcal{T}_s$.
The definition of the intersection-typing system, like the $\subseteq$ relation, has also been adapted from the typing system found in the R. Pollack (1995) notes, by adding the typing rule for the $Y$ constants:

**Definition 2.9.** [Intersection-type assignment]

$$(var)\ \frac{x : \bigcap_{\underline{n}} \tau_i \in \Gamma \qquad \tau \subseteq \bigcap_{\underline{n}} \tau_i}{\Gamma \Vdash x : \tau} \qquad (app)\ \frac{\Gamma \Vdash M : \bigcap_{\underline{n}} \tau_i \leadsto \tau \qquad \forall\, i \in \underline{n}.\ \Gamma \Vdash N : \tau_i}{\Gamma \Vdash MN : \tau}$$

$$(abs)\ \frac{x : \bigcap_{\underline{n}} \tau_i, \Gamma \Vdash M : \tau}{\Gamma \Vdash \lambda x.M : \bigcap_{\underline{n}} \tau_i \leadsto \tau}$$

$$(Y)\ \frac{}{\Gamma \Vdash Y_\sigma : (\bigcap_{\underline{n}} \tau_i \leadsto \tau_1 \cap \ldots \cap \bigcap_{\underline{n}} \tau_i \leadsto \tau_i) \leadsto \tau_j}\ (j \in \underline{n})$$

This is the initial definition, used as a basis for the mechanization, discussed in Chapter 6. Due to different obstacles in the formalization of the subject invariance proofs, this definition, along with the definition of intersection types was amended several times. The reasons for these changes are documented in Chapter 6.
The definition above also assumes that the context $\Gamma$ is *well-formed*:

**Definition 2.10.** [Well-formed intersection-type context]

Assuming that $\Gamma$ is a finite list, consisting of pairs of atoms *Var* and intersection types $\mathcal{T}$, $\Gamma$ is a *well-formed* context iff:

$$(nil)\ \frac{}{\textsf{Wf-ICtxt}\ [\,]} \qquad (cons)\ \frac{x \notin \text{dom}\ \Gamma \qquad \textsf{Wf-ICtxt}\ \Gamma}{\textsf{Wf-ICtxt}\ (x : \bigcap \tau_i, \Gamma)}$$

# 3. Methodology

The idea of formalizing a functional language in multiple theorem provers and objectively assessing the merits and pitfalls of the different formalizations is definitely not a new idea. The most well known attempt to do so on a larger scale is the PoplMark challenge, proposed in the "Mechanized Metatheory for the Masses: The PoplMark Challenge" paper by B. E. Aydemir et al. (2005). This paper prompted several formalizations of the benchmark typed $\lambda$-calculus, proposed by the authors of the challenge, in multiple theorem provers, such as Coq, Isabelle, Matita or Twelf. However, to the best of our knowledge, there has been no published follow-up work, drawing conclusions about the aptitude of different mechanizations, which would be useful in deciding the best mechanization approach to take in formalizing the $\lambda$-$Y$ calculus.
Whilst this project does not aim to answer the same question as the original challenge, namely:

> "How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine- checked proofs?" (B. E. Aydemir et al. (2005))

It draws inspiration from the criteria for the "benchmark mechanization", specified by the challenge, to find the best mechanization approach as well as the right set of tools for our purpose of effectively mechanizing the theory underpinning HOMC.

Our comparison proceeded in two stages of elimination, where the first stage was a comparison of the two chosen mechanizations of binders for the $\lambda$-$Y$ calculus (Chapter 4), namely nominal set and locally nameless representations of binders. The main reason for the fairly narrow selection of only two binder mechanizations, was the limited time available for this project. In order to at least partially achieve the goal of mechanizing the intersection type theory for the $\lambda$-$Y$ calculus, we decided to cut down the number of comparisons to the two (seemingly) most popular binder mechanizations (chosen by word of mouth and literature review of the field).
After comparing and choosing the optimal mechanization of binders, the next chapter then goes on to compare this mechanization in two different theorem provers, Isabelle and Agda.
The "winning" theorem prover from this round was then used to formalize intersection-types and prove subject invariance.

## 3.1 Evaluation criteria

The PoplMark challenge stated three main criteria for evaluating the submitted mechanizations of the benchmark calculus:

- Mechanization/implementation overheads
- Technology transparency
- Cost of entry

This project focuses mainly on the two criteria of mechanization overheads and technology transparency, since the focus of our comparison is to chose the best mechanization and theorem prover to use for implementing intersection types for the $\lambda$-$Y$ calculus, rather than asses the viability of theorem provers in general, which was the original goal of the PoplMark challenge. These criteria are described in greater detail below:

### 3.1.1 Technology transparency

Technology transparency, within the context of this work, is mostly concerned with the presentation of the theory inside a proof assistant, such as Isabelle or Agda. Whilst there is no direct measure of transparency, per se, it is almost always immediately obvious which presentation is more transparent, when one is presented with examples. This work makes a case for transparency, or the lack thereof, by providing side-by-side snippets from different mechanizations of the same theory.

> **Example 3.1.** As an example, we take the two different (though not completely distinct) styles of writing proofs in Isabelle, namely using apply-style proofs or the Isar proof language. First, to demonstrate the Isar proof language and showcase the technology transparency it affords, we take the proof that a square of an odd number is itself odd[1] and present the mechanized version in Isar:
>
> > **Lemma 3.1.** [The square of an odd number is also odd]
> >
> > *Proof.* By definition, if $n$ is an odd integer, it can be expressed as
> >
> > $$n = 2k + 1$$
> >
> > for some integer $k$. Thus
> >
> > $$\begin{aligned} n^2 &= (2k+1)^2 \\ &= (2k+1)(2k+1) \\ &= 4k^2 + 2k + 2k + 1 \\ &= 4k^2 + 4k + 1 \\ &= 2(2k^2 + 2k) + 1. \end{aligned}$$
> >
> > Since $2k^2 + 2k$ is an integer, $n^2$ is also odd.
> >
> > □
>
> Now, the same (albeit slightly simplified) proof is presented using the Isar language:

```
lemma sq_odd:
  fixes n and odd :: "nat ⇒ bool"
  defines "odd x ≡ ∃k. x = 2 * k + 1"
  assumes "odd n"
  shows "odd (n*n)"
proof -
  from assms obtain k where n_def: "n = 2 * k + 1"
    unfolding odd_def by auto
  then have "n * n = (2 * k + 1) * (2 * k + 1)" by simp
  then have "n * n = (4 * k * k) + (4 * k) + 1" by simp
  hence     "n * n = 2 * ((2 * k * k) + (2 * k)) + 1" by simp
  thus "odd (n * n)" unfolding odd_def by blast
qed
```

Clearly, this mechanized proof reads much like the rigorous paper proof that precedes it. When the same proof is presented using the "apply" style proof in Isabelle, it is immediately apparent that it is much less transparent, as we obfuscate the natural flow of the informal proof, hiding most of the reasoning in automation (the last line `by simp+`):

```
lemma sq_odd: "∧ n ::nat. (∃k. n = 2 * k + 1) ⇒ ∃k. n*n = 2 * k + 1"
apply (erule_tac P="λk. n = 2 * k + 1" in exE)
apply (rule_tac x="(2 * x * x) + (2 * x)" in exI)
apply (rule_tac s="(2 * x + 1) * (2 * x + 1)" in subst)
by simp+
```

(Note that the whole apply script can in fact just be substituted by the single line command: `by (auto, presburger)`)

The example given above demonstrates, that transparency is a comparative measure, as it depends directly on some point of reference. As is also apparent from the example, transparency can often come at a cost of brevity. The reason why apply-style proofs exist and are used, even though Isar proofs are generally regarded as the better alternative, is the fact that they can be significantly faster to write, as they are a lot less verbose. Of course, relying more on automation, these proofs naturally tend to be harder to follow. However, much like in an informal setting, where we rarely write proofs in a completely rigorous detail, especially those which are "uninteresting" from point of the whole theory, so the different styles of proofs are used for different proofs. The short, "boring" ones are often written using apply-style scripts, whereas longer more interesting lemmas use the Isar language, to make the reasoning intuitive, i.e. transparent.
This trade-off brings us to the second criterion, namely the mechanization overheads.

### 3.1.2 Mechanization/implementation overheads

When talking about mechanization overheads, we usually mean the additional theory needed to translate the informal theory we reason about on paper into the fully formal setting of a theorem prover.

[1] The proof was copied from https://en.wikipedia.org/wiki/Direct_proof

**Example 3.2.** To demonstrate what we mean by this, we will take the definition of intersection types and its implmentation in Agda (further discussed in Section 6.1). Taking the Definition 2.7 as a starting point, namely defining intersection types as:

$$\mathcal{T}_s ::= \varphi \mid \mathcal{T} \rightsquigarrow \mathcal{T}_s$$
$$\mathcal{T} ::= (\mathcal{T}_s \cap \ldots \cap \mathcal{T}_s)$$

we translate the strict types $\mathcal{T}_s$ to Agda in a straightforward way, since we only need to translate$\mathcal{T}_s$ into the style of a GADT (generalized algebraic datatype) definition:

```
data Tₛ where
  φ : Tₛ
  _~>_ : (s : T) -> (t : Tₛ) -> Tₛ
```

*Remark.* The definition above is perhaps more obvious, when $\mathcal{T}_s$ is presented inductively as:

$$\frac{}{\varphi \in \mathcal{T}_s} \qquad \frac{\tau \in \mathcal{T} \qquad \psi \in \mathcal{T}_s}{\tau \rightsquigarrow \psi \in \mathcal{T}_s}$$

The informal definition of $\mathcal{T}$, however, is slightly more complicated, since intuitively, $\mathcal{T}_s \cap \ldots \cap \mathcal{T}_s$ represents a finite set of elements of $\mathcal{T}_s$. We can describe the set of intersection terms $\mathcal{T}$ with the following inductive definition:

$$\frac{\{\tau_1, \ldots, \tau_n\} \subset \mathcal{T}_s}{\tau_1 \cap \ldots \cap \tau_n \in \mathcal{T}}$$

In order to encode this definition in Agda, we will have to rely on some definition of a finite set (since the rule above assumes knowledge of finite sets and the subset relation $\subset$ in its precondition).

Whilst the notion of a finite set is so trivial, we rarely bother axiomatizing it, Agda does not actually know about finite sets by default and its standard library only includes the definition of finite sets of natural numbers. We can instead use lists to "simulate" finite sets, as they are similar in many regards, i.e. the Agda implementation of lists includes the notion of subset inclusion for lists, so that one can write a proof of $[1, 2] \subseteq [2, 2, 1]$ easily:

```
data T where
  ∩ : List Tₛ -> T
```

Whilst this definition is now largely equivalent to the informal inductive definition, we have lost quite a bit of transparency as a result. Consider the strict type $\tau \cap \psi \rightsquigarrow \tau$, is written as `∩ (τ :: ψ :: []) ~> τ` in Agda. We can improve thigs somewhat by getting rid of the pointless constructor `∩`, by merging the two definitions of $\mathcal{T}$ and $\mathcal{T}_s$ into a single definition, namely:

```
data Tₛ where
  φ : Tₛ
  _~>_ : (s : List Tₛ) -> (t : Tₛ) -> Tₛ
```

*Remark.* This definition now corresponds to the merging of the two previously given inductive definitions of $\mathcal{T}$ and $\mathcal{T}_s$:

$$\frac{}{\varphi \in \mathcal{T}_s} \qquad \frac{\{\tau_1, \dots, \tau_n\} \subset \mathcal{T}_s \qquad \psi \in \mathcal{T}_s}{\tau_1 \cap \dots \cap \tau_n \leadsto \psi \in \mathcal{T}_s}$$

Now, $\tau \cap \psi \leadsto \tau$, corresponds to the Agda term `(τ :: ψ :: []) ~> τ`, which is still not ideal. We can, however, define some simple sugar notation:

```
_∩_ : Tₛ -> Tₛ -> List Tₛ
A ∩ B = A :: B :: []
```

so that we finally get the Agda term `τ ∩ ψ ~> τ` which now clearly corresponds to $\tau \cap \psi \leadsto \tau$.

As the above example clearly shows, the first/simplest measure of the amount of implementation overheads, is simply the length of the code/proof scrips, defining the terms and lemmas of a theory. Whilst the length of code might provide an indication of the possible level of implementation overheads, it is important to keep in mind, that brevity of code can often also depend on the level of transparency, as evidenced by both Example 3.1 and the one above, where the shorter code turned also the less transparent one. Depending on the priorities, we therefore often sacrifice either transparency for brevity or vice versa (which can greatly impact his simple metric for overheads).

Therefore, instead of simply looking at the length of the produced document, we also compare the number of lemmas, disregarding the length of each one. Even though this measure also carries disadvantages (one could, for example, in-line the whole Church Rosser proof into one giant lemma) it is less sensitive in regard to transparency.

Another aspect which ties into both transparency and mechanization overheads is the level of automation. As was demonstrated by Example 3.1, wherein the lemma could in fact be proved automatically with almost no user input, having low implementation overheads is often tied to the level of automation the tool provides.

More concretely, a tool with good automation will include a standard library of common definitions and theorems, so that the user does not have to re-prove these basic properties and instead can focus on the specific theory she/he wants to prove.

This is indeed largely the reason why we used Isabelle along with the nominal sets library[2], maintained by Christian Urban, where the theory was conveniently hidden away and managed for us by Isabelle's automatic provers, so that our mechanization overheads were minimal. However, there were several caveats to this, which we discuss in the next chapter.

On the other hand, the choice of locally nameless encoding, as opposed to using pure de Bruijn indices, was motivated by the claim that locally nameless encoding largely mitigates the disadvantages of de Bruijn indices especially when it comes to technology transparency. The LN encoding is also a lot more bare-bones and carries relatively small but manageable overheads.

In order to keep our comparison balanced, we didn't leverage Isabelle's automation to it's fullest, choosing instead, to keep some lemmas (especially in the nominal implementation) deliberately verbose, so as to keep them both more transparent and easier to compare with the locally nameless versions. Another reason to this was the choice of the second implementation language, Agda, which doesn't include as many automation features as Isabelle.

[2] http://www.inf.kcl.ac.uk/staff/urbanc/Nominal/

# 4. Nominal vs. Locally nameless

This chapter looks at the two different mechanizations of the $\lambda$-$Y$ calculus, introduced in the previous chapter, namely an implementation of the calculus using nominal sets and a locally nameless (LN) mechanization. Having presented the two approaches to formalizing binders in Section 2.1, this chapter explores the consequences of choosing either mechanization, especially in terms of technology transparency and overheads introduced as a result of the chosen mechanization.

Whilst we found that the nominal version of the definitions and proofs turned out to be more transparent than the locally nameless mechanization, there were some large overheads associated with the implementation of certain features in the $\lambda$-$Y$ calculus. The LN mechanization, on the other hand, carried a small but consistent level of overhead throughout the formalization, proving that it was indeed a good compromise between implementation overheads and transparency.

## 4.1 Overview

We chose the length of the implemented theory files as a simple measure of implementation overheads. As expected, the Locally nameless version of the calculus (1143 lines) was about 50% longer than the Nominal encoding (723 lines). However, this measure is not always ideal (due to the reasons outlined in Section 3.1.2), and we therefore also present the comparison between the two versions in terms of the individual definitions and lemmas that correspond to each other in the two mechanizations and the informal definitions/lemmas:

| Informal | Nominal | Locally nameless |
|---|---|---|
| Definition of terms | **nominal_datatype** *trm* | **datatype** *ptrm*<br>**inductive** *trm* |
| Definition of substitution | **nominal_function** *subst* | **fun** *opn*<br>**fun** *cls*<br>**fun** *subst* |
| Lemma 2.1 ($\forall M.\ \exists M'.\ M \ggg M'$) | **lemma** *pbeta_max_ex* | **lemma** *pbeta_max_ex*<br>**lemma** *fv_opn_cls_id2*<br>**lemma** *pbeta_max_cls* |
| Lemma 2.2 ($\forall M, M', M''.\ M \ggg M'' \wedge M \gg M' \implies M' \gg M''$) | **lemma** *pbeta_max_closes_pbeta*<br>**lemma** *pbeta_cases_2*<br>**lemma** *Lem2_5_1*<br>**lemma** *pbeta_lam_case_ex* | **lemma** *pbeta_max_closes_pbeta*<br>**lemma** *Lem2_5_1opn* |

| Informal | Nominal | Locally nameless |
|---|---|---|
| Theorem 2.2 (Subject reduction for $\Rightarrow_Y^*$) | **lemma** *beta_Y_typ*<br>**lemma** *subst_typ*<br>**lemma** *wt_terms_impl_wf_ctxt*<br>**lemma** *wt_terms_cases_2* | **lemma** *beta_Y_typ*<br>**lemma** *opn_typ*<br>**lemma** *wt_terms_impl_wf_ctxt* |

The table above lists the major lemmas discussed throughout this thesis, along with the names of these lemmas in the concrete implementations (these can be found in the Appendix), as well as the additional lemmas the proofs of these depend on. For example, the lemma *pbeta_max_ex* depends on *fv_opn_cls_id2* and *pbeta_max_cls* (which may themselves depend on other smaller lemmas). Overall, the mechanization using nominal sets includes 33 lemmas, whereas the locally nameless has 71 individual lemmas. The fact that whilst the LN mechanization includes more than twice as many lemmas as the nominal formalization, its only roughly 50% longer, meaning that many of these lemmas are short simple proofs, which supports our assertion that using the locally nameless representation of binders carries larger overhead, but keeps the difficulty of proving these additional lemmas low.

The rest of this chapter provides an overview of some of the technical points of the $\lambda$-$Y$ calculus mechanization which highlight the differences between the two mechanizations. However, we conclude that on the whole, neither mechanization proved to be significantly better than the other. This is especially true when it comes to proofs in both mechanizations. As the code printout in the Appendix clearly shows, both mechanizations have the same structure and largely the same syntax and formulation of lemmas.
Additionally, when taking a finer grained look at the length of code by section, rather than as a whole, the lengths of the main lemmas in both mechanizations are much closer, as the overheads of the locally nameless encoding occur mainly in the definitions of terms and substitution/open/close operations:

| | Nominal | Locally nameless |
|---|---|---|
| Definition of $\lambda$-$Y$ terms | 15 | 11 |
| Definition of well formed terms | - | 15 |
| Definition of the open operation | - | 18 |
| Definition of substitution | 56 | 124 |
| Definition of the close operation | - | 86 |
| $\beta Y$-reduction | 17 | 25 |
| Parallel $\beta Y$-reduction | 17 | 27 |
| Maximal parallel $\beta Y$-reduction | 49 | 60 |
| Lemma 2.1 | 24 | 107 |
| Lemma 2.2 | 156 | 145 |
| Proof of **dp**$(\gg)$ | 18 | 18 |
| Reflexive-transitive closure of $\beta Y$ | 116 | 231 |
| Simple-typing relation $\vdash$ | 238 | 258 |
| Church Rosser Theorem | 12 | 12 |

Whilst the LN mechanization proved to have significantly higher "obvious" mechanization overheads in terms of code length, the implementation using the nominal library proved to be more difficult to use at certain points, due to the more complex nominal sets theory that implicitly underpinned the mechanization. The LN mechanization proved to be much more simple in practice, even without any library support and the automation, which comes with using Nominal Isabelle.

## 4.2 Definitions

We give a brief overview of the basic definitions of well-typed terms and $\beta$-reduction, specific to both mechanizations. Unsurprisingly, the main differences in these definitions involve $\lambda$-binders.

### 4.2.1 Nominal sets representation

As was shown already in Section 2.1, nominal set representation of terms is largely identical with the informal definition, which is the main reason why this representation was chosen. This section will examine the implementation of $\lambda$-$Y$ calculus in Isabelle, using the Nominal package.

The declaration of the terms and types in Nominal Isabelle is handled using the reserved keywords **atom_decl** and **nominal_datatype**, which are special versions of the **typedecl** and **datatype** primitives, used in the usual Isabelle/HOL session:

```
atom_decl name

nominal_datatype type = O | Arr type type ("_ → _")

nominal_datatype trm =
  Var name
| App trm trm
| Lam x::name t::trm  binds x in t ("Lam [_]. _" [100, 100] 100)
| Y type
```

The special **binds _ in _** syntax in the `Lam` constructor declares `x` to be bound in the body `t`, telling Nominal Isabelle that `Lam` terms should be identified up to $\alpha$-equivalence, where a term $\lambda x.x$ and $\lambda y.y$ are considered identical/equal, because both $x$ and $y$ are bound in the two respective terms, and can both be $\alpha$-converted to the same term, for example $\lambda z.z$. In fact, proving such a lemma in Nominal Isabelle is trivial:

```
lemma "Lam [x]. Var x = Lam [y]. Var y" by simp
```

The special **nominal_datatype** declaration also generates definitions of free variables/freshness and other simplification rules. (Note: These can be inspected in Isabelle, using the **print_theorems** command.)

Other definitions, such as $\beta$-reduction and the notion of substitution are also unchanged with regards to the usual definition (except for the addition of the $Y$ case, which is trivial):

**Definition 4.1.** [Capture-avoiding substitution]

$$\begin{aligned}
x[S/y] &= \begin{cases} S & \text{if } x \equiv y \\ x & \textit{otherwise} \end{cases} \\
(MN)[S/y] &= (M[S/y])(N[S/y]) \\
x \sharp y, S \implies (\lambda x.M)[S/y] &= \lambda x.(M[S/y]) \\
(Y_\sigma)[S/y] &= Y_\sigma
\end{aligned}$$

The side-condition $x \sharp y, S$ in the definition above can be read as "$x$ is fresh in $N$", namely, the atom $x$ is not the same as $y$ and does not appear in $S$, i.e. for a $\lambda$-term $M$, we have $x \sharp M$ iff $x \notin \mathsf{FV}(M)$.

Whilst on paper, all this definition is unchanged from the informal presentation, there are a few caveats when it comes to actually implementing these definitions in Isabelle, using the Nominal package. Since this definition of substitution includes the freshness condition, it cannot be defined using the usual structural recursion via the **primrec** or **fun** keywords, generally used for this purpose. Instead we have to define capture avoiding substitution using a **nominal_function** declaration:

```
nominal_function
  subst :: "trm ⇒ name ⇒ trm ⇒ trm"  ("_ [_ ::= _]" [90, 90, 90] 90)
where
  "(Var x)[y ::= s] = (if x = y then s else (Var x))"
| "(App t1 t2)[y ::= s] = App (t1[y ::= s]) (t2[y ::= s])"
| "atom x ♯ (y, s) ⇒ (Lam [x]. t)[y ::= s] = Lam [x].(t[y ::= s])"
| "(Y t)[y ::= s] = Y t"
```

Unlike using the usual **fun** declaration of a recursive function in Isabelle, where Isabelle automatically checks the definition for pattern completeness (for the term being pattern matched on) and overlap. The **fun** definition also automatically checks/proves termination of such recursive functions and generates simplification rules, which can be used for equational reasoning involving the function.

Unfortunately, this isn't the case for the **nominal_function** declaration, where there are several goals (13 in the case of the `subst` definition) which the user has to manually prove about the function definition, including proving termination, and pattern disjointness and completeness. This turned out to be a bit problematic, as the goals involved proving properties like:

```
⋀x t xa ya sa ta.
  eqvt_at subst_sumC (t, ya, sa) ⇒
  eqvt_at subst_sumC (ta, ya, sa) ⇒
  atom x ♯ (ya, sa) ⇒ atom xa ♯ (ya, sa) ⇒
  [[atom x]]lst. t = [[atom xa]]lst. ta ⇒
  [[atom x]]lst. subst_sumC (t, ya, sa) =
    [[atom xa]]lst. subst_sumC (ta, ya, sa)
```

Whilst most of the goals were trivial, proving cases involving $\lambda$-terms involved a substantial un-

derstanding of the internal workings of Isabelle and the Nominal package early on into the mechanization and as a novice to using Nominal Isabelle, understanding and proving these properties proved challenging.
Whilst our formalization required only a handful of other recursive function definitions, in a different theory with significantly more function definitions, proving such goals from scratch would prove a challenge to a Nominal Isabelle newcomer as well as a tedious implementation overhead.

### 4.2.2 Locally nameless representation

As we have seen, on paper at least, the definitions of terms and capture-avoiding substitution, using nominal sets, are unchanged from the usual informal definitions. The situation is somewhat different for the locally nameless mechanization. Since the LN approach combines the named and de Bruijn representations, there are two different constructors for free and bound variables:

#### 4.2.2.1 Pre-terms

**Definition 4.2.** [LN pre-terms]

$$M ::= x \mid n \mid MM \mid \lambda M \mid Y_\sigma \text{ where } x \in Var \text{ and } n \in \mathbb{N}$$

Similarly to the de Bruijn presentation of binders, the $\lambda$-term no longer includes a bound variable, so a named representation term $\lambda x.x$ becomes $\lambda 0$ in LN. As was mentioned in Section 2.1, the set of pre-terms, defined in Definition 4.2, is a superset of $\lambda$-$Y$ terms and includes terms which are not well formed $\lambda$-$Y$ terms.

**Example 4.1.** The pre-term $\lambda 3$ is not a well-formed $\lambda$-$Y$ term, since the bound variable index is out of scope. In other words, there is no corresponding (named) $\lambda$-$Y$ term to $\lambda 3$.

Since we don't want to work with terms that do not correspond to $\lambda$-$Y$ terms, we have to introduce the notion of a *well-formed term*, which restricts the set of pre-terms to only those that correspond to $\lambda$-$Y$ terms (i.e. this inductive definition ensures that there are no "out of bounds" indices in a given pre-term):

**Definition 4.3.** [Well-formed terms]

$$(fvar)\ \frac{}{\textbf{term}(x)} \qquad (Y)\ \frac{}{\textbf{term}(Y_\sigma)}$$

$$(lam)\ \frac{x \notin FV(M) \qquad \textbf{term}(M^x)}{\textbf{term}(\lambda M)} \qquad (app)\ \frac{\textbf{term}(M) \qquad \textbf{term}(M)}{\textbf{term}(MN)}$$

Already, we see that this formalization introduces some overheads with respect to the informal/nominal encoding of the $\lambda$-$Y$ calculus.
The upside of this definition of $\lambda$-$Y$ terms becomes apparent when we start thinking about $\alpha$-equivalence and capture-avoiding substitution. Since the LN terms use de Bruijn levels for bound variables, there is only one way to write the term $\lambda x.x$ or $\lambda y.y$ as a LN term, namely $\lambda 0$. As the $\alpha$-equivalence classes of named $\lambda$-$Y$ terms collapse into a singleton $\alpha$-equivalence class in a LN representation, the notion of $\alpha$-equivalence becomes trivial.

As a result of using LN representation of binders, the notion of substitution is split into two distinct operations. One operation is the substitution of bound variables, called *opening*. The other is substitution, defined only for free variables.

**Definition 4.4.** [Opening and substitution]

We will usually assume that $S$ is a well-formed LN term when proving properties about substitution and opening. The abbreviation $M^N \equiv \{0 \to N\}M$ is used throughout this chapter.

i) Opening:

$$
\begin{aligned}
\{k \to S\}x &= x \\
\{k \to S\}n &= \begin{cases} S & \text{if } k \equiv n \\ n & \textit{otherwise} \end{cases} \\
\{k \to S\}(MN) &= (\{k \to S\}M)(\{k \to S\}N) \\
\{k \to S\}(\lambda M) &= \lambda(\{k+1 \to S\}M) \\
\{k \to S\}Y_\sigma &= Y_\sigma
\end{aligned}
$$

ii) Substitution:

$$
\begin{aligned}
x[S/y] &= \begin{cases} S & \text{if } x \equiv y \\ x & \textit{otherwise} \end{cases} \\
n[S/y] &= n \\
(MN)[S/y] &= (M[S/y])(N[S/y]) \\
(\lambda M)[S/y] &= \lambda.(M[S/y]) \\
Y_\sigma[S/y] &= Y_\sigma
\end{aligned}
$$

Having defined the *open* operation, we turn back to the definition of well formed terms, specifically to the $(lam)$ rule, which has the precondition $\textbf{term}(M^x)$. Intuitively, for the given term $\lambda M$, the term $M^x$ is obtained by replacing all indices bound to the outermost $\lambda$ by $x$. Then, if $M^x$ is well formed, so is $\lambda M$.

**Example 4.2.** For example, taking the term $\lambda\lambda 0(z\,1)$, we can construct the following proof-tree, showing that the term is well formed:

$$
\begin{array}{c}
(lam)\,\cfrac{(lam)\,\cfrac{(app)\,\cfrac{(fvar)\,\cfrac{}{\textbf{term}(y)} \qquad (app)\,\cfrac{(fvar)\,\cfrac{}{\textbf{term}(z)} \qquad (fvar)\,\cfrac{}{\textbf{term}(x)}}{\textbf{term}(z\,x)}}{\textbf{term}((0(z\,x))^y)}}{\textbf{term}((\lambda 0(z\,1))^x)}}{\textbf{term}(\lambda\lambda 0(z\,1))}
\end{array}
$$

We assumed that $x \not\equiv y \not\equiv z$ in the proof tree above and thus omitted the $x \notin \textbf{FV}\ldots$ branches, as they are not important for this example.
If on the other hand, we try construct a similar tree for a term which is obviously not well formed, such as $\lambda\lambda 2(z\,1)$, we get a proof tree with a branch which cannot be closed ($\textbf{term}(2)$):

$$
\dfrac{\text{term}(2) \qquad \dfrac{\dfrac{}{\text{term}(z)}\,(fvar) \qquad \dfrac{}{\text{term}(x)}\,(fvar)}{\text{term}(z\,x)}\,(app)}{\dfrac{\dfrac{\text{term}((2(z\,x))^{y})}{\text{term}((\lambda 2(z\,1))^{x})}\,(lam)}{\text{term}(\lambda\lambda 2(z\,1))}\,(lam)}\,(app)
$$

#### 4.2.2.2 $\beta$-reduction for LN terms

Finally, we examine the formulation of $\beta$-reduction in the LN presentation of the $\lambda$-$Y$ calculus. Since we only want to perform $\beta$-reduction on valid $\lambda$-$Y$ terms, the inductive definition of $\beta$-reduction in the LN mechanization now includes the precondition that the terms appearing in the reduction are well formed:

**Definition 4.5.** [$\beta$-reduction (LN)]

$$
(red_L)\ \dfrac{M \Rightarrow_Y M' \qquad \text{term}(N)}{MN \Rightarrow_Y M'N} \qquad (red_R)\ \dfrac{\text{term}(M) \qquad N \Rightarrow_Y N'}{MN \Rightarrow_Y M'N}
$$

$$
(abs)\ \dfrac{x \notin \text{FV}(M) \cup \text{FV}(M') \qquad M^x \Rightarrow_Y (M')^x}{\lambda M \Rightarrow_Y \lambda M'} \qquad (\beta)\ \dfrac{\text{term}(\lambda M) \qquad \text{term}(N)}{(\lambda M)N \Rightarrow_Y M^N}
$$

$$
(Y)\ \dfrac{\text{term}(M)}{Y_\sigma M \Rightarrow_Y M(Y_\sigma M)}
$$

As expected, the *open* operation is now used instead of substitution in the $(\beta)$ rule.
The $(abs)$ rule is also slightly different, also using the *open* in its precondition. Intuitively, the usual formulation of the $(abs)$ rule states that in order to prove that $\lambda x.M$ reduces to $\lambda x.M'$, we can simply "un-bind" $x$ in both $M$ and $M'$ and show that $M$ reduces to $M'$ (reasoning bottom-up from the conclusion to the premises). Since in the usual formulation of the $\lambda$-calculus, there is no distinction between free and bound variables, this change (where $x$ becomes free) is implicit. In the LN presentation, however, this operation is made explicit by opening both $M$ and $M'$ with some free variable $x$ (not appearing in either $M$ nor $M'$), which replaces the bound variables/indices (bound to the outermost $\lambda$) with $x$.
While this definition is equivalent to the usual/informal definition, the induction principle this definition yields may not always be sufficient, especially in situations where we want to open up a term with a free variable which is not only fresh in $M$ and $M'$, but possibly in a wider context. We therefore followed the approach of B. Aydemir et al. (2008) and re-defined the $(abs)$ rule (and other definitions involving the choice of fresh/free variables) using *cofinite quantification*:

$$
(abs)\ \dfrac{\forall x \notin L.\ M^x \Rightarrow_Y M'^x}{\lambda M \Rightarrow_Y \lambda M'}
$$

For an example, where this formulation using *cofinite quantification* was necessary, see Lemma 4.2).

## 4.3 Proofs

Having described the implementations of the two binder representations along with the definitions of capture-avoiding substitution using nominal sets and the corresponding *subsitution* and *open* operations in the LN mechanization, we come the the main part of the comparison, namely the proof of the Church Rosser theorem. This section examines specific instances of some of the major lemmas which form parts of this bigger result. The general outline of the proof has been described in Section 2.3.2.

### 4.3.1 Lemma 2.1

The first major result in both implementations is Lemma 2.1, which states that for every $\lambda$-$Y$ term $M$, there is a term $M'$, s.t. $M \ggg M'$.

> *Remark.* This result is trivial for $\gg$, as we can easily prove the derived rule $(refl^*)$, but not for $\ggg$:
>
> > **Lemma 4.1.** [$\gg$ admits $(refl^*)$]
> >
> > The following rule is admissible in the deduction system $\gg$:
> >
> > $$(refl^*)\ \frac{}{M \gg M}$$
> >
> > *Proof.* By induction on $M$. □

Since $\ggg$ restricts the use of the $(app)$ rule to terms which do not contain a $\lambda$ or $Y$ as its left-most sub term, Lemma 4.1 does not hold in $\ggg$ for terms like $(\lambda x.x)y$, namely, $(\lambda x.x)y \ggg (\lambda x.x)y$ is not a valid reduction (see Example 2.4). It is, however, not difficult to see that such terms can simply be $\beta$-reduced until all the redexes have been contracted, so that we have $(\lambda x.x)y \ggg y$ for the term above.
Seen as a weaker version of Lemma 4.1, the proof of Lemma 2.1, at least in theory, should then only differ in the case of an application, where we have do a case analysis on the left sub-term of any given $M$.

This is indeed the case when using the nominal mechanization, where the proof looks like this:

```
lemma pbeta_max_ex:
  fixes M
  shows "∃M'. M >>> M'"
apply (induct M rule:trm.induct)
apply auto
apply (case_tac "not_abst S")
apply (case_tac "not_Y S")
apply auto[1]
proof goal_cases
case (1 P Q P' Q')
```

```
11    then obtain σ where 2: "P = Y σ" using not_Y_ex by auto
12    have "App (Y σ) Q >>> App Q' (App (Y σ) Q')"
13    apply (rule_tac pbeta_max.Y)
14    by (rule 1(2))
15    thus ?case unfolding 2 by auto
16  next
17  case (2 P Q P' Q')
18    thus ?case
19    apply (nominal_induct P P' avoiding: Q Q' rule:pbeta_max.strong_induct)
20    by auto
21  qed
```

After applying induction and calling `auto`, which is Isabelle's automatic prover that does simple term rewriting and basic proof search, we can inspect the remaining goals at line 5, to see that the only goal that remains is the case of $M$ being an application, naley we have to prove the following:

$$\forall S\ T\ U\ V.\ S \ggg U \implies T \ggg V \implies \exists M'.\ ST \ggg M'$$

Lines 6 and 7 in the proof script then correspond to doing a case analysis on $S$ (where $M = ST$). We end up with 3 goals, corresponding to $S$ either being a $\lambda$-term, $Y$-term or neither (shown below in reverse order):

```
1. ... not_abst S ⟹ not_Y S ⟹ ∃M'. App S T >>> M'
2. ... not_abst S ⟹ ¬ not_Y S ⟹ ∃M'. App S T >>> M'
3. ... ¬ not_abst S ⟹ ∃M'. App S T >>> M'
```

The first goal is discharged by calling `auto` again (line 8), since we can simply apply the $(app)$ rule in this instance. The two remaining cases are discharged with the additional information that $S$ is either a $\lambda$-term or a $Y$-term.

So far, we have looked at the version of the proof using nominal Isabelle and this is especially apparent in line 19, where we use the stronger `nominal_induct` rule, with the extra parameter `avoiding: Q Q'`, which ensures that any new bound variables will be sufficiently fresh with regards to `Q` and `Q'`, in that the fresh variables won't appear in either of the terms.
Since bound variables are distinct in the LN representation, the equivalent proof simply uses the usual induction rule (line 19):

```
1  lemma pbeta_max_ex:
2    fixes M assumes "trm M"
3    shows "∃M'. M >>> M'"
4  using assms apply (induct M rule:trm.induct)
5  apply auto
6  apply (case_tac "not_abst t1")
7  apply (case_tac "not_Y t1")
8  apply auto[1]
9  proof goal_cases
10 case (1 P Q P' Q')
11   then obtain σ where 2: "P = Y σ" using not_Y_ex by auto
12   have "App (Y σ) Q >>> App Q' (App (Y σ) Q')"
```

```
13    apply (rule_tac pbeta_max.Y)
14    by (rule 1(4))
15    thus ?case unfolding 2 by auto
16  next
17  case (2 P Q P' Q')
18    from 2(3,4,5,1,2) show ?case
19    apply (induct P P' rule:pbeta_max.induct)
20    by auto
21  next
22  case (3 L M)
23    then obtain x where 4:"x ∉ L ∪ FV M" by (meson FV_finite finite_UnI x_Ex)
24    with 3 obtain M' where 5: "M^FVar x >>> M'" by auto
25
26    have 6: "⋀y. y ∉ FV M' ∪ FV M ∪ {x} ⟹ M^FVar y >>> (\\x^M')^FVar y"
27    unfolding opn'_def cls'_def
28    apply (subst(3) fv_opn_cls_id2[where x=x])
29    using 4 apply simp
30    apply (rule_tac pbeta_max_cls)
31    using 5 opn'_def by (auto simp add: FV_simp)
32
33    show ?case
34    apply rule
35    apply (rule_tac L="FV M' ∪ FV M ∪ {x}" in pbeta_max.abs)
36    using 6 by (auto simp add: FV_finite)
37  qed
```

As one can immediately see, this proof proceeds exactly in the same fashion, as the nominal one, up to line 20. However, unlike in the nominal version of the proof, in the LN proof, the `auto` call at line 8 could not automatically prove the case where $M$ is a $\lambda$-term.

This is perhaps not too surprising, since the LN encoding is a lot more "bare bones", and thus there is little that would aid Isabelle's automation. The nominal package, on the other hand, was designed to make reasoning with binders as painless as possible, which definitely shows in this example.

When we compare the two goals for the $\lambda$ case in both versions of the proof, we clearly see the differences in the treatment of binders:

$$\begin{aligned}\text{Nominal: } & \forall x\, M.\ \exists M'.\ M \ggg M' \implies \exists M'.\lambda x.M \ggg M' \\ \text{Locally nameless: } & \forall L\, M.\ \text{fin } L \implies \text{term}(\lambda.M) \implies (\forall x \notin L.\ \exists M''.\ M^x \ggg M'') \\ & \implies \exists M'.\ \lambda.M \ggg M'\end{aligned}$$

Unlike in the nominal proof, where from $M \ggg M'$ we get $\lambda x.M \ggg \lambda x.M'$ by $(abs)$ immediately, the proof of $\exists M'.\ \lambda.M \ggg M'$ in the LN mechanization is not as trivial.

The difficulty arises with the precondition $\forall x \notin L.\ M^x \ggg (M')^x$ in the LN version of the $(abs)$ rule:

$$(abs)\ \frac{\exists M'.\ \forall x \notin L.\ M^x \ggg (M')^x}{\exists M'.\ \lambda M \ggg \lambda M'}$$ [1]

This version of the rule with the existential quantification shows the subtle difference between the inductive hypothesis $\forall x \notin L.\ \exists M'.\ M^x \ggg (M')^x$[2] we have, and the premise $\exists M'.\ \forall x \notin L.\ M^x \ggg (M')^x$ that we want to show. In order to prove the latter, we assume that there is some $M'$ for a specific $x \notin L$ s.t. $M^x \ggg (M')^x$.

At this point, we cannot proceed without re-examining the definition of *opening*, especially in that this operation lacks an inverse. Whereas in a named representation, where bound variables are bound via context only, LN terms have specific constructors for free and bound variables together with an operation for turning bound variables into free variables, namely the *open* function. In this proof, however, we need the inverse operation, wherein we turn a free variable into a bound one. We call this the *close* operation:

**Definition 4.6.** [Close operation]

This definition was adapted from the B. Aydemir et al. (2008) paper. We adopt the following convention, writing ${}^{\backslash x}M \equiv \{0 \leftarrow x\}M$.

$$\begin{aligned}
\{k \leftarrow x\}y &= \begin{cases} k & \text{if } x \equiv y \\ y & \textit{otherwise} \end{cases} \\
\{k \leftarrow s\}n &= n \\
\{k \leftarrow s\}(MN) &= (\{k \leftarrow s\}M)(\{k \leftarrow s\}N) \\
\{k \leftarrow s\}(\lambda M) &= \lambda(\{k+1 \leftarrow s\}M) \\
\{k \leftarrow s\}Y_\sigma &= Y_\sigma
\end{aligned}$$

**Example 4.3.** To demonstrate the close operation, take the term $\lambda xy$. Applying the close operation with the free variable $x$, we get ${}^{\backslash x}(\lambda xy) = \lambda 1y$. Whilst the original term might have been well formed, the closed term, as is the case here, may not be.

Intuitively, it is easy to see that closing a well formed term and then opening it with the same free variable produces the original term, namely $({}^{\backslash x}M)^x \equiv M$. This can be made even more general with the following lemma about the relationship between the open, close and substitution operations:

**Lemma 4.2.** $\textbf{term}(M) \implies \{k \rightarrow y\}\{k \leftarrow x\}M = M[y/x]$

*Proof.* By induction on the relation $\textbf{term}(M)$. The rough outline of the $(lam)$ case, which is the only non-trivial case, is shown below:

By *IH*, we have $\forall z \notin L.\ \{k+1 \rightarrow y\}\{k+1 \leftarrow x\}M^z = (M^z)[y/x]$. Then:

[1] While the original goal is $\exists M'.\ \lambda.M \ggg M'$, since there is only one possible "shape" for the right-hands side term, namely $M'$ must be a $\lambda$-term, we can easily rewrite this goal as $\exists M'.\ \lambda.M \ggg \lambda.M'$.

[2] It can easily be shown that any pre-term $M$ can be written using another pre-term $N$ s.t. $M \equiv N^x$ for some $x$.

$$\{k \to y\}\{k \leftarrow x\}(\lambda M) = (\lambda M)[y/x] \iff \tag{4.1}$$
$$\lambda(\{k+1 \to y\}\{k+1 \leftarrow x\}M) = \lambda(M[y/x]) \iff \tag{4.2}$$
$$\{k+1 \to y\}\{k+1 \leftarrow x\}M = M[y/x] \iff \tag{4.3}$$
$$\{0 \to z\}\{k+1 \to y\}\{k+1 \leftarrow x\}M = \{0 \to z\}(M[y/x]) \iff \tag{4.4}$$
$$\{k+1 \to y\}\{k+1 \leftarrow x\}\{0 \to z\}M = \{0 \to z\}(M[y/x]) \iff \tag{4.5}$$
$$\{k+1 \to y\}\{k+1 \leftarrow x\}\{0 \to z\}M = (\{0 \to z\}M)[y/x] \tag{4.6}$$

Starting from the goal (4.1), we expand the definitions of *open*, *close* and substitution for the $\lambda$ case in (4.2). (4.3) holds by injectivity of $\lambda$. Then, by choosing a sufficiently fresh $z$ that does not appear in the given context $L$ as well as in neither $\mathbf{FV}(M)$ nor $\{x, y\}$, we have (4.4). We can reorder the open and close operations in (4.5) because it can never be the case that $k+1 = 0$ and $z$ is different from both $x$ and $y$. Finally, (4.6) follows from the fact that we have chosen a $z$ that does not appear in $M$ and is different from $y$.
We can now see that $\{k+1 \to y\}\{k+1 \leftarrow x\}\{0 \to z\}M = (\{0 \to z\}M)[y/x]$ is in fact the *IH* $\{k+1 \to y\}\{k+1 \leftarrow x\}M^z = (M^z)[y/x]$.

$\square$

Having defined the *close* operation and shown that it satisfies certain properties with respect to the *open* operation and substitution, we can now "close" the term $M'$, with respect to the $x$ we fixed earlier and thus show that $\forall y \notin L.\ M^y \ggg (^{\backslash x}M')^y$.

### 4.3.2 Lemma 2.2

While it may seem that the nominal mechanization was universally more concise and easier to work in than the locally nameless implementation, there were a few instances where using the nominal library turned out to be more difficult to understand and use. One such instance, namely defining a **`nominal_function`**, was already discussed. Another example can be found in the implmentation of Lemma 2.2, which is stated as:

$$\forall M, M', M_{max}.\ M \ggg M_{max} \wedge M \gg M' \implies M' \gg M_{max}$$

The proof of this lemma proceeds by induction on the relation $\ggg$. Here we will focus on the $(\beta)$ case, i.e. when we have $M \ggg M_{max}$ by the application of $(\beta)$, first giving an informal proof and then focusing on the implementation specifics in both mechanizations:

#### 4.3.2.1 $(\beta)$ case

We have $M \equiv (\lambda x.P)Q$ and $M_{max} \equiv P_{max}[Q_{max}/x]$, and therefore $(\lambda x.P)Q \ggg P_{max}[Q_{max}/x]$ and $(\lambda x.P)Q \gg M'$.
By performing case analysis on the reduction $(\lambda x.P)Q \gg M'$, we know that $M' \equiv (\lambda x.P')Q'$ or $M' \equiv P'[Q'/x]$ for some $P', Q'$, since only these two trees are valid:

$$(abs)\,(app)\ \dfrac{\dfrac{\dfrac{\vdots}{P \gg P'}}{\lambda x.P \gg \lambda x.P'} \qquad \dfrac{\vdots}{Q \gg Q'}}{(\lambda x.P)Q \gg (\lambda x.P')Q'} \qquad \text{or} \qquad (\beta)\ \dfrac{\dfrac{\vdots}{P \gg P'} \qquad \dfrac{\vdots}{Q \gg Q'}}{(\lambda x.P)Q \gg P'[Q'/x]}$$

For the first case, where $M' \equiv (\lambda x.P')Q'$, by $IH$, we have $P' \gg P_{max}$ and $Q' \gg Q_{max}$. Thus, we can prove that $M' \gg P_{max}[Q_{max}/x]$:

$$(IH)\,(\beta)\ \dfrac{\dfrac{}{P' \gg P_{max}} \qquad (IH)\ \dfrac{}{Q' \gg Q_{max}}}{(\lambda x.P')Q' \gg P_{max}[Q_{max}/x]}$$

In the case where $M' \equiv P'[Q'/x]$, we also have $P' \gg P_{max}$ and $Q' \gg Q_{max}$ by $IH$. The result $M' \gg P_{max}[Q_{max}/x]$ follows from the following auxiliary lemma:

> **Lemma 4.3.** [Parallel substitution]
>
> The following rule is admissible in $\gg$:
>
> $$(||_{subst})\ \dfrac{M \gg M' \qquad N \gg N'}{M[N/x] \gg M'[N'/x]}$$

#### 4.3.2.2 Nominal implementation

The code below shows the proof of the $(\beta)$ case, described above:

```
1  case (beta x Q Qmax P Pmax)
2    from beta(1,7) show ?case
3    apply (rule_tac pbeta_cases_2)
4    apply (simp, simp)
5    proof -
6    case (goal2 Q' P')
7      with beta have "P' ≫ Pmax" "Q' ≫ Qmax" by simp+
8      thus ?case unfolding goal2 apply (rule_tac Lem2_5_1) by simp+
9    next
10   case (goal1 P' Q')
11     with beta have ih: "P' ≫ Pmax" "Q' ≫ Qmax" by simp+
12     show ?case unfolding goal1
13     apply (rule_tac pbeta.beta) using goal1 beta ih
14     by simp_all
15   qed
```

There were a few quirks when implementing this proof in the nominal mechanization, specifically in line 3, where the case analysis on the shape of $M'$ needed to be performed. Applying the automatically generated `pbeta.cases` rule yielded the following goal for the case where $M' \equiv P'[Q'/x]$:

```
2. ⋀xa Q' R P'.
       [[atom x]]lst. P = [[atom xa]]lst. R ⟹
       M' = P' [xa ::= Q'] ⟹
       atom xa ♯ Q ⟹ atom xa ♯ Q' ⟹ R ≫ P' ⟹ Q ≫ Q' ⟹
       M' ≫ Pmax [x ::= Qmax]
```

Obviously, this is not the desired shape of the goal, because we obtained a weaker premise, where we have some $R$, such that $\lambda x.P \equiv_\alpha \lambda xa.R$ (this is essentially what `[[atom x]]lst. P = [[atom xa]]lst. R` states) and therefore we get a $P'$ where $M' \equiv P'[Q'/xa]$. What we actually want is a term $P'$ s.t. $M' \equiv P'[Q'/x]$, i.e. $x = xa$. In order to "force" $x$ and $xa$ to actually be the same atom, we had to prove the following "cases" lemma:

```
lemma pbeta_cases_2:
  shows "atom x ♯ t ⟹ App (Lam [x]. s) t ≫ a2 ⟹
    (⋀s' t'. a2 = App (Lam [x]. s') t' ⟹ atom x ♯ t' ⟹
        s ≫ s' ⟹ t ≫ t' ⟹ P) ⟹
    (⋀t' s'. a2 = s' [x ::= t'] ⟹ atom x ♯ t ⟹ atom x ♯ t' ⟹
        s ≫ s' ⟹ t ≫ t' ⟹ P) ⟹ P"
⋮
```

In the lemma above, `(⋀t' s'. a2 = s' [x ::= t'] ⟹ atom x ♯ t ⟹ atom x ♯ t' ⟹ s ≫ s' ⟹ t ≫ t' ⟹ P) ⟹ P` corresponds to the case with the premises we want to have, instead of the ones we get from the "cases" lemma generated as part of the definition of $\gg$.

The proof of this lemma required proving another lemma shown below, which required descending into nominal set theory that was previously mostly hidden away from the mechanization (the proofs of the `have` lemmas were omitted for brevity):

```
lemma "(Lam [x]. s) ≫ s' ⟹ ∃t. s' = Lam [x]. t ∧ s ≫ t"
proof (cases "(Lam [x]. s)" s' rule:pbeta.cases, simp)
  case (goal1 _ _ x')
    then have 1: "s ≫ ((x' ↔ x) • M')" ...
    from goal1 have 2: "(x' ↔ x) • s' = Lam [x]. ((x' ↔ x) • M')" ...
    from goal1 have "atom x ♯ (Lam [x']. M')" using fresh_in_pbeta ...
    with 2 have "s' = Lam [x]. ((x' ↔ x) • M')" ...
    with 1 show ?case by auto
qed
```

Clearly, the custom "cases" lemma was necessary from a purely technical view, as it would be deemed too trivial to bother proving in an informal setting. The need for such a lemma also demonstrates that whilst the nominal package package tries to hide away the details of the theory, every once in a while, the user has to descent into nominal set theory, to prove certain properties about binders, not handled by the automation.
For us, the nominal package thus proved to be a double edged sword, as it initially provided a fairly low cost of entry (there was practically no need to understand any nominal set theory to get started), but proved to be much more challenging to understand in certain places, such as when proving `pbeta_cases_2`.
Whilst the finial `pbeta_cases_2` proof turned out to be fairly short thanks to automation of the

nominal set theory, it took some time to work out the proof outline in such a ways as to leverage Isabelle's automation to a high degree.
The LN mechanization, whilst having bigger overheads in terms of extra definitions and lemmas that had to be proven "by hand", was in fact a lot more transparent as a result, as the degree of difficulty after the initial cost of entry did not rise significantly with more complicated lemmas.

#### 4.3.2.3 LN implementation

The troublesome case analysis in the Nominal version of the proof was much more straight forward in the LN proof. In fact, there was no need to prove a separate lemma similar to `pbeta_cases_2`, since the auto-generated `pbeta.cases` was sufficient. The only overhead in this version of the lemma came from the use of Lemma 4.3, in that the lemma was first proved in it's classical formulation using substitution, but due to the way substitution of bound terms is handled in the LN mechanization (using the *open function*), a "helper" lemma was proved to convert this result to one using *open*:

> **Lemma 4.4.** [Parallel open]
>
> The following rule is admissible in the LN version of $\gg$:
>
> $$(||_{open})\ \frac{\forall x \notin L.\ M^x \gg M'^x \qquad N \gg N'}{M^N \gg M'^{N'}}$$

The reason why Lemma 4.4 wasn't proved directly is partially due to the order of implementation of the two mechanizations of the $\lambda$-$Y$ calculus. Since the nominal version, along with all the proofs was carried out first, the LN version of the calculus ended up being more of a port of the nominal theory into a locally nameless setting.
The LN mechanization, being a port of the nominal theory, has both advantages and disadvantages. On the one hand, it ensures a greater consistency between the two theories and easier direct comparison of lemmas, but on the other hand, it meant that certain lemmas could have been made shorter and more "tailored" to the LN mechanization.

# 5. Isabelle vs. Agda

Having previously looked at two approaches to mechanizing binders in Isabelle and having concluded that there were only minor differences between the nominal and LN approaches, we proceeded to the next round of our comparison, by implementing the locally nameless version of the $\lambda$-$Y$ calculus along with proofs of confluence in Agda.
Whilst using Nominal sets in Isabelle turned out to be slightly more transparent and shorter, the big disadvantage, especially for Agda, was the fact that the nominal set theory would have been much more tedious to use, if we had to, say, implement it from scratch. Whilst Nominal Isabelle is a fairly mature library, formalizing the theory of nominal sets and providing ample sugar for the user to hide away the theory, Agda is a lot more bare-bones and the "penalty" incurred by formalizing and then using nominal sets would have been significantly higher.
Instead, we chose to implement the locally nameless version, which proved to have consistent, but fairly minimal overheads, requiring relatively little extra theory. This is supported by the fact that the LN version of the calculus and proofs in Isabelle was roughly 1140 lines of code, whereas the Agda version was only slightly longer at 1350 lines, which, when adjusted to the same spacing/formatting comes down to roughly 1230 lines. This rough metric also demonstrates that whilst Isabelle's automation can be a lot more powerful than Agda's, in our case (partially by design), there were only a few instances where Isabelle clearly had the upper hand. Over all, it turned out, once again, that there were only small, often just cosmetic, differences between the two implementations.

## 5.1 Overview

One of the most apparent differences between Agda and Isabelle is the treatment of functions and proofs in both languages. Whilst in Isabelle, there is always a clear syntactic distinction between programs and proofs, Agda's richer dependent-type system allows constructing proofs as programs. This distinction is especially visible in inductive proofs, which have a completely distinct syntax in Isabelle. As proofs are not objects which can be directly manipulated in Isabelle, to modify the proof goal, user commands such as `apply rule` or `by auto` are used:

```
lemma subst_fresh: "x ∉ FV t ⟹ t[x ::= u] = t"
apply (induct t)
by auto
```

In the proof above, the command `apply (induct t)` takes a proof object with the goal `x ∉ FV t ⟹ t[x ::= u] = t`, and applies the induction principle for `t`, generating 5 new proof obligations:

```
proof (prove)
goal (5 subgoals):
1. ⋀xa. x ∉ FV (FVar xa) ⇒ FVar xa [x ::= u] = FVar xa
2. ⋀xa. x ∉ FV (BVar xa) ⇒ BVar xa [x ::= u] = BVar xa
3. ⋀t1 t2.
     (x ∉ FV t1 ⇒ t1 [x ::= u] = t1) ⇒
     (x ∉ FV t2 ⇒ t2 [x ::= u] = t2) ⇒
     x ∉ FV (App t1 t2) ⇒ App t1 t2 [x ::= u] = App t1 t2
4. ⋀t. (x ∉ FV t ⇒ t [x ::= u] = t) ⇒ x ∉ FV (Lam t) ⇒
     Lam t [x ::= u] = Lam t
5. ⋀xa. x ∉ FV (Y xa) ⇒ Y xa [x ::= u] = Y xa
```

These can then discharged by the call to `auto`, which is a command that invokes the automatic solver, that tries to prove all the goals in the given context.

In contrast to this, in an Agda proof, the proof objects are available to the user directly. Instead of using commands modifying the proof state, one begins with a definition of the lemma:

```
subst-fresh : ∀ x t u -> (x∉FVt : x ∉ (FV t)) -> (t [ x ::= u ]) ≡ t
subst-fresh x t u x∉FVt = ?
```

The `?` acts as a ‘hole’ which the user fills in, by incrementally constructing the proof. Using the Emacs/Atom editor’s “agda-mode”, once can apply a case split to `t` (corresponding to the `apply (induct t)` call in Isabelle), generating the following definition:

```
subst-fresh : ∀ x t u -> (x∉FVt : x ∉ (FV t)) -> (t [ x ::= u ]) ≡ t
subst-fresh x (bv i) u x∉FVt = {!   0!}
subst-fresh x (fv x₁) u x∉FVt = {!   1!}
subst-fresh x (lam t) u x∉FVt = {!   2!}
subst-fresh x (app t t₁) u x∉FVt = {!   3!}
subst-fresh x (Y t₁) u x∉FVt = {!   4!}
```

When the above definition is compiled, Agda generates 5 goals needed to ‘fill’ each hole:

```
?0  :  (bv i [ x ::= u ]) ≡ bv i
?1  :  (fv x₁ [ x ::= u ]) ≡ fv x₁
?2  :  (lam t [ x ::= u ]) ≡ lam t
?3  :  (app t t₁ [ x ::= u ]) ≡ app t t₁
?4  :  (Y t₁ [ x ::= u ]) ≡ Y t₁
```

As one can see, there is a clear correspondence between the 5 generated goals in Isabelle and the cases of the Agda proof above.

Due to this correspondence, reasoning in both systems is often largely similar. Whereas in Isabelle, one modifies the proof indirectly by issuing commands to modify proof goals, in Agda, one generates proofs directly by writing a program-as-proof, which satisfies the type constraints given in the definition.

## 5.2 Automation

As seen in the first example, Isabelle relies on automation in its proofs. It includes several automatic provers of varying complexity, including `simp`, `auto`, `blast`, `metis` and others. These are usually tactics/programs which automatically apply rewrite-rules, until the goal is discharged. If the tactic fails to discharge a goal within a set number of steps, it stops and lets the user direct the proof. The use of tactics in Isabelle is commonly used to prove trivial goals, which usually follow from simple rewriting of definitions or case analysis of certain variables.

**Example 5.1.** For example, the proof goal

```
⋀xa. x ∉ FV (FVar xa) ⟹ FVar xa [x ::= u] = FVar xa
```

will be proved by first unfolding the definition of substitution for `FVar`

```
(FVar xa)[x ::= u] = (if xa = x then u else FVar xa)
```

and then deriving `x ≠ xa` from the assumption `x ∉ FV (FVar xa)`.
Applying these steps explicitly, we get:

```
lemma subst_fresh: "x ∉ FV t ⟹ t[x ::= u] = t"
apply (induct t)
apply (subst subst.simps(1))
apply (drule subst[OF FV.simps(1)])
apply (drule subst[OF Set.insert_iff])
apply (drule subst[OF Set.empty_iff])
apply (drule subst[OF HOL.simp_thms(31)])
⋮
```

where the goal now has the following shape:

```
1. ⋀xa. x ≠ xa ⟹ (if xa = x then u else FVar xa) = FVar xa
```

From this point, the simplifier rewrites `xa = x` to `False` and `(if False then u else FVar xa)` to `FVar xa` in the goal.
The use of tactics and automated tools is heavily ingrained in Isabelle and it is actually impossible (i.e. impossible for me) to not use `simp` at this point in the proof, partly because one gets so used to discharging such trivial goals automatically and partly because it becomes nearly impossible to do the last two steps explicitly without having a detailed knowledge of the available commands and tactics in Isabelle (i.e. I don't).
Doing these steps explicitly quickly becomes cumbersome, as one needs to constantly look up the names of basic lemmas, such as `Set.empty_iff`, which is a simple rewrite rule `(?c ∈ {}) = False`.

Unlike Isabelle, Agda does not include nearly as much automation. The only proof search tool included with Agda is Agsy, which is similar, albeit often weaker than the `simp` tactic. It may therefore seem that Agda will be much more cumbersome to reason in than Isabelle. This, however, turns out not to be the case (at least in this formalization), in part due to Agda's type system and the powerful pattern matching as well as direct access to the proof goals. Automation did not play as major a part in this project as it might have, especially in this round of the comparison, since the LN mechanization had to be implemented from scratch and thus, the proofs written in Isabelle were only later modified to leverage some automation. However, since most proofs required induction, which theorem provers are generally not very good at performing without user guidance, the only place where automation was really apparent was in the case of a few lemmas involving equational reasoning, like the "open-swap" lemma:

> **Lemma 5.1.** $k \neq n \implies x \neq y \implies \{k \to x\}\{n \to y\}M = \{n \to y\}\{k \to x\}M$

Whilst in Isabelle, this was a trivial case of applying induction on the term $M$ and letting `auto` prove all the remaining cases. In Agda, this was a lot more painful, as the cases had to be constructed and proved more or less manually, yielding this rather long(er) proof:

```
^-^-swap : ∀ k n x y m -> ¬(k ≡ n) -> ¬(x ≡ y) ->
  [ k >> fv x ] ([ n >> fv y ] m) ≡ [ n >> fv y ] ([ k >> fv x ] m)
^-^-swap k n x y (bv i) k≠n x≠y with n ≟ i
^-^-swap k n x y (bv .n) k≠n x≠y | yes refl with k ≟ n
^-^-swap n .n x y (bv .n) k≠n x≠y | yes refl | yes refl = ⊥-elim (k≠n refl)
^-^-swap k n x y (bv .n) k≠n x≠y | yes refl | no _ with n ≟ n
^-^-swap k n x y (bv .n) k≠n x≠y | yes refl | no _ | yes refl = refl
^-^-swap k n x y (bv .n) k≠n x≠y | yes refl | no _ | no n≠n =
  ⊥-elim (n≠n refl)
^-^-swap k n x y (bv i) k≠n x≠y | no n≠i with k ≟ n
^-^-swap n .n x y (bv i) k≠n x≠y | no n≠i | yes refl = ⊥-elim (k≠n refl)
^-^-swap k n x y (bv i) k≠n x≠y | no n≠i | no _ with k ≟ i
^-^-swap k n x y (bv .k) k≠n x≠y | no n≠i | no _ | yes refl = refl
^-^-swap k n x y (bv i) k≠n x≠y | no n≠i | no _ | no k≠i with n ≟ i
^-^-swap k i x y (bv .i) k≠n x≠y | no n≠i | no _ | no k≠i | yes refl =
  ⊥-elim (n≠i refl)
^-^-swap k n x y (bv i) k≠n x≠y | no n≠i | no _ | no k≠i | no _ = refl
^-^-swap k n x y (fv z) k≠n x≠y = refl
^-^-swap k n x y (lam m) k≠n x≠y =
  cong lam (^-^-swap (suc k) (suc n) x y m (λ sk≡sn → k≠n (≡-suc sk≡sn)) x≠y)
^-^-swap k n x y (app t1 t2) k≠n x≠y rewrite
  ^-^-swap k n x y t1 k≠n x≠y | ^-^-swap k n x y t2 k≠n x≠y = refl
^-^-swap k n x y (Y _) k≠n x≠y = refl
```

## 5.3 Proofs-as-programs in Agda

As was already mentioned, Agda treats proofs as programs, and therefore provides direct access to proof objects. In Isabelle, the proof goal is usually of the form:

```
lemma x: "assm-1 ⇒ ... ⇒ assm-n ⇒ concl"
```

Using the 'apply-style' reasoning in Isabelle can become burdensome, if one needs to modify or

reason with the assumptions, as was seen in Example 5.1. In this example, the `drule` tactic, which is used to apply rules to the premises rather than the conclusion, was applied repeatedly. Other times, we might have to use structural rules for exchange or weakening, which are necessary purely for `organizational` purposes of the proof.
In Agda, such rules are not necessary, since the example above looks like a function definition:

```
x assm-1 ... assm-n = ?
```

Here, `assm-1` to `assm-n` are simply arguments to the function x, which expects something of type `concl` in the place of `?`. This presentation allows one to use the given assumptions arbitrarily, perhaps passing them to another function/proof or discarding them if not needed.
This way of reasoning is also supported in Isabelle to some extent, via the use of the Isar proof language, where (the snippet of) the proof of `subst_fresh` can be expressed in the following way:

```
lemma subst_fresh':
  assumes "x ∉ FV t"
  shows "t[x ::= u] = t"
using assms proof (induct t)
case (FVar y)
  from FVar.prems have "x ∉ {y}" unfolding FV.simps(1) .
  then have "x ≠ y" unfolding Set.insert_iff Set.empty_iff HOL.simp_thms(31) .
  then show ?case unfolding subst.simps(1) by simp
next
⋮
qed
```

This representation is more natural (and readable) to humans, as the assumptions have been separated and can be referenced and used in a clearer manner. For example, in the line

```
from FVar.prems have "x ∉ {y}"
```

the premise `FVar.prems` is added to the context of the goal `x ∉ {y}`:

```
proof (prove)
using this:
  x ∉ FV (FVar y)

goal (1 subgoal):
 1. x ∉ {y}
```

The individual reasoning steps described in the previous section have also been separated out into ‘mini-lemmas’ (the command `have` creates an new proof goal which has to be proved and then becomes available as an assumption in the current context) along the lines of the intuitive reasoning discussed initially. While this proof is more human readable, it is also more verbose and potentially harder to automate, as generating valid Isar style proofs is more difficult, due to ‘Isar-style’ proofs being obviously more complex than ‘apply-style’ proofs.

Whilst using the Isar proof language gives us a finer control and better structuring of proofs, one still references proofs only indirectly. Looking at the same proof in Agda, we have the following definition for the case of free variables:

```
subst-fresh' x (fv y) u x∉FVt = {!   0!}
```

---

```
?0  :  fv y [ x ::= u ] ≡ fv y
```

The proof of this case is slightly different from the Isabelle proof. In order to understand why, we need to look at the definition of substitution for free variables in Agda:

```
fv y [ x ::= u ] with x ≟ y
... | yes _ = u
... | no _ = fv y
```

This definition corresponds to the Isabelle definition, however, instead of using an if-then-else conditional, the Agda definition uses the `with` abstraction to pattern match on `x ≟ y`. The `_≟_` function takes the arguments `x` and `y`, which are natural numbers, and decides syntactic equality, returning a `yes p` or `no p`, where `p` is the proof object showing their in/equality.
Since the definition of substitution does not require the proof object of the equality of `x` and `y`, it is discarded in both cases. If `x` and `y` are equal, `u` is returned (case `... | yes _ = u`), otherwise `fv y` is returned.

In order for Agda to be able to unfold the definition of `fv y [ x ::= u ]`, it needs the case analysis on `x ≟ y`:

```
subst-fresh' x (fv y) u x∉FVt with x ≟ y
... | yes p = {!   0!}
... | no ¬p = {!   1!}
```

---

```
?0  :  (fv y [ x ::= u ] | yes p) ≡ fv y
?1  :  (fv y [ x ::= u ] | no ¬p) ≡ fv y
```

In the second case, when `x` and `y` are different, Agda can automatically fill in the hole with `refl`. Notice that unlike in Isabelle, where the definition of substitution had to be manually unfolded (the command `unfolding subst.simps(1)`), Agda performs type reduction automatically and can rewrite the term `(fv y [ x ::= u ] | no .¬p)` to `fv y` when type-checking the expression.

For the case where `x` and `y` are equal, one can immediately derive a contradiction from the fact that `x` cannot be equal to `y`, since `x` is not a free variable in `fv y`. The type of false propositions is ⊥ in Agda. Given ⊥, one can derive any proposition. To derive ⊥, we first inspect the type of `x∉FVt`, which is `x ∉ y :: []`. Further examining the definition of ∉, we find that `x ∉ xs = ¬ x ∈ xs`, which further unfolds to `x ∉ xs = x ∈ xs → ⊥`. Thus to obtain ⊥, we simply have to show that `x ∈ xs`, or in this specific instance `x ∈ y :: []`. The definition of ∈ is itself just sugar for `x ∈ xs = Any (_≈_ x) xs`, where `Any P xs` means that there is an element of the list `xs` which satisfies `P`. In this instance, `P = (_≈_ x)`, thus an inhabitant of the type `Any (_≈_ x) (y :: [])` can be constructed if one has a proof that at least one element in `y :: []` is equivalent to `x`. As it happens, such a proof was given as an argument in `yes p`:

```
False : ⊥
False = x∉FVt (here p)
```

The finished case looks like this (note that `⊥-elim` takes ⊥ and produces something of arbitrary type):

```
subst-fresh' x (fv y) u x∉FVt with x ≟ y
... | yes p = ⊥-elim False
  where
  False : ⊥
  False = x∉FVt (here p)
... | no ¬p = refl
```

We can even transform the Isabelle proof to closer match the Agda proof:

```
case (FVar y)
  show ?case
  proof (cases "x = y")
  case True
    with FVar have False by simp
    thus ?thesis ..
  next
  case False then show ?thesis unfolding subst.simps(1) by simp
  qed
```

Thus, we can see that using Isar style proofs and Agda reasoning ends up being rather similar in practice.

## 5.4 Pattern matching

Another reason why automation in the form of explicit proof search tactics needn't play such a significant role in Agda, is the more sophisticated type system of Agda (compared to Isabelle). Since Agda uses a dependent type system, there are often instances where the type system imposes certain constraints on the arguments/assumptions in a definition/proof and partially acts as a proof search tactic, by guiding the user through simple reasoning steps. Since Agda proofs are programs, unlike Isabelle 'apply-style' proofs, which are really proof scripts, one cannot intuitively view and step through the intermediate reasoning steps done by the user to prove a lemma. The way one proves a lemma in Agda is to start with a lemma with a 'hole', which is the proof goal, and iteratively refine the goal until this proof object is constructed. The way Agda's pattern matching makes constructing proofs easier can be demonstrated with the following example.

**Example 5.2.** The following lemma states that the parallel-$\beta$ maximal reduction preserves local closure:

$$t \ggg t' \implies \mathsf{term}(t) \wedge \mathsf{term}(t')$$

For simplicity, we will prove a slightly simpler version, namely: $t \ggg t' \implies \mathsf{term}(t)$. For comparison, this is a short, highly automated proof in Isabelle:

```
lemma pbeta_max_trm_r : "t >>> t' ⇒ trm t"
apply (induct t t' rule:pbeta_max.induct)
apply (subst trm.simps, simp)+
by (auto simp add: lam trm.Y trm.app)
```

In Agda, we start with the following definition:

```
>>>-Term-l : ∀ {t t'} -> t >>> t' -> Term t
>>>-Term-l t>>>t' = {!  0!}
```

---

```
?0  :  Term .t
```

Construction of this proof follows the Isabelle script, in that the proof proceeds by induction on $t \ggg t'$, which corresponds to the command `apply (induct t t' rule:pbeta_max.induct)`. As seen earlier, induction in Agda simply corresponds to a case split. The agda-mode in Emacs/Atom can perform a case split automatically, if supplied with the variable which should be used for the case analysis, in this case `t>>>t'`.

*Remark.* Note that Agda is very liberal with variable names, allowing almost any ASCII or Unicode characters, and it is customary to give descriptive names to the variables, usually denoting their type. In this instance, `t>>>t'` is a variable of type `t >>> t'`. Due to Agda's relative freedom in variable names, whitespace is important, as `t>>> t'` is very different from `t >>> t'` (the first is parsed a two variables `t>>>` and `t'`, whereas the second is parsed as the variable `t`, the relation symbol `>>>` and another variable `t'`).

```
>>>-Term-l : ∀ {t t'} -> t >>> t' -> Term t
>>>-Term-l refl = {!  0!}
>>>-Term-l reflY = {!  1!}
>>>-Term-l (app x t>>>t' t>>>t'') = {!  2!}
>>>-Term-l (abs L x) = {!  3!}
>>>-Term-l (beta L cf t>>>t') = {!  4!}
>>>-Term-l (Y t>>>t') = {!  5!}
```

---

```
?0  :  Term (fv .x)
?1  :  Term (Y .σ)
?2  :  Term (app .m .n)
?3  :  Term (lam .m)
?4  :  Term (app (lam .m) .n)
?5  :  Term (app (Y .σ) .m)
```

The newly expanded proof now contains 5 'holes', corresponding to the 5 constructors for the $>>>$ reduction. The first two goals are trivial, since any free variable or Y is a closed term. Here, one can use the agda-mode again, applying 'Refine', which is like a simple proof search, in that it will try to advance the proof by supplying an object of the correct type for the specified 'hole'. Applying 'Refine' to `{!   0!}` and `{!   1!}` yields:

```
>>>-Term-l : ∀ {t t'} -> t >>> t' -> Term t
>>>-Term-l refl = var
>>>-Term-l reflY = Y
>>>-Term-l (app x t>>>t' t>>>t'') = {!   0!}
>>>-Term-l (abs L x) = {!   1!}
>>>-Term-l (beta L cf t>>>t') = {!   2!}
>>>-Term-l (Y t>>>t') = {!   3!}
```

---

```
?0  :  Term (app .m .n)
?1  :  Term (lam .m)
?2  :  Term (app (lam .m) .n)
?3  :  Term (app (Y .σ) .m)
```

Since the constructor for `var` is `var : ∀ x -> Term (fv x)`, it is easy to see that the `hole` can be closed by supplying `var` as the proof of `Term (fv .x)`.
A more interesting case is the `app` case, where using 'Refine' yields:

```
>>>-Term-l : ∀ {t t'} -> t >>> t' -> Term t
>>>-Term-l refl = var
>>>-Term-l reflY = Y
>>>-Term-l (app x t>>>t' t>>>t'') = app {!   0!} {!   1!}
>>>-Term-l (abs L x) = {!   2!}
>>>-Term-l (beta L cf t>>>t') = {!   3!}
>>>-Term-l (Y t>>>t') = {!   4!}
```

---

```
?0  :  Term .m
?1  :  Term .n
?2  :  Term (lam .m)
?3  :  Term (app (lam .m) .n)
?4  :  Term (app (Y .σ) .m)
```

Here, the refine tactic supplied the constructor `app`, as it's type `app : ∀ e₁ e₂ -> Term e₁ -> Term e₂ -> Term (app e₁ e₂)` fit the 'hole' (`Term (app .m .n)`), generating two new 'holes', with the goal `Term .m` and `Term .n`. However, trying 'Refine' again on either of the 'holes' yields no result. This is where one applies the induction hypothesis, by adding `>>>-Term-l t>>>t'` to `{!   0!}` and applying 'Refine' again, which closes the 'hole' `{!   0!}`. Perhaps confusingly, `>>>-Term-l t>>>t'` produces a proof of `Term .m`. To see why this is, one has to inspect the type of `t>>>t'` in this context. Helpfully, the agda-mode provides just this function, which infers the type of `t>>>t'` to be `.m >>> .m'`. Similarly, `t>>>t''` has the type `.n >>> .n'`. Renaming `t>>>t'` and `t>>>t''` to `m>>>m'` and `n>>>n'` respectively, now makes the recursive call obvious:

```
>>>-Term-l : ∀ {t t'} -> t >>> t' -> Term t
>>>-Term-l refl = var
>>>-Term-l reflY = Y
>>>-Term-l (app x m>>>m' n>>>n') = app (>>>-Term-l m>>>m') {!   0!}
>>>-Term-l (abs L x) = {!   1!}
>>>-Term-l (beta L cf t>>>t') = {!   2!}
>>>-Term-l (Y t>>>t') = {!   3!}
```

---

```
?0  :  Term .n
?1  :  Term (lam .m)
?2  :  Term (app (lam .m) .n)
?3  :  Term (app (Y .σ) .m)
```

The goal `Term .n` follows in exactly the same fashion. Applying 'Refine' to the next 'hole' yields:

```
>>>-Term-l : ∀ {t t'} -> t >>> t' -> Term t
>>>-Term-l refl = var
>>>-Term-l reflY = Y
>>>-Term-l (app x m>>>m' n>>>n') =
  app (>>>-Term-l m>>>m') (>>>-Term-l n>>>n')
>>>-Term-l (abs L x) = lam {!   0!} {!   1!}
>>>-Term-l (beta L cf t>>>t') = {!   2!}
>>>-Term-l (Y t>>>t') = {!   3!}
```

---

```
?0  :  FVars
?1  :  {x = x₁ : ℕ} → x₁ ∉ ?0 L x → Term (.m ^' x₁)
?2  :  Term (app (lam .m) .n)
?3  :  Term (app (Y .σ) .m)
```

At this stage, the interesting goal is `?1`, due to the fact that it is dependent on `?0`. Indeed, replacing `?0` with `L` (which is the only thing of the type `FVars` available in this context) changes goal `?1` to `{x = x₁ : ℕ} → x₁ ∉ L → Term (.m ^' x₁)`:

```
>>>-Term-l : ∀ {t t'} -> t >>> t' -> Term t
>>>-Term-l refl = var
>>>-Term-l reflY = Y
>>>-Term-l (app x m>>>m' n>>>n') =
  app (>>>-Term-l m>>>m') (>>>-Term-l n>>>n')
>>>-Term-l (abs L x) = lam L {!   0!}
>>>-Term-l (beta L cf t>>>t') = {!   1!}
>>>-Term-l (Y t>>>t') = {!   2!}
```

---

```
?0  :  {x = x₁ : ℕ} → x₁ ∉ L → Term (.m ^' x₁)
?1  :  Term (app (lam .m) .n)
?2  :  Term (app (Y .σ) .m)
```

Since the goal/type of `{! 0!}` is `{x = x₁ : ℕ} → x₁ ∉ L → Term (.m ^' x₁)`, applying 'Refine' will generate a lambda expression `(λ x∉L → {! 0!})`, as this is obviously the only 'constructor' for a function type. Again, confusingly, we supply the recursive call `>>>-Term-l (x x∉L)` to `{! 0!}`. By examining the type of `x`, we get that `x` has the type `{x = x₁ : ℕ} → x₁ ∉ L → (.m ^' x₁) >>> (.m' ^' x₁)`. Then `(x x∉L)` is clearly of the type `(.m ^' x₁) >>> (.m' ^' x₁)`. Thus `>>>-Term-l (x x∉L)` has the desired type `Term (.m ^' .x)` (note that `.x` and `x` are not the same in this context).

Doing these steps explicitly was not in fact necessary, as the automatic proof search 'Agsy' is capable of automatically constructing proof objects for all of the cases above. Using 'Agsy' in both of the last two cases, the completed proof is given below:

```
>>>-Term-l : ∀ {t t'} -> t >>> t' -> Term t
>>>-Term-l refl = var
>>>-Term-l reflY = Y
>>>-Term-l (app x m>>>m' n>>>n') =
  app (>>>-Term-l m>>>m') (>>>-Term-l n>>>n')
>>>-Term-l (abs L x) = lam L (λ x∉L → >>>-Term-l (x x∉L))
>>>-Term-l (beta L cf t>>>t') = app
   (lam L (λ {x} x∉L → >>>-Term-l (cf x∉L)))
   (>>>-Term-l t>>>t')
>>>-Term-l (Y t>>>t') = app Y (>>>-Term-l t>>>t')
```

# 6. Intersection types

Having compared different mechanizations and implementation languages for the simply typed $\lambda$-$Y$ calculus in the previous two chapters, we arrived at the "winning" combination of a locally nameless mechanization using Agda. Carrying on in this setting, we present the formalization of intersection types for the $\lambda$-$Y$ calculus along with the proof of subject invariance for intersection types.

Whilst the theory formalized so far "only" includes the basic definitions of intersection type assignment and the proof of subject invariance, these proofs turned out to be significantly more difficult than their simply typed counterparts (e.g. in case of sub-tying and subject reduction lemmas). Indeed the whole formalization of simple types, along with the proof of the Church Rosser theorem, is roughly only 1350 lines of code in Agda, in comparison to about 1890 lines, for the intersection typing together with proofs of subject invariance.

Even though the proof is not novel, there is, to our knowledge, no known fully formal version of it for the $\lambda$-$Y$ calculus. The chapter mainly focuses on the engineering choices that were made in order to simplify the proofs as much as possible, as well as the necessary implementation overheads and compromises that were made.

The chapter is presented in sections, each explaining implementation details for a specific lemma or definition, introduced in Section 2.4. Some of the definitions presented early on in this chapter undergo several revisions, as we discuss the necessities for these changes in a pseudo-chronological manner, in which they arose during the implementation stage of this project.

## 6.1 Intersection types in Agda

The first implementation detail we had to consider was the implementation of the definition of intersection types themselves. Unlike simple types, the definition of intersection-types is split into two mutually recursive definitions of strict (`IType` $/\mathcal{T}_s$) and intersection (`ITypeℓ` $/\mathcal{T}$) types:

```
data ITypeℓ : Set
data IType : Set

data IType where
  φ : IType
  _~>_ : (s : ITypeℓ) -> (t : IType) -> IType

```

```
8  data ITypeℓ where
9    ∩ : List IType -> ITypeℓ
```

The reason why the intersection `ITypeℓ` is defined as a list of strict types `IType` in line 9, is due to the (usually) implicit requirement that the types in $\mathcal{T}$ be finite. The decision to use lists as an implementation of fine sets was taken, because the Agda standard library includes a definition of lists with definitions of list membership $\in$ and other associated lemmas, which proved to be useful for definitions of the $\subseteq$ relation on types.

From the above definition, it is obvious that the split definitions of `IType` and `ITypeℓ` are somewhat redundant, in that `ITypeℓ` only has one constructor ∩ and therefore, any instance of `ITypeℓ` in the definition of `IType` can simply be replaced by `List IType`:

```
data IType : Set where
  φ : IType
  _~>_ : List IType -> IType -> IType
```

## 6.2 Type refinement

One of the first things we needed to add to the notion of intersection type assignment (and as a result also to the $\subseteq$ relation on intersection types) was the notion of simple-type refinement. The main idea of intersection types for $\lambda$-$Y$ terms is for the intersection types to somehow "refine" the simple types. Intuitively, this notion should capture the relationship between the "shape" of the intersection and simple types.

To demonstrate the reason for introducing type refinement, we look at the initial formulation of the (intersection) typing rule $(Y)$:

$$(Y)\ \frac{}{\Gamma \Vdash Y_\sigma : (\bigcap_{\underline{n}} \tau_i \rightsquigarrow \tau_1 \cap \ldots \cap \bigcap_{\underline{n}} \tau_i \rightsquigarrow \tau_i) \rightsquigarrow \tau_j}\ (j \in \underline{n})$$

The lack of connection between simple and intersection types in the typing relation is especially apparent here, as $\bigcap_{\underline{n}} \tau_i$ seems to be chosen arbitrarily. Once we reformulate the above definition to include type refinement, the choice of $\bigcap_{\underline{n}} \tau_i$ makes more sense, since we know that $\tau_1, \ldots, \tau_i$ will somehow be related to the simple type $\sigma$:

$$(Y)\ \frac{\bigcap_{\underline{n}} \tau_i :: \sigma}{\Gamma \Vdash Y_\sigma : (\bigcap_{\underline{n}} \tau_i \rightsquigarrow \tau_1 \cap \ldots \cap \bigcap_{\underline{n}} \tau_i \rightsquigarrow \tau_i) \rightsquigarrow \tau_j}\ (j \in \underline{n})$$

The refinement relation has been defined in Kobayashi (2009) (amongst others) and is presented below:

**Definition 6.1.** [Intersection type refinement]

Since intersection types are defined in terms of strict ($\mathcal{T}_s$) and non-strict ($\mathcal{T}$) intersection types, the definition of refinement (::) is split into two versions, one for strict and another for non-strict types. In the definition below, $\tau$ ranges over strict intersection types $\mathcal{T}_s$, with $\tau_i, \tau_j$ ranging over non-strict intersection types $\mathcal{T}$, and $A, B$ range over simple types $\sigma$:

$$(base)\ \frac{}{\varphi :: \mathrm{o}} \qquad (arr)\ \frac{\tau_i ::_\ell A \qquad \tau_j ::_\ell B}{\tau_i \rightsquigarrow \tau_j :: A \rightarrow B}$$

$$(nil)\ \frac{}{\omega ::_\ell A} \qquad (cons)\ \frac{\tau :: A \qquad \tau_i ::_\ell A}{\tau, \tau_i ::_\ell A}$$

Having a notion of type refinement, we then modified the subset relation on intersection types, s.t. $\subseteq$ is defined only for pairs of intersection types, which refine the same simple type:

**Definition 6.2.** $[\subseteq^A]$

In the definition below, $\tau, \tau'$ range over $\mathcal{T}_s$, $\tau_i, \ldots, \tau_n$ range over $\mathcal{T}$ and $A, B$ range over $\sigma$:

$$(base)\ \frac{}{\varphi \subseteq^{\mathrm{o}} \varphi} \qquad (arr)\ \frac{\tau_i \subseteq^A_\ell \tau_j \qquad \tau_m \subseteq^B \tau_n}{\tau_j \rightsquigarrow \tau_m \subseteq^{A \rightarrow B} \tau_i \rightsquigarrow \tau_n}$$

$$(nil)\ \frac{\tau_i ::_\ell A}{\omega \subseteq^A_\ell \tau_i} \qquad (cons)\ \frac{\exists \tau' \in \tau_j.\ \tau \subseteq^A \tau' \qquad \tau_i \subseteq^A_\ell \tau_j}{\tau, \tau_i \subseteq^A_\ell \tau_j}$$

## 6.3 Well typed $\subseteq$

The presentation of the $\subseteq$ relation in Definition 2.8 differs quite significantly from the one presented above. The main difference is obviously the addition of type refinement, but the definition now also includes the $(base)$ rule, which allows one to derive the previously implicitly stated reflexivity and transitivity rules.
Another departure from the original definition is the formulation of the following two properties as the $(nil)$ and $(cons)$ rules:

$$\forall\, i \in \underline{n}.\ \ \tau_i \subseteq \bigcap\nolimits_{\underline{n}} \tau_i$$
$$\forall\, i \in \underline{n}.\ \ \tau_i \subseteq \tau \implies \bigcap\nolimits_{\underline{n}} \tau_i \subseteq \tau$$

To give a motivation as to why we chose a different formulation of these properties, we first examine the original definition and show why it's not rigorous enough for a well typed Agda definition. As we've shown in Section 6.1, the definition of intersection types is implicitly split into strict `IType`s and intersections, encoded as `List IType`s. All the preceding definitions follow this split with the strict and non strict versions of the type refinement (:: and $::_\ell$ respectively) and sub-typing relations ($\subseteq$ and $\subseteq_\ell$ respectively).
If we tried to turn the first property above into a rule, such as:

$$(prop'\ 1)\ \frac{\tau \in \tau_i}{\tau \subseteq \tau_i}$$

where $\tau$ is a strict type `IType` and $\tau_i$ is an intersection `List IType`, we would immediately get a type error, because the type signature of $\subseteq$ (which does not include type refinement) is:

```
data _⊆_ : IType -> IType -> Set
```

In order to get a well typed version of this rule, we would have to write something like:

$$(prop'\ 1)\ \frac{\tau \in \tau_i}{[\tau] \subseteq_\ell \tau_i}$$

Similarly for the second property, the well typed version might be formulated as:

$$(prop'\ 2)\ \frac{\forall \tau' \in \tau_i.\ [\tau'] \subseteq_\ell \tau}{\tau_i \subseteq_\ell \tau}$$

However, in the rule above, we assumed/forced $\tau$ to be an intersection, yet the property does not enforce this, and thus the two rules above do not actually capture the two properties from Definition 2.8.

**Example 6.1.** To demonstrate this, take the two intersection types $((\psi \cap \tau) \to \psi) \cap ((\psi \cap \tau \cap \rho) \to \psi)$ and $(\psi \cap \tau) \to \psi$. According to the original definition, we will have:

$$(prop\ 2)\ \dfrac{(refl)\ \dfrac{}{(\psi \cap \ldots} \qquad (prop\ 3)\ \dfrac{(prop\ 2)\ \dfrac{(prop\ 1)\ \dfrac{}{\psi \subseteq \psi \cap \tau \cap \rho} \qquad (prop\ 1)\ \dfrac{}{\tau \subseteq \psi \cap \tau \cap \rho}}{\psi \cap \tau \subseteq \psi \cap \tau \cap \rho} \qquad (refl)\ \dfrac{}{\psi \subseteq \psi}}{(\psi \cap \tau \cap \rho) \to \psi \subseteq (\psi \cap \tau) \to \psi}}{((\psi \cap \tau) \to \psi) \cap ((\psi \cap \tau \cap \rho) \to \psi) \subseteq (\psi \cap \tau) \to \psi}$$

When we try to prove the above using the well typed rules, we first need to coerce $(\psi \cap \tau) \to \psi$ into an intersection. Then, we try to construct the derivation tree:

$$(prop'\ 2)\ \dfrac{(refl)\ \dfrac{}{[[\psi, \tau] \to \psi] \subseteq_\ell [[\psi, \tau] \to \psi]} \qquad [[\psi, \tau, \rho] \to \psi] \subseteq_\ell [[\psi, \tau] \to \psi]}{[[\psi, \tau] \to \psi, [\psi, \tau, \rho] \to \psi] \subseteq_\ell [[\psi, \tau] \to \psi]}$$

The open branch $[[\psi, \tau, \rho] \to \psi] \subseteq_\ell [[\psi, \tau] \to \psi]$ in the example clearly demonstrates that the current formulation of the two properties clearly doesn't quite capture the intended meaning.

Since we know by reflexivity that $\tau \subseteq \tau$, we can reformulate $(prop'\ 1)$ as:

$$(prop''\ 1)\ \frac{\exists \tau' \in \tau_i.\ \tau \subseteq \tau'}{[\tau] \subseteq_\ell \tau_i}$$

Using this rule, we can now complete the previously open branch in the example above:

$$(prop'\ 2)\ \dfrac{\vdots \qquad (prop''\ 1)\ \dfrac{(arr)\ \dfrac{(prop'\ 2)\ \dfrac{(prop''\ 1)\ \dfrac{(refl)\ \dfrac{}{\psi \subseteq \psi}}{[\psi] \subseteq_\ell [\psi, \tau, \rho]} \qquad (prop''\ 1)\ \dfrac{(refl)\ \dfrac{}{\tau \subseteq \tau}}{[\tau] \subseteq_\ell [\psi, \tau, \rho]}}{[\psi, \tau] \subseteq_\ell [\psi, \tau, \rho]} \qquad (refl)\ \dfrac{}{\psi \subseteq \psi}}{[\psi, \tau, \rho] \to \psi \subseteq [\psi, \tau] \to \psi}}{[[\psi, \tau, \rho] \to \psi] \subseteq_\ell [[\psi, \tau] \to \psi]}}{[[\psi, \tau] \to \psi, [\psi, \tau, \rho] \to \psi] \subseteq_\ell [[\psi, \tau] \to \psi]}$$

Also, since the only rules that can proceed $(prop'\ 2)$ in the derivation tree are $(refl)$ or $(prop'\ 1)$, and it's easy to see that in case of $(refl)$ preceding, we can always apply $(prop'\ 1)$ before $(refl)$, we can in fact merge $(prop'\ 1)$ and $(prop'\ 2)$ into the single rule:

$$(prop'\ 12)\ \frac{\forall \tau' \in \tau_i.\ \exists \tau'' \in \tau.\ \tau' \subseteq \tau''}{\tau_i \subseteq_\ell \tau}$$

The final version of this rule, as it appears in Definition 6.2, is simply an iterated version, split into the $(nil)$ and $(cons)$ cases, to mirror the constructors of lists, since these rules "operate" with `List IType`. This iterated style of rules was adopted throughout this chapter for all definitions involving `List IType`, wherever possible, since it is more natural to work with, in Agda.

**Example 6.2.** To illustrate this, take the following lemma about type refinement:

**Lemma 6.1.** The following rule is admissible in the typing refinement relation $::_\ell$:

$$(+\!\!+)\ \frac{\tau_i ::_\ell A \qquad \tau_j ::_\ell A}{\tau_i +\!\!+ \tau_j ::_\ell A}$$

*Proof.* By induction on $\tau_i ::_\ell A$:

- $(nil)$: Therefore $\tau_i \equiv []$ and $[] +\!\!+ \tau_j \equiv \tau_j$. Thus $\tau_j ::_\ell A$ holds by assumption.
- $(cons)$: We have $\tau_i \equiv \tau, \tau_s$. Thus we know that $\tau :: A$ and $\tau_s ::_\ell A$. Then, by IH, we have $\tau_s +\!\!+ \tau_j ::_\ell A$ and thus:

$$(cons)\ \frac{(assm)\ \dfrac{}{\tau :: A} \qquad (IH)\ \dfrac{}{\tau_s +\!\!+ \tau_j ::_\ell A}}{\tau, \tau_s +\!\!+ \tau_j ::_\ell A}$$

□

For comparison, the same proof in Agda reads much the same as the "paper" one, given above:

```
++-::'ℓ : ∀ {A τᵢ τⱼ} -> τᵢ ::'ℓ A -> τⱼ ::'ℓ A -> (τᵢ ++ τⱼ) ::'ℓ A
++-::'ℓ nil τⱼ::'A = τⱼ::'A
++-::'ℓ (cons τ::'A τₛ::'A) τⱼ::'A = cons τ::'A (++-::'ℓ τₛ::'A τⱼ::'A)
```

## 6.4 Intersection-type assignment

Having modified the initial definition of sub-typing and added the notion of type refinement, we now take a look at the definition of intersection type assignment and the modifications that were needed for the mechanization.

Whilst before, intersection typing consisted of the triple $\Gamma \Vdash M : \tau)$, where $\Gamma$ was the intersection type context, $M$ was an untyped $\lambda$-$Y$ term and $\tau$ was an intersection type, this information is not actually sufficient when we introduce type refinement. As we've shown with the $(Y)$ rule, the refinement relation $::$ provides a connection between intersection and simple types. We therefore want $M$ in the triple to be a simply typed $\lambda$-$Y$ term.
Even though, we could use the definition of simple types from the previous chapters, this notation would be rather cumbersome.

**Example 6.3.** Consider the simply typed term $\{\} \vdash \lambda x.x : A \to A$ being substituted for the untyped $\lambda x.x$ in $\{\} \Vdash \lambda x.x : (\tau \cap \varphi) \leadsto \varphi$ (where $\tau :: A$ and $\varphi :: A$):

$$\{\} \Vdash (\{\} \vdash \lambda x.x : A \to A) : (\tau \cap \varphi) \leadsto \varphi$$

Already, this simple example demonstrates the clutter of using *Curry*-style simple types in conjunction with the intersection typing.

Instead of using the *a la Curry* simple typing, presented in the example above, we chose to define typed $\lambda$-$Y$ terms *a la Church*. However, since we are using the Locally Nameless representation of binders, we actually give the definition of simply typed pre-terms:

**Definition 6.3.** [Simply typed pre-terms a la Church]

For every simple type $A$, the set of simply typed pre-terms $\Lambda_A$ is inductively defined in the following way:

$$(fv)\ \frac{}{x \in \Lambda_A} \qquad (bv)\ \frac{}{n \in \Lambda_A} \qquad (app)\ \frac{s \in \Lambda_{B \to A} \qquad t \in \Lambda_B}{st \in \Lambda_A} \qquad (lam)\ \frac{s \in \Lambda_B}{\lambda_A.s \in \Lambda_{A \to B}}$$

$$(Y)\ \frac{}{Y_A \in \Lambda_{(A \to A) \to A}}$$

It's easy to see that the definition of Church-style simply typed $\lambda$-$Y$ pre-terms differs form the untyped pre-terms only in the $\lambda$ case, with the addition of the extra typing information, much like in the case of $Y$. We also adopt a typing convention, where we write $M_{\{A\}}$ to mean $M \in \Lambda_A$.

The next hurdle we faced in defining the intersection typing relation was the formulation of the $(Y)$ rule. The intuition behind this rule is to type a $Y_A$ constant with a type $\tau$ s.t. $\tau :: (A \to A) \to A$. If we used the $\lambda_\cap^{BCD}$ types (introduced in Section 2.4), we could easily have $\tau \equiv (\bigcap_{\underline{n}} \tau_i \leadsto \bigcap_{\underline{n}} \tau_i) \leadsto \bigcap_{\underline{n}} \tau_i$, where $\bigcap_{\underline{n}} \tau_i :: A$. However, as we have restricted ourselves to strict-intersection types, the initial definition for the $(Y)$ rule was the somewhat cumbersome:

$$(Y)\ \frac{\bigcap_{\underline{n}} \tau_i :: \sigma}{\Gamma \Vdash Y_\sigma : (\bigcap_{\underline{n}} \tau_i \leadsto \tau_1 \cap \ldots \cap \bigcap_{\underline{n}} \tau_i \leadsto \tau_i) \leadsto \tau_j}\ (j \in \underline{n})$$

The implementation of this rule clearly demonstrates the complexity, which made it difficult to reason with in proofs:

```
Y :    ∀ {Γ A τᵢ τ} -> (τᵢ::A : τᵢ ::'ℓ A) -> (τ∈τᵢ : τ ∈ τᵢ) ->
    ---------------------------------------------------------
    Γ ⊩ Y A : (∩ (Data.List.map (λ τₖ -> (∩ τᵢ ~> τₖ)) τᵢ) ~> τ)
```

Even though Agda's main strength is its the powerful pattern matching, it was quickly realized that pattern matching on the type `(∩ (Data.List.map (λ τₖ -> (∩ τᵢ ~> τₖ)) τᵢ) ~> τ)` is difficult due to the map function, which appears inside the definition.
Several modifications were made to the rule, until we arrived at it's current form. To create a compromise between the unrestricted intersection-types of $\lambda_\cap^{BCD}$, which made expressing the $(Y)$ rule much simpler, and the strict typing, which provided a restricted version of type derivation over the $\lambda_\cap^{BCD}$ system, we modified strict types to include intersections on both the left and right sub-terms of a strict type:

**Definition 6.4.** [Semi-strict intersection types]

$$\mathcal{T}_s ::= \varphi \mid (\mathcal{T}_s \cap \ldots \cap \mathcal{T}_s) \leadsto (\mathcal{T}_s \cap \ldots \cap \mathcal{T}_s)$$

*Remark.* Using strict intersection types and having two intersection typing relations $\Vdash$ and $\Vdash_\ell$ makes proving lemmas about the system much easier. The clearest example of this is the nice property of *inversion*, one gets "for free" with strict types. Take, for example, the term $\Gamma \Vdash uv : \tau$ (where for the puproses of this example, $uv$ is a term of the simply-typed $\lambda$-calculus and not a $\lambda$-$Y$ term). Since for the $\Vdash$ relation, $uv$ can only be given a strict intersection type, we can easily prove the following inversion lemma:

**Lemma 6.2.** [Inversion Lemma for $(app)$]

In the following lemma, $\tau_i$ is an intersection, i.e. a list of strict intersection types $\mathcal{T}_s$:

$$\Gamma \Vdash uv : \tau \iff \exists \tau_i.\ \Gamma \Vdash u : \tau_i \leadsto \tau \wedge \Gamma \Vdash_\ell v : \tau_i$$

Such a lemma is in fact not even needed in Agda, since the shape of the term $uv$ and the type $\tau$ uniquely determine that the derivation tree must have had an application of the $(app)$ rule at its base. In an Agda proof, such as:

```
sample-lemma Γ⊩uv:τ = ?
```

One can perform a case analysis on the variable `Γ⊩uv:τ` (the type of which is `Γ ⊩ app u v : τ`) and obtain:

```
sample-lemma (app Γ⊩u:τᵢ~>τ Γ⊩v:τᵢ) = ?
```

In the $\lambda_\cap^{BCD}$ system (or similar), such an inversion lemma would be a lot more complicated and might look something like:

$$\begin{aligned}\Gamma \Vdash uv : \tau \iff & \exists k \geq 1.\ \exists \tau_1, \ldots, \tau_k, \psi_1, \ldots, \psi_k. \\ & \tau \subseteq \psi_1 \cap \ldots \cap \psi_k \wedge \\ & \forall i \in \{1, \ldots, k\}.\ \Gamma \Vdash u : \tau_i \leadsto \psi_i \wedge \Gamma \Vdash v : \tau_i\end{aligned}$$

The semi-strict typing loses some of the advantages of the strict types, as we will later modify the typing relation, losing the "free" *inversion* properties that we currently have, i.e. for a given term $uv$, Lemma 6.2 won't be trivial any more. However, the complexity of the inversion lemmas for semi-strict typing is still lower than that of the unrestricted intersection-typing systems.

The final version of the $(Y)$ rule, along with the other modified rules of the typing relation are presented below:

**Definition 6.5.** [Intersection-type assignment]

This definition assumes that the typing context $\Gamma$, which is a list of triples $(x, \tau_i, A)$, is well formed. For each triple, written as $x : \tau_i ::_\ell A$, this means that the free variable $x$ does not appear elsewhere in the domain of $\Gamma$. Each intersection type $\tau_i$, associated with a variable $x$, also refines a simple type $A$. In the definition below, we also assume the following convention $\bigcap \tau \equiv [\tau]$:

$$(var)\ \frac{\exists (x : \tau_i ::_\ell A) \in \Gamma.\ \bigcap \tau \subseteq^A_\ell \tau_i}{\Gamma \Vdash x_{\{A\}} : \tau}$$

$$(app)\ \frac{\Gamma \Vdash u_{\{A\to B\}} : \tau_i \rightsquigarrow \tau_j \qquad \Gamma \Vdash_\ell v_{\{A\}} : \tau_i}{\Gamma \Vdash uv : \tau}\ (\bigcap \tau \subseteq^B_\ell \tau_j)$$

$$(abs)\ \frac{\forall x \notin L.\ (x : \tau_i ::_\ell A), \Gamma \Vdash_\ell m^x : \tau_j}{\Gamma \Vdash \lambda_A.m : \tau_i \rightsquigarrow \tau_j} \qquad (Y)\ \frac{\exists \tau_x.\ \bigcap(\tau_x \rightsquigarrow \tau_x) \subseteq^{A\to A}_\ell \tau_i \wedge \tau_j \subseteq^A_\ell \tau_x}{\Gamma \Vdash_s Y_A : \tau_i \rightsquigarrow \tau_j}$$

$$(nil)\ \frac{}{\Gamma \Vdash_\ell m : \omega} \qquad (cons)\ \frac{\Gamma \Vdash m : \tau \qquad \Gamma \Vdash_\ell m : \tau_i}{\Gamma \Vdash_\ell m : \tau, \tau_i}$$

## 6.5 Proof of subject expansion

An interesting property of the intersection types is the fact that they admit both subject expansion and subject reduction, namely $\Vdash$ is closed under $\beta$-equality. In this section, we will focus on the subject expansion lemma:

**Theorem 6.1.** [Subject expansion for $\Vdash$ / $\Vdash_\ell$]

i) $\Gamma \Vdash m : \tau \implies m \Rightarrow_Y m' \implies \Gamma \Vdash m' : \tau$
ii) $\Gamma \Vdash_\ell m : \tau_i \implies m \Rightarrow_Y m' \implies \Gamma \Vdash_\ell m' : \tau_i$

The proof of this theorem follows by induction on the $\beta$-reduction $m \Rightarrow_Y m'$. We will focus on the $(Y)$ reduction rule and show that given a well typed term $\Gamma \Vdash m(Y_\sigma m) : \tau$, s.t. $Y_\sigma m \Rightarrow_Y m(Y_\sigma m)$, we can also type $Y_\sigma m$ with the same intersection type $\tau$.
We will start with a very high-level overview of the proof. Having assumed, $\Gamma \Vdash m(Y_\sigma m) : \tau$, we must necessarily have the following derivation tree for the intersection typing relation $\Vdash$:

$$(app)\ \frac{\dfrac{\vdots}{\Gamma \Vdash_s m : \tau_i \rightsquigarrow \tau_j} \qquad \dfrac{\vdots}{\Gamma \Vdash_\ell Y_A m : \tau_i}}{\Gamma \Vdash m(Y_A m) : \tau}\ ([\tau] \subseteq^A_\ell \tau_j)$$

**Figure 6.1: Analysis of the shape of the derivation tree for $\Gamma \Vdash m(Y_A m) : \tau$**

We have two cases, where $\tau_i$ is an empty intersection $\tau_i \equiv \omega$ or a non-empty list of strict intersection types $\tau_i \equiv [\tau_1, \ldots, \tau_n]$.

### 6.5.1 $\tau_i \equiv \omega$

From Figure 6.1 above, we have $(1) : \Gamma \Vdash_s m : \omega \rightsquigarrow \tau_j$. Then, we can construct the following proof tree:

$$(app)\ \frac{(Y)\ \dfrac{[[\tau] \rightsquigarrow [\tau]] \subseteq^{A\to A}_\ell [\omega \rightsquigarrow [\tau]] \wedge [\tau] \subseteq^A_\ell [\tau]}{\Gamma \Vdash Y : [\omega \rightsquigarrow [\tau]] \rightsquigarrow [\tau]} \qquad (cons)\ \dfrac{\Gamma \Vdash m : \omega \rightsquigarrow [\tau] \qquad (nil)\ \dfrac{}{\Gamma \Vdash_\ell m : \omega}}{\Gamma \Vdash_\ell m : [\omega \rightsquigarrow [\tau]]}}{\Gamma \Vdash Y_A m : \tau}$$

The only (non-trivial) open branch in the above tree is to show that $\Gamma \Vdash m : \omega \rightsquigarrow [\tau]$. In order to do so, we need to use the sub-typing lemma for intersection types:

**Lemma 6.3.** [Sub-typing for $\Vdash$ / $\Vdash_\ell$]

In the definition below, the binary relation $\subseteq_\Gamma$ is defined for any well-formed contexts $\Gamma$ and $\Gamma'$, where for each triple $(x : \tau_i ::_\ell A) \in \Gamma$, there is a corresponding triple $(x : \tau_j ::_\ell A) \in \Gamma'$ s.t. $\tau_i \subseteq^A_\ell \tau_j$ :

$$(\subseteq)\ \frac{\Gamma \Vdash m_{\{A\}} : \tau}{\Gamma' \Vdash m_{\{A\}} : \tau'}\ (\Gamma \subseteq_\Gamma \Gamma', \tau' \subseteq^A \tau) \qquad (\subseteq_\ell)\ \frac{\Gamma \Vdash_\ell m_{\{A\}} : \tau_i}{\Gamma' \Vdash_\ell m_{\{A\}} : \tau_j}\ (\Gamma \subseteq_\Gamma \Gamma', \tau_j \subseteq^A_\ell \tau_i)$$

*Proof.* Ommited.

□

Thus, we have:

$$(\subseteq)\ \frac{(1)\ \dfrac{}{\Gamma \Vdash m : \omega \rightsquigarrow \tau_j}}{\Gamma \Vdash m : \omega \rightsquigarrow [\tau]}\ (\Gamma \subseteq_\Gamma \Gamma, \omega \rightsquigarrow [\tau] \subseteq^{A \to A} \omega \rightsquigarrow \tau_j)$$

### 6.5.2 $\tau_i \equiv [\tau_1, \ldots, \tau_n]$

This case of the proof is a lot more involved and required several additional rules and lemmas. We will outline the main ideas of the proof in this section.

*Remark.* The first thing to note is that since $\tau_i$ is a non-empty list of semi-strict intersection types, it will have the shape:

$$(cons)\ \frac{\dfrac{\vdots}{\Gamma \Vdash Y_A m : \tau_1} \quad (cons)\ \dfrac{\dfrac{\vdots}{\Gamma \Vdash Y_A m : \tau_2} \quad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma \Vdash Y_A m : \tau_n} \quad (nil)\ \dfrac{}{\Gamma \Vdash_\ell Y_A m : \omega}}{\Gamma \Vdash_\ell Y_A m : [\tau_n]}}{\vdots}}{\Gamma \Vdash_\ell Y_A m : [\tau_2, \ldots, \tau_n]}}{\Gamma \Vdash_\ell Y_A m : \tau_i}$$

We will simplify this notation slightly and just write:

$$\frac{\dfrac{\vdots}{\Gamma \Vdash Y_A m : \tau_1} \quad \dfrac{\vdots}{\Gamma \Vdash Y_A m : \tau_2} \quad \ldots \quad \dfrac{\vdots}{\Gamma \Vdash Y_A m : \tau_n}}{\Gamma \Vdash_\ell Y_A m : \tau_i}$$

In order to type $Y_A m$ with $\tau$, we first have to show that for every tree above, we can find a type $\tau'_k$ s.t. $\Gamma \Vdash m : \tau'_k \rightsquigarrow \tau'_k$ and $[\tau_k] \subseteq^A_\ell \tau'_k$:

**Lemma 6.4.** $\Gamma \Vdash Y_A m : \tau \implies \exists \tau'. \Gamma \Vdash_\ell m : [\tau' \rightsquigarrow \tau'] \land [\tau] \subseteq^A_\ell \tau'$

*Proof.* Unfolding the typing tree of $\Gamma \Vdash Y_A m : \tau$, we have:

$$
(app)\ \dfrac{\dfrac{[\tau_x \rightsquigarrow \tau_x] \subseteq^{A \to A} \tau_i \wedge \tau_j \subseteq^{A} \tau_x}{\Gamma \Vdash_s Y_A : \tau_i \rightsquigarrow \tau_j} \qquad \Gamma \Vdash_\ell m : \tau_i}{\Gamma \Vdash Y_A m : \tau}\ ([\tau] \subseteq^{A}_{\ell} \tau_j)
$$

Then it follows by transitivity, that $[\tau] \subseteq^{A}_{\ell} \tau_x$, and $\Gamma \Vdash_\ell m : [\tau_x \rightsquigarrow \tau_x]$ by sub-typing:

$$
(\subseteq_\ell)\ \dfrac{\Gamma \Vdash_\ell m : \tau_i}{\Gamma \Vdash_\ell m : [\tau_x \rightsquigarrow \tau_x]}\ (\Gamma \subseteq_\Gamma \Gamma, [\tau_x \rightsquigarrow \tau_x] \subseteq^{A \to A}_{\ell} \tau_i) \qquad (trans)\ \dfrac{[\tau] \subseteq^{A}_{\ell} \tau_j \qquad \tau_j \subseteq^{A}_{\ell} \tau_x}{[\tau] \subseteq^{A}_{\ell} \tau_x}
$$

□

Since for every type $\tau_k \in \tau_i$ we now have $\Gamma \Vdash_\ell m : [\tau'_k \rightsquigarrow \tau'_k]$ as well as $[\tau_k] \subseteq^{A}_{\ell} \tau'_k$, we want to "mege" all these types given to $m$:

$$
(???)\ \dfrac{(\text{Lemma 6.4})\ \dfrac{\Gamma \Vdash Y_A m : \tau_1}{\Gamma \Vdash_\ell m : [\tau'_1 \rightsquigarrow \tau'_1]} \quad \dots \quad (\text{Lemma 6.4})\ \dfrac{\Gamma \Vdash Y_A m : \tau_n}{\Gamma \Vdash_\ell m : [\tau'_n \rightsquigarrow \tau'_n]}}{\Gamma \Vdash m : \tau'_1 +\!\!+ \dots +\!\!+ \tau'_n \rightsquigarrow \tau'_1 +\!\!+ \dots +\!\!+ \tau'_n}
$$

such that we have $\tau'_i \equiv \tau'_1 +\!\!+ \dots +\!\!+ \tau'_n$ where $\tau_i \subseteq^{A}_{\ell} \tau'_i$.

To illustrate how to prove the last step in the tree above (i.e. how to derive the (???) rule), we will look at a simpler example, where $\tau_i \equiv [\tau_1, \tau_2]$. Thus, we want to show:

$$
\dfrac{(\text{Lemma 6.4})\ \dfrac{\Gamma \Vdash Y_A m : \tau_1}{\Gamma \Vdash_\ell m : [\tau'_1 \rightsquigarrow \tau'_1]} \qquad (\text{Lemma 6.4})\ \dfrac{\Gamma \Vdash Y_A m : \tau_2}{\Gamma \Vdash_\ell m : [\tau'_2 \rightsquigarrow \tau'_2]}}{\Gamma \Vdash m : \tau'_1 +\!\!+ \tau'_2 \rightsquigarrow \tau'_1 +\!\!+ \tau'_2}
$$

Using the sub-typing lemma we can show:

$$
(\subseteq_\ell)\ \dfrac{\Gamma \Vdash_\ell m : [\tau'_1 \rightsquigarrow \tau'_1]}{\Gamma \Vdash_\ell m : [\tau'_1 +\!\!+ \tau'_2 \rightsquigarrow \tau'_1]}\ ([\tau'_1 +\!\!+ \tau'_2 \rightsquigarrow \tau'_1] \subseteq^{A \to A}_{\ell} [\tau'_1 \rightsquigarrow \tau'_1])
$$

since we have:

$$
(cons)\ \dfrac{\dfrac{(\subseteq^*)^{1}\ \dfrac{}{\tau'_1 \subseteq^{A} \tau'_1 +\!\!+ \tau'_2} \qquad (refl)\ \dfrac{}{\tau'_1 \subseteq^{A} \tau'_1}}{\tau'_1 +\!\!+ \tau'_2 \rightsquigarrow \tau'_1 \subseteq^{A \to A} \tau'_1 \rightsquigarrow \tau'_1} \qquad (nil)\ \dfrac{}{\omega \subseteq^{A \to A}_{\ell} [\tau'_1 \rightsquigarrow \tau'_1]}}{[\tau'_1 +\!\!+ \tau'_2 \rightsquigarrow \tau'_1] \subseteq^{A \to A}_{\ell} [\tau'_1 \rightsquigarrow \tau'_1]}
$$

Similarly, we can also prove $\Gamma \Vdash_\ell m : [\tau'_1 +\!\!+ \tau'_2 \rightsquigarrow \tau'_2]$, but at this point there is no way we can merge these two types, to produce $\Gamma \Vdash_\ell m : [\tau'_1 +\!\!+ \tau'_2 \rightsquigarrow \tau'_1 +\!\!+ \tau'_2]$.
In order to proceed, we had to introduce a new rule to the typing relation $\Vdash$, to allow us to derive the type above:

$$
(\rightsquigarrow\cap)\ \dfrac{\Gamma \Vdash m_{\{A \to B\}} : \tau_i \rightsquigarrow \tau_j \qquad \Gamma \Vdash m_{\{A \to B\}} : \tau_i \rightsquigarrow \tau_k}{\Gamma \Vdash m_{\{A \to B\}} : \tau_i \rightsquigarrow \tau_{jk}}\ (\tau_{jk} \subseteq^{B} \tau_j +\!\!+ \tau_k)
$$

[1]This is a derived rule defined for the subset relation on lists, i.e. if the list $\tau_k$ is a subset of $\tau_n$, then it is also a subtype of $\tau_n$. Th derivation of this rule trivially follows from the $(refl)$, $(nil)$ and $(cons)$ rules.

Introducing this rule created a host of complications, the chief of which was the fact that we lost our "free" inversion lemmas, as it is now no longer obvious from the shape of the term, which rule was used last in the type derivation tree.

**Example 6.4.** Consider a term $\lambda_A.m$ s.t. $\Gamma \Vdash \lambda_A.m : \tau$. Since $\tau$ must necessarily be of the shape $\psi_i \rightsquigarrow \psi_j$, either of these two derivation trees could be valid:

$$(abs)\ \frac{\dfrac{\vdots}{\forall x \notin L.\ (x : \psi_i ::_\ell A), \Gamma \Vdash_\ell (m^x)_{\{B\}} : \psi_j}}{\Gamma \Vdash (\lambda_A.m)_{\{A \to B\}} : \psi_i \rightsquigarrow \psi_j}$$

$$(\rightsquigarrow\cap)\ \frac{\dfrac{\vdots}{\Gamma \Vdash_s (\lambda_A.m)_{\{A\to B\}} : \psi_i \rightsquigarrow \tau_j} \qquad \dfrac{\vdots}{\Gamma \Vdash_s (\lambda_A.m)_{\{A\to B\}} : \psi_i \rightsquigarrow \tau_k}}{\Gamma \Vdash_s (\lambda_A.m)_{\{A\to B\}} : \psi_i \rightsquigarrow \psi_j}\ (\psi_j \subseteq^B \tau_j +\!\!+ \tau_k)$$

However, it's easy to see that since these derivation trees must be finite, if we apply $(\rightsquigarrow\cap)$ multiple times, eventually, all of the branches will have to have an application of the $(abs)$ rule.

Besides having to derive inversion lemmas for the $(abs)$ and $(Y)$ rules, another derived rule, namely the sub-typing rule, breaks. In order to fix this rule, we also had to add an axiom scheme corresponding to the $(\rightsquigarrow\cap)$ rule, to the definition of the type subset relation:

$$(\rightsquigarrow\cap_\ell)\ \frac{[\tau_i \rightsquigarrow (\tau_j +\!\!+ \tau_k)] +\!\!+ \tau_m ::_\ell A \to B}{[\tau_i \rightsquigarrow (\tau_j +\!\!+ \tau_k)] +\!\!+ \tau_m \subseteq_\ell^{A\to B} [\tau_i \rightsquigarrow \tau_j,\ \tau_i \rightsquigarrow \tau_k] +\!\!+ \tau_m}$$

*Remark.* Initially, we tried to simply add $(\rightsquigarrow\cap_\ell)$ to the type subset relation and add the sub-typing rule to $\Vdash$, instead of having the rule $(\rightsquigarrow\cap)$. However, this made the inversion lemmas, as well as some other lemmas too difficult to prove. Finding the right balance in the formalization of the $(\rightsquigarrow\cap)/(\rightsquigarrow\cap_\ell)$ and the sub-typing rules proved to be perhaps the most challenging part of the formalization of intersection types.

We can now finish or proof for the case where $\tau_i \equiv [\tau_1, \tau_2]$:

$$(\rightsquigarrow\cap)\ \frac{(\in_\ell)^2\ \dfrac{(\subseteq_\ell)\ \dfrac{(\text{Lemma 6.4})\ \dfrac{\Gamma \Vdash Y_A m : \tau_1}{\Gamma \Vdash_\ell m : [\tau_1' \rightsquigarrow \tau_1']}}{\Gamma \Vdash_\ell m : [\tau_1' +\!\!+ \tau_2' \rightsquigarrow \tau_1']}}{\Gamma \Vdash m : \tau_1' +\!\!+ \tau_2' \rightsquigarrow \tau_1'} \qquad (\in_\ell)\ \dfrac{(\subseteq_\ell)\ \dfrac{(\text{Lemma 6.4})\ \dfrac{\Gamma \Vdash Y_A m : \tau_2}{\Gamma \Vdash_\ell m : [\tau_2' \rightsquigarrow \tau_2']}}{\Gamma \Vdash_\ell m : [\tau_1' +\!\!+ \tau_2' \rightsquigarrow \tau_2']}}{\Gamma \Vdash m : \tau_1' +\!\!+ \tau_2' \rightsquigarrow \tau_2'}}{\Gamma \Vdash m : \tau_1' +\!\!+ \tau_2' \rightsquigarrow \tau_1' +\!\!+ \tau_2'}$$

Having demonstrated the case when $\tau_i \equiv [\tau_1, \tau_2]$, the proof when $\tau_i$ is arbitrarily long proceeds in much the same way:

[2] The derived rule $(\in_\ell)$ is used to convert between the strict typing relation $\Vdash$ and intersection typing $\Vdash_\ell$ and is stated as: $\Gamma \Vdash_\ell m : \tau_i \wedge \tau \in \tau_i \implies \Gamma \Vdash m : \tau$

**Lemma 6.5.** Assuming $\tau_i$ is a non-empty intersection, we have:
$\Gamma \Vdash_\ell Y_A m : \tau_i \implies \exists \tau_i'.\, \Gamma \Vdash m : \tau_i' \rightsquigarrow \tau_i' \land \tau_i \subseteq^A_\ell \tau_i'$

Using Lemma 6.5, we can finally show that $\Gamma \Vdash Y_A m : \tau$.

First, from Figure 6.1, we have: $(1) : \Gamma \Vdash m : \tau_i \rightsquigarrow \tau_j$ ,
$(2) : [\tau] \subseteq^A_\ell \tau_j$ and
$(3) : \Gamma \Vdash_\ell Y_A m : \tau_i$.

Since $\tau_i$ is not empty, from (3) and Lemma 6.5, we have some $\tau_i'$ s.t. $(4) : \Gamma \Vdash m : \tau_i' \rightsquigarrow \tau_i'$ and
$(5) : \tau_i \subseteq^A_\ell \tau_i'$.

We can therefore derive $\Gamma \Vdash_\ell m : [\tau_i' \rightsquigarrow ([\tau] +\!\!+ \tau_i')]$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \Vdash m : \tau_i \rightsquigarrow \tau_j}(1)}{\Gamma \Vdash m : \tau_i' \rightsquigarrow [\tau]}(\subseteq)\;(\tau_i' \rightsquigarrow [\tau] \subseteq^{A\to A} \tau_i \rightsquigarrow \tau_j) \qquad \dfrac{}{\Gamma \Vdash m : \tau_i' \rightsquigarrow \tau_i'}(4)}{\Gamma \Vdash m : \tau_i' \rightsquigarrow ([\tau] +\!\!+ \tau_i')}(\rightsquigarrow\cap) \qquad \dfrac{}{\Gamma \Vdash_\ell m : \omega}(nil)}{\Gamma \Vdash_\ell m : [\tau_i' \rightsquigarrow ([\tau] +\!\!+ \tau_i')]}(cons)$$

*Remark.* In the sub-typing rule, used in the tree above, $\tau_i' \rightsquigarrow [\tau] \subseteq^{A\to A} \tau_i \rightsquigarrow \tau_j$ follows by:

$$\dfrac{\dfrac{}{\tau_i \subseteq^A_\ell \tau_i'}(5) \qquad \dfrac{}{[\tau] \subseteq^A_\ell \tau_j}(2)}{(\tau_i' \rightsquigarrow [\tau] \subseteq^{A\to A} \tau_i \rightsquigarrow \tau_j)}(arr)$$

Finally, putting all the pieces together, we get $\Gamma \Vdash Y_A m : \tau$:

$$\dfrac{\dfrac{\begin{array}{c}[([\tau] +\!\!+ \tau_i') \rightsquigarrow ([\tau] +\!\!+ \tau_i')] \subseteq^{A\to A}_\ell [\tau_i' \rightsquigarrow ([\tau] +\!\!+ \tau_i')] \land \\ [\tau] \subseteq^A_\ell [\tau] +\!\!+ \tau_i'\end{array}}{\Gamma \Vdash Y : [\tau_i' \rightsquigarrow ([\tau] +\!\!+ \tau_i')] \rightsquigarrow [\tau]}(Y) \qquad \Gamma \Vdash_\ell m : [\tau_i' \rightsquigarrow ([\tau] +\!\!+ \tau_i')]}{\Gamma \Vdash Y_A m : \tau}(app)$$

## 6.6 Proofs of termination for the LN representation

This final section of the chapter briefly describes an interesting implementation quirk/overhead, encountered when proving the substitution lemma, which was required for the proofs of both subject expansion and reduction:

**Lemma 6.6.** [Substitution lemma]

Given that $m$ and $n$ are both well formed terms and $x \notin \mathbf{dom}\ \Gamma$, we have:

$$\Gamma \Vdash m_{\{A\}}[n_{\{B\}}/x] : \tau \iff \exists \tau_i.\ (x : \tau_i ::_\ell B), \Gamma \Vdash m_{\{A\}} : \tau \land \Gamma \Vdash_\ell n_{\{B\}} : \tau_i$$

In the backwards direction ($\Leftarrow$), this proof is fairly straight forward and follows much like the homonymous lemma for simple types.

The other direction ($\Rightarrow$), used in the proof of subject expansion, turned out to be more complicated in Agda. This part of the proof proceeds by induction on the well formed term $m$, and whilst trying to prove the goal, when $m$ is a $\lambda$-term, Agda's termination checker would fail. To show why this was the case, we first examine the definition for the $\lambda$-case:

```
subst-⊩-2 : ∀ {A B Γ τ x} ->
  {m : Λ A} {n : Λ B} ->
  ΛTerm m -> ΛTerm n ->
  x ∉ dom Γ -> Γ ⊩ (m Λ[ x ::= n ]) : τ ->
  ∃(λ τᵢ -> ( ((x , τᵢ , B) :: Γ) ⊩ m : τ ) × ( Γ ⊩ℓ n : τᵢ ))
⋮
subst-⊩-2 {A → B} {C} {Γ} {τ ~> τ'} {x}
  {lam .A p} {n}
  (lam L {.p} cf) trm-n
  x∉Γ (abs L' cf') = ?
```

Informally, the (pieces of) definition above can be read as:

- `lam .A p` : $m \equiv \lambda_A.p$
- `(lam L .p cf)` : $p$ is a well formed $\lambda$-term, s.t. we have $\forall x' \notin L.\ \textbf{term}(p^{x'})$ for some finite $L$.
  (This is captured by the type of `cf`, which is `x₁ ∉ L → ΛTerm (Λ[ 0 >> fv x₁ ] p)`.)
- `trm-n` : $n$ is a well-formed $\lambda$-$Y$ term
- `(abs L' cf')` : the last rule in the derivation tree of $\Gamma \Vdash \lambda_A.p_{\{B\}}[n_{\{C\}}/x] : \tau \rightsquigarrow \tau'$ was the $(abs)$ rule and therefore we have `cf'`, which encodes the premise, that there is some finite $L'$ s.t. $\forall x' \notin L'.\ (x' : \tau ::_\ell A), \Gamma \Vdash_\ell (p[n/x])^{x'} : \tau$.

The proof proceeds, by first showing that we can obtain a fresh $x'$ s.t. $\textbf{term}(p^{x'})$. By picking a sufficiently fresh $x'$, we can also derive $(x' : \tau ::_\ell A), \Gamma \Vdash_\ell (p^{x'})[n/x] : \tau$, essentially swapping the substitution and opening from the assumption above.
However, when we then try to apply the induction hypothesis, which corresponds to a recursive call in Agda, we get an error, claiming that termination checking failed. In order to see why this happens, we will ignore the explicit arguments passed to `subst-⊩ℓ-2` and instead focus on the implicit arguments:

```
ih = subst-⊩ℓ-2 {A} {C} {(x' , τ , A) :: Γ} {τ'} {x}
  {Λ[ 0 >> fv x' ] p} {n} ...
```

Agda's termination checking relies on the fact that the data-types, being pattern matched on, get structurally smaller in recursive calls[3]. Thus, the parameter `Λ[ 0 >> fv x' ] m` in this definition is obviously problematic, as Agda doesn't know that $p^{x'}$ is structurally smaller than $m$ (i.e. $\lambda_A.p$), even though we know that is the case, as the open operation simply replaces a bound variable with a free one. However, whilst $p^{x'}$ is not "bigger" than $p$, it is not, strictly speaking, structurally smaller than $\lambda_A.p$ and therefore, structural induction/recursion principles cannot be used in this definition.

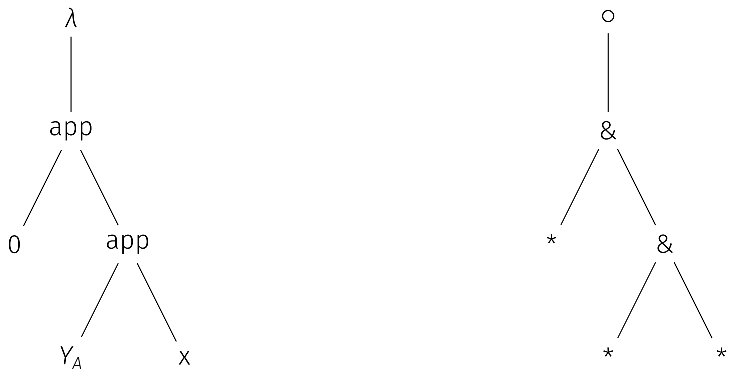
Whilst one can suppress termination checking for a specific definition/lemma in Agda, by adding

[3]The details on how the termination checking algorithm in Agda works are sparse, so we are not actually sure about the specifics of how the termination check fails.

the {-# TERMINATING #-} pragma in front of the definition, it is generally not a good idea to do this, even though we know that the definition is actually terminating. We initially contemplated using well-founded recursion to prove that the proof terminates, but having little experience in Agda, this looked quite complicated.
Instead, we devised a simple "hack" to allow Agda to prove termination for this (slightly modified) lemma, by first defining "skeleton" terms $T$, which capture the structure of $\lambda$-$Y$ terms:

**Definition 6.6.** $T ::= * \mid \circ T \mid T \,\&\, T$

**Example 6.5.** As an illustration, take the LN $\lambda$-$Y$ term $\lambda.0(Y_A x)$. We can represented this term as a tree (left). Then, we simply replace any $\lambda$ and $Y_\sigma$ with $\circ$, application becomes & and any free or bound variables are represented as * in the skeleton tree (on the right):




Thus, the skeleton term of $\lambda.0(Y_A x)$ is $\circ(* \,\&\, (* \,\&\, *))$.

Next, we defined the congruence relation $\sim_T$ between locally nameless $\lambda$-$Y$ terms and skeleton terms:

**Definition 6.7.** [$\sim_T$ relation]

In the following definition, $m, p, q$ range over simply-typed locally nameless $\lambda$-$Y$ terms and $s, t$ range over skeleton terms $T$:

$$(bvar)\ \frac{}{n \sim_T *} \qquad (fvar)\ \frac{}{x \sim_T *} \qquad (Y)\ \frac{}{Y_A \sim_T *}$$

$$(un)\ \frac{m \sim_T t}{\lambda_A.m \sim_T \circ t} \qquad (bin)\ \frac{p \sim_T s \qquad q \sim_T t}{pq \sim_T s \,\&\, t}$$

Having defined the $\sim_T$ relation, we could now augment our substitution lemma proof with a skeleton tree corresponding to the LN term $m$, performing (simultaneous) induction on the congruence relation $m \sim_T t$. For the $\lambda$-case, we have a skeleton tree of the form $\circ t$ (for $m \equiv \lambda_A.p$)\$. The inductive hypothesis call is now:

```
ih = subst-⊩ℓ-2 {A} {C} {(x' , τ , A) :: Γ} {τ'} {x}
  {Λ[ 0 >> fv x' ] m} {n} {t} (opn-~T-inv m~t) ...
```

where $t$ is the skeleton tree corresponding to $p$, and by Lemma 6.7 (opn-~T-inv), also to $p^{x'}$:

**Lemma 6.7.** $m \sim_T t \implies \{k \to x\}m \sim_T t$

*Proof.* By induction on the relation $m \sim_T t$. The only interesting case is $(bvar)$. We have two cases, when $n = k$ or $n \neq k$. In both cases, we have $n \sim_T *$, thus in case $n \neq k$, the result follows by assumption, otherwise we have $n\{k \to x\} \equiv x$ and thus $x \sim_T *$ by $(fvar)$.

□

After this modification, Agda (grudgingly) accepted the definition as terminating, even though this rather complicated inductive/recursive definition of the proof now takes a rather long time to compile (slowdown by up to a factor of 6 compared to other theories of similar length).

# References


Aydemir, Brian E., Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. "Mechanized Metatheory for the Masses: The Poplmark Challenge." In *Theorem Proving in Higher Order Logics: 18th International Conference, Tphols 2005, Oxford, Uk, August 22-25, 2005. Proceedings*, edited by Joe Hurd and Tom Melham, 50–65. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/11541868_4[4].

Aydemir, Brian, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. "Engineering Formal Metatheory." In *Proceedings of the 35th Annual Acm Sigplan-Sigact Symposium on Principles of Programming Languages*, 3–15. POPL '08. New York, NY, USA: ACM. doi:10.1145/1328438.1328443[5].

Bakel, Steffen van. 2003. "Semantics with Intersection Types." http://www.doc.ic.ac.uk/~svb/SemIntTypes/Notes.pdf.

Barendregt, Henk, Wil Dekkers, and Richard Statman. 2013. *Lambda Calculus with Types*. New York, NY, USA: Cambridge University Press.

Berghofer, Stefan, and Christian Urban. 2006. "A Head-to-Head Comparison of de Bruijn Indices and Names." In *IN Proc. Int. Workshop on Logical Frameworks and Metalanguages: THEORY and Practice*, 46–59.

Clairambault, Pierre, and Andrzej S. Murawski. 2013. "Böhm Trees as Higher-Order Recursive Schemes." In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*, 91–102. doi:10.4230/LIPIcs.FSTTCS.2013.91[6].

Harper, Robert, Furio Honsell, and Gordon Plotkin. 1993. "A Framework for Defining Logics." *J. ACM* 40 (1). New York, NY, USA: ACM: 143–84. doi:10.1145/138027.138060[7].

Kobayashi, Naoki. 2009. "Types and Higher-Order Recursion Schemes for Verification of Higher-Order Programs." In *Proceedings of the 36th Annual Acm Sigplan-Sigact Symposium on Principles of Programming Languages*, 416–28. POPL '09. New York, NY, USA: ACM. doi:10.1145/1480881.1480933[8].

———. 2013. "Model Checking Higher-Order Programs." *J. ACM* 60 (3). New York, NY, USA: ACM: 20:1–20:62. doi:10.1145/2487241.2487246[9].

Ong, C.-H. L. 2006. "On Model-Checking Trees Generated by Higher-Order Recursion Schemes." In *Proceedings of the 21st Annual Ieee Symposium on Logic in Computer Science*, 81–90. LICS '06. Washington, DC, USA: IEEE Computer Society. doi:10.1109/LICS.2006.38[10].

Pfenning, F., and C. Elliott. 1988. "Higher-Order Abstract Syntax." In *Proceedings of the Acm Sigplan 1988 Conference on Programming Language Design and Implementation*, 199–208. PLDI '88. New York, NY, USA: ACM.


[4] https://doi.org/10.1007/11541868_4

[5] https://doi.org/10.1145/1328438.1328443

[6] https://doi.org/10.4230/LIPIcs.FSTTCS.2013.91

[7] https://doi.org/10.1145/138027.138060

[8] https://doi.org/10.1145/1480881.1480933

[9] https://doi.org/10.1145/2487241.2487246

[10] https://doi.org/10.1109/LICS.2006.38

doi:10.1145/53990.54010[11].

Pfenning, Frank, and Carsten Schürmann. 1999. “Automated Deduction — Cade-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings.” In, 202–6. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/3-540-48660-7_14[12].

Pollack, Robert. 1995. “Polishing up the Tait-Martin-Löf Proof of the Church-Rosser Theorem.”

Ramsay, Steven J., Robin P. Neatherway, and C.-H. Luke Ong. 2014. “A Type-Directed Abstraction Refinement Approach to Higher-Order Model Checking.” *SIGPLAN Not.* 49 (1). New York, NY, USA: ACM: 61–72. doi:10.1145/2578855.2535873[13].

Takahashi, M. 1995. “Parallel Reductions in λ-Calculus.” *Information and Computation* 118 (1): 120–27. http://www.sciencedirect.com/science/article/pii/S0890540185710577.

Tsukada, Takeshi, and C.-H. Luke Ong. 2014. “Compositional Higher-Order Model Checking via $$-Regular Games over Böhm Trees.” In *Proceedings of the Joint Meeting of the Twenty-Third Eacsl Annual Conference on Computer Science Logic (Csl) and the Twenty-Ninth Annual Acm/Ieee Symposium on Logic in Computer Science (Lics)*, 78:1–78:10. CSL-Lics ’14. New York, NY, USA: ACM. doi:10.1145/2603088.2603133[14].

[11] https://doi.org/10.1145/53990.54010

[12] https://doi.org/10.1007/3-540-48660-7_14

[13] https://doi.org/10.1145/2578855.2535873

[14] https://doi.org/10.1145/2603088.2603133

# Appendix

## Nominal implementation in Isabelle

```
theory LamYNom
imports "Nominal2-Isabelle/Nominal/Nominal2" begin
```

### Definition of $\lambda$-$Y$ terms

```
atom_decl name

nominal_datatype type = O | Arr type type ("_ → _")

nominal_datatype trm =
  Var name
| App trm trm
| Lam x::name l::trm  binds x in l ("Lam [_]. _" [100, 100] 100)
| Y type
```

### Definition of substitution

```
nominal_function
  subst :: "trm ⇒ name ⇒ trm ⇒ trm"  ("_ [_ ::= _]" [90, 90, 90] 90)
where
  "(Var x)[y ::= s] = (if x = y then s else (Var x))"
| "(App t1 t2)[y ::= s] = App (t1[y ::= s]) (t2[y ::= s])"
| "atom x ♯ (y, s) ⟹ (Lam [x]. t)[y ::= s] = Lam [x].(t[y ::= s])"
| "(Y t)[y ::= s] = Y t"
⟨proof⟩
nominal_termination (eqvt)
⟨proof⟩

lemma forget:
  shows "atom x ♯ t ⟹ t[x ::= s] = t"
⟨proof⟩

lemma fresh_type:
  fixes n :: name and t :: type
  shows "atom n ♯ t"
⟨proof⟩
```

```
lemma fresh_fact:
  fixes z::"name"
  assumes a: "atom z ♯ s"
      and b: "z = y ∨ atom z ♯ t"
  shows "atom z ♯ t[y ::= s]"
⟨proof⟩
```

```
lemma substitution_lemma:
  assumes a: "x ≠ y" "atom x ♯ u"
  shows "t[x ::= s][y ::= u] = t[y ::= u][x ::= s[y ::= u]]"
⟨proof⟩
```

## $\beta Y$-reduction

```
inductive
  beta_Y :: "trm ⇒ trm ⇒ bool" (" _ ⇒ _" [80,80] 80)
where
  red_L[intro]: "⟦ M ⇒ M' ⟧ ⟹ App M N ⇒ App M' N"
| red_R[intro]: "⟦ N ⇒ N' ⟧ ⟹ App M N ⇒ App M N'"
| abs[intro]: "⟦ M ⇒ M' ⟧ ⟹ Lam [x]. M ⇒ Lam [x]. M'"
| beta[intro]: "⟦ atom x ♯ N ⟧ ⟹ App (Lam [x]. M) N ⇒ M[x ::= N]"

| Y[intro]: "App (Y σ) M ⇒ App M (App (Y σ) M)"

equivariance beta_Y
nominal_inductive beta_Y
  avoids beta: "x" | abs: "x"
⟨proof⟩
```

## Parallel $\beta Y$-reduction

```
inductive
  pbeta :: "trm ⇒ trm ⇒ bool" ("_ ≫ _" [80,80] 80)
where
  refl[intro]: "(Var x) ≫ (Var x)"
| reflY[intro]: "Y σ ≫ Y σ"
| app[intro]: "⟦ M ≫ M' ; N ≫ N' ⟧ ⟹ App M N ≫ App M' N'"
| abs[intro]: "⟦ M ≫ M' ⟧ ⟹ Lam [x]. M ≫ Lam [x]. M'"
| beta[intro]: "⟦ atom x ♯ N ; atom x ♯ N' ; M ≫ M' ; N ≫ N' ⟧ ⟹ App (Lam
[x]. M) N ≫ M'[x ::= N']"
| Y[intro]: "⟦ M ≫ M' ⟧ ⟹ App (Y σ) M ≫ App M' (App (Y σ) M')"
equivariance pbeta

nominal_inductive pbeta
 avoids beta: "x" | abs: "x"
⟨proof⟩
```

## Maximal parallel $\beta Y$-reduction

```
nominal_function
  not_abst :: "trm ⇒ bool"
where
  "not_abst (Var x) = True"
| "not_abst (App t1 t2) = True"
| "not_abst (Lam [x]. t) = False"
| "not_abst (Y t) = True"
⟨proof⟩
nominal_termination (eqvt) ⟨proof⟩

nominal_function
  not_Y :: "trm ⇒ bool"
where
  "not_Y (Var x) = True"
| "not_Y (App t1 t2) = True"
| "not_Y (Lam [x]. t) = True"
| "not_Y (Y t) = False"
⟨proof⟩
nominal_termination (eqvt) ⟨proof⟩

inductive
  pbeta_max :: "trm ⇒ trm ⇒ bool" ("_ >>> _" [80,80] 80)
where
  refl[intro]: "(Var x) >>> (Var x)"
| reflY[intro]: "Y σ >>> Y σ"
| app[intro]: "⟦ not_abst M ; not_Y M ;  M >>> M' ; N >>> N' ⟧ ⟹ App M N >>>
App M' N'"
| abs[intro]: "⟦ M >>> M' ⟧ ⟹ Lam [x]. M >>> Lam [x]. M'"
| beta[intro]: "⟦ atom x ♯ N ; atom x ♯ N' ; M >>> M' ; N >>> N' ⟧ ⟹ App (Lam
[x]. M) N >>> M'[x ::= N']"
| Y[intro]: "⟦ M >>> M' ⟧ ⟹ App (Y σ) M >>> App M' (App (Y σ) M')"
equivariance pbeta_max

nominal_inductive pbeta_max
  avoids beta: "x" | abs: "x"
⟨proof⟩

lemma not_Y_ex: "¬(not_Y M) ⟹ ∃σ. M = Y σ"
⟨proof⟩
```

## Lemma 2.1

```
lemma pbeta_max_ex:
  fixes M
  shows "∃M'. M >>> M'"
⟨proof⟩
```

## Lemma 2.2

```
lemma subst_rename:
  assumes a: "atom y ♯ t"
  shows "t[x ::= s] = ((y ↔ x) • t)[y ::= s]"
⟨proof⟩

lemma fresh_in_pbeta: "s ≫ s' ⟹ atom (x::name) ♯ s ⟹  atom x ♯ s'"
⟨proof⟩

lemma pbeta_lam_case_ex: "(Lam [x]. s) ≫ s' ⟹ ∃t. s' = Lam [x]. t ∧ s ≫ t"
⟨proof⟩

lemma pbeta_cases_2:
  shows "atom x ♯ t ⟹ App (Lam [x]. s) t ≫ a2 ⟹
    (⋀s' t'. a2 = App (Lam [x]. s') t' ⟹ atom x ♯ t' ⟹ s ≫ s' ⟹ t ≫ t'
⟹ P) ⟹
    (⋀t' s'. a2 = s' [x ::= t'] ⟹ atom x ♯ t ⟹ atom x ♯ t' ⟹ s ≫ s' ⟹
t ≫ t' ⟹ P) ⟹ P"
⟨proof⟩

lemma Lem2_5_1:
  assumes "s ≫ s'"
      and "t ≫ t'"
      shows "(s[x ::= t]) ≫ (s'[x ::= t'])"
⟨proof⟩

lemma pbeta_max_closes_pbeta:
  fixes a b d
  assumes "a >>> d"
  and "a ≫ b"
  shows "b ≫ d"
⟨proof⟩
```

## Proof of dp(≫)

```
lemma Lem2_5_2:
  assumes "a ≫ b"
      and "a ≫ c"
    shows "∃d. b ≫ d ∧ c ≫ d"
⟨proof⟩
```

## Reflexive-transitive closure of $\beta Y$

```
inductive close :: "(trm ⇒ trm ⇒ bool) ⇒ trm ⇒ trm ⇒ bool" ("_* _ _" [80,80]
80) for R::"trm ⇒ trm ⇒ bool"
where
  base[intro]: "R a b ⟹ R* a b"
| refl[intro]: "R* a a"
```

```
| trans[intro]: "⟦ R* a b ; R* b c ⟧ ⟹ R* a c"
```

## Proof of dp($\Rightarrow_Y^*$)

```
definition DP :: "(trm ⇒ trm ⇒ bool) ⇒ (trm ⇒ trm ⇒ bool) ⇒ bool" where
"DP R T = (∀a b c. R a b ∧ T a c ⟶ (∃d. T b d ∧ R c d))"

lemma DP_R_R_imp_DP_R_Rc_pbeta:
  assumes "DP pbeta pbeta"
    shows "DP pbeta (close pbeta)"
⟨proof⟩

lemma DP_R_R_imp_DP_Rc_Rc_pbeta:
  assumes "DP pbeta pbeta"
    shows "DP (close pbeta) (close pbeta)"
⟨proof⟩

lemma pbeta_refl[intro]: "s ≫ s"
⟨proof⟩

lemma M1': "M ⇒ M' ⟹ M ≫ M'"
⟨proof⟩

lemma M1: "beta_Y* M M' ⟹ pbeta* M M'"
⟨proof⟩

lemma red_r_close: "beta_Y* N N' ⟹ beta_Y* (App M N) (App M N')"
⟨proof⟩

lemma red_l_close: "beta_Y* M M' ⟹ beta_Y* (App M N) (App M' N)"
⟨proof⟩

lemma abs_close: "beta_Y* M M' ⟹ beta_Y* (Lam [x]. M) (Lam [x]. M')"
⟨proof⟩


lemma M2: "pbeta* M M' ⟹ beta_Y* M M'"
⟨proof⟩
```

## Simple-typing relation ⊢

```
inductive wf_ctxt :: "(name × type) list ⇒ bool"
where
    nil:  "wf_ctxt []"
  | cons:  "⟦ wf_ctxt Γ ; atom x ♯ Γ ⟧ ⟹ wf_ctxt ((x,σ)#Γ)"
equivariance wf_ctxt

inductive
  wt_terms :: "(name × type) list ⇒ trm ⇒ type ⇒ bool" ("_ ⊢ _ : _")
```

where

```
  var: "⟦ (x,σ) ∈ set Γ ; wf_ctxt Γ ⟧ ⟹ Γ ⊢ Var x : σ"
| app: "⟦ Γ ⊢ M : σ → τ ; Γ ⊢ N : σ ⟧ ⟹ Γ ⊢ App M N : τ"

| abs: "⟦ atom x ♯ Γ ; ((x,σ)#Γ) ⊢ M : τ ⟧ ⟹ Γ ⊢ Lam [x]. M : σ → τ"
| Y: "⟦ wf_ctxt Γ ⟧ ⟹  Γ ⊢ Y σ : (σ → σ) → σ"
```

equivariance `wt_terms`

nominal_inductive `wt_terms`
avoids `abs: "x"`
⟨proof⟩

## Subject reduction theorem for $\Rightarrow_Y$

lemma `wf_ctxt_cons: "wf_ctxt ((x, σ)#Γ) ⟹ wf_ctxt Γ ∧ atom x ♯ Γ"`
⟨proof⟩

lemma `wt_terms_impl_wf_ctxt: "Γ ⊢ M : σ ⟹ wf_ctxt Γ"`
⟨proof⟩

lemma `weakening:`
  fixes `Γ Γ' M σ`
  assumes `"Γ ⊢ M : σ"` and `"set Γ ⊆ set Γ'"`
  and `"wf_ctxt Γ'"`
  shows `"Γ' ⊢ M : σ"`
⟨proof⟩

lemma `wf_ctxt_exchange: "wf_ctxt ((x,σ) # (y,π) # Γ) ⟹ wf_ctxt ((y,π) # (x,σ) # Γ)"`
⟨proof⟩

lemma `exchange: "(x,σ) # (y,π) # Γ ⊢ M : δ ⟹ (y,π) # (x,σ) # Γ ⊢ M : δ"`
⟨proof⟩

lemma `wt_terms_cases_2:`
  shows `"Γ ⊢ Lam [x]. M : a3 ⟹ atom x ♯ Γ⟹ (⋀σ τ. a3 = σ → τ ⟹ ((x, σ)#Γ) ⊢ M : τ ⟹ P) ⟹ P"`
⟨proof⟩

lemma `subst_typ_aux: "(x, τ) # Γ ⊢ Var y : σ ⟹ x = y ⟹ τ = σ"`
⟨proof⟩

lemma `subst_typ:`
  assumes `"((x,τ)#Γ) ⊢ M : σ"` and `"Γ ⊢ N : τ"`
  shows `"Γ ⊢ M[x ::= N] : σ"`
⟨proof⟩

lemma `beta_Y_typ:`
  assumes `"Γ ⊢ M : σ"`

and *"M ⇒ M'"*
shows *"Γ ⊢ M' : σ"*
⟨proof⟩

lemma *beta_Y_c_typ:*
assumes *"Γ ⊢ M : σ"*
and *"beta_Y* M M'"*
shows *"Γ ⊢ M' : σ"*
⟨proof⟩

## Church Rosser Theorem

lemma *church_rosser_typ:*
assumes *"Γ ⊢ a : σ"*
and *"beta_Y* a b"*
and *"beta_Y* a c"*
shows *"∃d. beta_Y* b d ∧ beta_Y* c d ∧ Γ ⊢ d : σ"*
⟨proof⟩
end

# Locally Nameless implementation in Isabelle

theory *LamYNmless*
imports *Main* begin

## Definition of λ-Y pre-terms

typedecl *atom*

axiomatization where
*atom_inf: "infinite (UNIV :: atom set)"*

datatype *type = O | Arr type type ("_ → _")*

datatype *ptrm = FVar atom | BVar nat | App ptrm ptrm | Lam ptrm | Y type*

## Definition of the open operation

```
fun opn :: "nat ⇒ ptrm ⇒ ptrm ⇒ ptrm" ("{_ → _} _")  where
"{k → u} (FVar x) = FVar x" |
"{k → u} (BVar i) = (if i = k then u else BVar i)" |
"{k → u} (App t1 t2) = App ({k → u} t1)({k → u} t2)" |
"{k → u} (Lam t) = Lam ({(k+1) → u} t)" |
"{k → u} (Y σ) = Y σ"
```

```
definition opn':: "ptrm ⇒ ptrm ⇒ ptrm" ("_^_") where
"opn' t u ≡ {0 → u} t"
```

```
lemma bvar_0_open_any:"BVar 0^M = M"
⟨proof⟩
```

```
lemma bvar_Suc_n_open_any:"(BVar (Suc n))^M = BVar (Suc n)"
⟨proof⟩
```

## Definition of well formed terms

```
inductive trm :: "ptrm ⇒ bool" where
var: "trm (FVar x)" |
app: "⟦ trm t1 ; trm t2 ⟧ ⟹ trm (App t1 t2)" |
lam: "⟦ finite L ; (⋀x. x ∉ L ⟹ trm (t^(FVar x))) ⟧ ⟹ trm (Lam t)" |
Y: "trm (Y σ)"
thm trm.simps
```

```
lemma bvar_not_trm: "trm (BVar n) ⟹ False"
⟨proof⟩
```

```
lemma x_Ex: "⋀L:: atom set. finite L ⟹ ∃x. x ∉ L"
⟨proof⟩
```

## Definition of substitution

fun *FV :: "ptrm ⇒ atom set"* where
*"FV (FVar x) = {x}" |*
*"FV (BVar i) = {}" |*
*"FV (App t1 t2) = (FV t1) ∪ (FV t2)" |*
*"FV (Lam t) = FV t" |*
*"FV (Y σ) = {}"*

lemma *FV_finite: "finite (FV u)"*
⟨proof⟩

primrec *subst :: "ptrm ⇒ atom ⇒ ptrm ⇒ ptrm" ("_ [_ ::= _]" [90, 90, 90] 90)*
where
*"(FVar x)[z ::= u] = (if x = z then u else FVar x)" |*
*"(BVar x)[z ::= u] = BVar x" |*
*"(App t1 t2)[z ::= u] = App (t1[z ::= u]) (t2[z ::= u])" |*
*"(Lam t)[z ::= u] = Lam (t[z ::= u])" |*
*"(Y σ)[z ::= u] = (Y σ)"*

lemma *subst_fresh: "x ∉ FV t ⟹ t[x ::= u] = t"*
⟨proof⟩

lemma *subst_fresh2: "x ∉ FV t ⟹ t = t[x ::= u]"*
⟨proof⟩

lemma *opn_trm_core: "i ≠ j ⟹ {j → v} e = {i → u}({j → v} e) ⟹ e = {i → u} e"*
⟨proof⟩

lemma *opn_trm: "trm e ⟹ e = {k → t}e"*
⟨proof⟩

lemma *opn_trm2: "trm e ⟹ {k → t}e = e"*
⟨proof⟩

lemma *subst_open: "trm u ⟹ ({n → w}t)[x ::= u] = {n → w [x ::= u]} (t [x ::= u])"*
⟨proof⟩

lemma *subst_open2: "trm u ⟹ {n → w [x ::= u]} (t [x ::= u]) = ({n → w}t)[x ::= u]"*
⟨proof⟩

lemma *fvar_subst_simp: "x ≠ y ⟹ FVar y = FVar y[x ::= u]"*
⟨proof⟩

lemma *fvar_subst_simp2: "u = FVar x[x ::= u]"*
⟨proof⟩

lemma *subst_open_var: "trm u ⟹ x ≠ y ⟹ (t^FVar y)[x ::= u] = (t [x ::= u])^FVar y"*
⟨proof⟩

lemma *subst_open_var2: "trm u ⟹ x ≠ y ⟹ (t [x ::= u])^FVar y = (t^FVar y)[x ::= u]"*
⟨proof⟩

lemma *subst_intro: "trm u ⟹ x ∉ FV t ⟹ (t^FVar x)[x::=u] = t^u"*
⟨proof⟩

lemma *subst_intro2: "trm u ⟹ x ∉ FV t ⟹ t^u = (t^FVar x)[x::=u]"*
⟨proof⟩

lemma *subst_trm: "trm e ⟹ trm u ⟹ trm(e[x::=u])"*
⟨proof⟩

lemma *trm_opn: "trm (Lam M) ∧ trm N ⟹ trm M^N"*
⟨proof⟩

## Definition of the close operation

fun *cls :: "nat ⇒ atom ⇒ ptrm ⇒ ptrm" ("{_ <- _} _")* where
*"{k <- x} (FVar y) = (if x = y then BVar k else FVar y)" |*
*"{k <- x} (BVar i) = BVar i" |*
*"{k <- x} (App t1 t2) = App ({k <- x} t1)({k <- x} t2)" |*
*"{k <- x} (Lam t) = Lam ({(k+1) <- x} t)" |*
*"{k <- x} (Y σ) = Y σ"*

definition *cls':: "atom ⇒ ptrm ⇒ ptrm" ("_^_")* where
*"cls' x t ≡ {0 <- x} t"*

lemma *FV_simp: "⟦ x ∉ FV M ; x ≠ y ⟧ ⟹ x ∉ FV {k → FVar y} M"*
⟨proof⟩

lemma *FV_simp2: "x ∉ FV M ∪ FV N ⟹ x ∉ FV {k → N}M"*
⟨proof⟩

lemma *FV_simp3: "x ∉ FV {k → N}M ⟹ x ∉ FV M"*
⟨proof⟩

lemma *FV_simp4: "x ∉ FV M ⟹ x ∉ FV {k <- y} M"*
⟨proof⟩

lemma *FV_simp5: "x ∉ FV M ∪ FV N ⟹ x ∉ FV (M[y ::= N])"*
⟨proof⟩

lemma *fv_opn_cls_id: "x ∉ FV t ⟹ {k <- x}{k → FVar x}t = t"*
⟨proof⟩

lemma *fv_opn_cls_id2: "x ∉ FV t ⟹ t = {k <- x}{k → FVar x}t"*
⟨proof⟩

lemma *opn_cls_swap: "k ≠ m ⟹ x ≠ y ⟹ {k <- x}{m → FVar y}M = {m → FVar y}{k <- x}M"*
⟨proof⟩

lemma *opn_cls_swap2: "k ≠ m ⟹ x ≠ y ⟹ {m → FVar y}{k <- x}M = {k <- x}{m → FVar y}M"*
⟨proof⟩

lemma *opn_opn_swap: "k ≠ m ⟹ x ≠ y ⟹ {k → FVar x}{m → FVar y}M = {m → FVar y}{k → FVar x}M"*
⟨proof⟩

lemma *cls_opn_eq_subst: "trm M ⟹ ({k → FVar y} {k <- x} M) = (M[x ::= FVar y])"*
⟨proof⟩

lemma *cls_opn_eq_subst2: "trm M ⟹ (M[x ::= FVar y]) =({k → FVar y} {k <- x} M)"*
⟨proof⟩

## $\beta Y$-reduction

inductive *beta_Y :: "ptrm ⇒ ptrm ⇒ bool" (*infix *"⇒" 300)*
where
*red_L[intro]: "⟦ trm N ; M ⇒ M' ⟧ ⟹ App M N ⇒ App M' N"*
*| red_R[intro]: "⟦ trm M ; N ⇒ N' ⟧ ⟹ App M N ⇒ App M N'"*
*| abs[intro]: "⟦ finite L ; (⋀x. x ∉ L ⟹ M^(FVar x) ⇒ M'^(FVar x)) ⟧ ⟹ Lam M ⇒ Lam M'"*
*| beta[intro]: "⟦ trm (Lam M) ; trm N ⟧ ⟹ App (Lam M) N ⇒ M^N"*
*| Y[intro]: "trm M ⟹ App (Y σ) M ⇒ App M (App (Y σ) M)"*

lemma *trm_beta_Y_simp1: "M ⇒ M' ⟹ trm M ∧ trm M'"*
⟨proof⟩

## Parallel $\beta Y$-reduction

inductive
*pbeta :: "ptrm ⇒ ptrm ⇒ bool" ("_ ≫ _" [80,80] 80)*
where
*refl[intro]: "(FVar x) ≫ (FVar x)"*
*| reflY[intro]: "Y σ ≫ Y σ"*
*| app[intro]: "⟦ M ≫ M' ; N ≫ N' ⟧ ⟹ App M N ≫ App M' N'"*
*| abs[intro]: "⟦ finite L ; (⋀x. x ∉ L ⟹ M^(FVar x) ≫ M'^(FVar x)) ⟧ ⟹ Lam M ≫ Lam M'"*
*| beta[intro]: "⟦ finite L ; (⋀x. x ∉ L ⟹ M^(FVar x) ≫ M'^(FVar x)) ; N ≫*

```
N' ⟧ ⟹ App (Lam M) N ≫ M'^N'"
| Y[intro]: "⟦ M ≫ M' ⟧ ⟹ App (Y σ) M ≫ App M' (App (Y σ) M')"
```

lemma *trm_pbeta_simp1: "M ≫ M' ⟹ trm M ∧ trm M'"*
⟨proof⟩

## Maximal parallel $\beta Y$-reduction

```
fun not_abst :: "ptrm ⇒ bool"
where
  "not_abst (FVar x) = True"
| "not_abst (BVar x) = True"
| "not_abst (App t1 t2) = True"
| "not_abst (Lam t) = False"
| "not_abst (Y t) = True"
```

```
fun not_Y :: "ptrm ⇒ bool"
where
  "not_Y (FVar x) = True"
| "not_Y (BVar x) = True"
| "not_Y (App t1 t2) = True"
| "not_Y (Lam t) = True"
| "not_Y (Y t) = False"
```

```
inductive
  pbeta_max :: "ptrm ⇒ ptrm ⇒ bool" ("_ >>> _" [80,80] 80)
where
  refl[intro]: "(FVar x) >>> (FVar x)"
| reflY[intro]: "Y σ >>> Y σ"
| app[intro]: "⟦ not_abst M ; not_Y M ;  M >>> M' ; N >>> N' ⟧ ⟹ App M N >>>
App M' N'"
| abs[intro]: "⟦ finite L ; (⋀x. x ∉ L ⟹ M^(FVar x) >>> M'^(FVar x)) ⟧ ⟹
Lam M >>>  Lam M'"
| beta[intro]: "⟦ finite L ; (⋀x. x ∉ L ⟹ M^(FVar x) >>> M'^(FVar x)) ; N >>>
N' ⟧ ⟹ App (Lam M) N >>> M'^N'"
| Y[intro]: "⟦ M >>> M' ⟧ ⟹ App (Y σ) M >>> App M' (App (Y σ) M')"
```

lemma *trm_pbeta_max_simp1: "M >>> M' ⟹ trm M ∧ trm M'"*
⟨proof⟩

lemma *pbeta_beta': "finite L ⟹ (⋀x. x ∉ L ⟹ M^(FVar x) ≫ M'^(FVar x)) ⟹ N ≫ N' ⟹ App (Lam M) N ≫ {0 → N'} M'"*
⟨proof⟩

lemma *not_Y_ex: "¬(not_Y M) ⟹ ∃σ. M = Y σ"*
⟨proof⟩

lemma *not_abst_simp: "not_abst M ⟹ not_abst {k → FVar y} {k <- x} M"*
⟨proof⟩

```
lemma not_Y_simp: "not_Y M ⟹ not_Y {k → FVar y} {k <- x} M"
⟨proof⟩
```

## Lemma 2.1

```
lemma pbeta_max_beta': "finite L ⟹ (⋀x. x ∉ L ⟹ M^(FVar x) >>> M'^(FVar x))
⟹ N >>> N' ⟹ App (Lam M) N >>> {0 → N'} M'" ⟨proof⟩
```

```
lemma Lem2_5_1_beta_max:
  assumes "s >>> s'"
      shows "(s[x ::= FVar y]) >>> (s'[x ::= FVar y])"
⟨proof⟩
```

```
lemma pbeta_max_cls: "t >>> d ⟹ y ∉ FV t ∪ FV d ∪ {x} ⟹ {k → FVar y}{k <-
x}t >>> {k → FVar y}{k <- x}d"
⟨proof⟩
```

```
lemma pbeta_max_ex:
  fixes M assumes "trm M"
  shows "∃M'. M >>> M'"
⟨proof⟩
```

## Lemma 2.2

```
lemma Lem2_5_1:
  assumes "s ≫ s'"
      and "t ≫ t'"
      shows "(s[x ::= t]) ≫ (s'[x ::= t'])"
⟨proof⟩
```

```
lemma Lem2_5_1opn:
  assumes "⋀x. x ∉ L ⟹ s^FVar x ≫ s'^FVar x" and "finite L"
      and "t ≫ t'"
      shows "s^t ≫ s'^t'"
⟨proof⟩
```

```
lemma pbeta_max_closes_pbeta:
  fixes a b d
  assumes "a >>> d"
  and "a ≫ b"
  shows "b ≫ d"
⟨proof⟩
```

## Proof of dp(≫)

```
lemma Lem2_5_2:
  assumes "a ≫ b"
      and "a ≫ c"
```

```
    shows "∃d. b ≫ d ∧ c ≫ d"
⟨proof⟩
```

## Reflexive-transitive closure of $\beta Y$

```
inductive close :: "(ptrm ⇒ ptrm ⇒ bool) ⇒ ptrm ⇒ ptrm ⇒ bool" ("_* _  _"
[80,80] 80) for R::"ptrm ⇒ ptrm ⇒ bool"
where
  base[intro]: "R a b ⟹ R* a b"
| refl[intro]: "R* a a"
| trans[intro]: "⟦ R* a b ; R* b c ⟧ ⟹ R* a c"

definition DP :: "(ptrm ⇒ ptrm ⇒ bool) ⇒ (ptrm ⇒ ptrm ⇒ bool) ⇒ bool" where
"DP R T = (∀a b c. R a b ∧ T a c ⟶ (∃d. T b d ∧ R c d))"

lemma DP_R_R_imp_DP_R_Rc_pbeta:
  assumes "DP pbeta pbeta"
    shows "DP pbeta (close pbeta)"
⟨proof⟩

lemma DP_R_R_imp_DP_Rc_Rc_pbeta:
  assumes "DP pbeta pbeta"
    shows "DP (close pbeta) (close pbeta)"
⟨proof⟩

lemma pbeta_refl[intro]: "trm s ⟹ s ≫ s"
⟨proof⟩

lemma M1': "M ⇒ M' ⟹ M ≫ M'"
⟨proof⟩

lemma M1: "beta_Y* M M' ⟹ pbeta* M M'"
⟨proof⟩

lemma red_r_close: "beta_Y* N N' ⟹ trm M ⟹ beta_Y* (App M N) (App M N')"
⟨proof⟩

lemma red_l_close: "beta_Y* M M' ⟹ trm N ⟹ beta_Y* (App M N) (App M' N)"
⟨proof⟩

lemma beta_Y_beta': "trm (Lam M) ⟹ trm N ⟹ App (Lam M) N ⇒ {0 → N} M"
⟨proof⟩

lemma Lem2_5_1'_beta_Y:
  assumes "s ⇒ s'"
      shows "(s[x ::= FVar y]) ⇒ (s'[x ::= FVar y])"
⟨proof⟩

lemma beta_Y_lam_cls: "a ⇒ b ⟹ Lam {0 <- x} a ⇒ Lam {0 <- x} b"
```

⟨proof⟩

lemma *abs_close: "⟦ ⋀x. x ∉ L ⟹ beta_Y* (M^FVar x) (M'^FVar x) ; finite L ⟧ ⟹ beta_Y* (Lam M) (Lam M')"*
⟨proof⟩

lemma *M2: "pbeta* M M' ⟹ beta_Y* M M'"*
⟨proof⟩

## Simple-typing relation ⊢

type_synonym *ctxt = "(atom × type) list"*

inductive *wf_ctxt :: "ctxt ⇒ bool"* where
*nil: "wf_ctxt []" |*
*cons: "⟦ x ∉ fst`set C ; wf_ctxt C ⟧ ⟹ wf_ctxt ((x, σ)#C)"*

inductive *wt_trm :: "ctxt ⇒ ptrm ⇒ type ⇒ bool" ("_ ⊢ _ : _")* where
*var: "⟦ wf_ctxt Γ ; (x,σ) ∈ set Γ ⟧ ⟹ Γ ⊢ FVar x : σ" |*
*app: "⟦ Γ ⊢ t1 : τ → σ ; Γ ⊢ t2 : τ ⟧ ⟹ Γ ⊢ App t1 t2 : σ" |*
*abs: "⟦ finite L ; (⋀x. x ∉ L ⟹ ((x,σ)#Γ) ⊢ (t^(FVar x)) : τ) ⟧ ⟹ Γ ⊢ Lam t : σ → τ" |*
*Y: "⟦ wf_ctxt Γ ⟧ ⟹ Γ ⊢ Y σ : (σ → σ) → σ"*

lemma *wf_ctxt_cons: "wf_ctxt ((x, σ)#Γ) ⟹ wf_ctxt Γ ∧ x ∉ fst`set Γ"*
⟨proof⟩

lemma *wt_terms_impl_wf_ctxt: "Γ ⊢ M : σ ⟹ wf_ctxt Γ"*
⟨proof⟩

lemma *opn_typ_aux: "(x, τ) # Γ ⊢ FVar y : σ ⟹ x = y ⟹ τ = σ"*
⟨proof⟩

lemma *weakening:*
  fixes *Γ Γ' M σ*
  assumes *"Γ ⊢ M : σ"* and *"set Γ ⊆ set Γ'"*
  and *"wf_ctxt Γ'"*
  shows *"Γ' ⊢ M : σ"*
⟨proof⟩

lemma *wf_ctxt_exchange: "wf_ctxt ((x,σ) # (y,π) # Γ) ⟹ wf_ctxt ((y,π) # (x,σ) # Γ)"*
⟨proof⟩

lemma *exchange: "(x,σ) # (y,π) # Γ ⊢ M : δ ⟹ (y,π) # (x,σ) # Γ ⊢ M : δ"*
⟨proof⟩

lemma *trm_wt_trm: "Γ ⊢ M : σ ⟹ trm M"*
⟨proof⟩

```
lemma subst_typ:
  assumes "trm M" and "((x,τ)#Γ) ⊢ M : σ" and "Γ ⊢ N : τ"
  shows "Γ ⊢ M[x ::= N] : σ"
⟨proof⟩

lemma opn_typ:
  fixes L
  assumes "finite L" "⋀x. x ∉ L ⟹ ((x,τ)#Γ) ⊢ M^FVar x : σ" and "Γ ⊢ N : τ"
  shows "Γ ⊢ M^N : σ"
⟨proof⟩

lemma beta_Y_typ:
  assumes "Γ ⊢ M : σ"
  and "M ⇒ M'"
  shows "Γ ⊢ M' : σ"
⟨proof⟩

lemma beta_Y_c_typ:
  assumes "Γ ⊢ M : σ"
  and "beta_Y* M M'"
  shows "Γ ⊢ M' : σ"
⟨proof⟩
```

## Church Rosser Theorem

```
lemma church_rosser_typ:
  assumes "Γ ⊢ a : σ"
      and "beta_Y* a b"
      and "beta_Y* a c"
    shows "∃d. beta_Y* b d ∧ beta_Y* c d ∧ Γ ⊢ d : σ"
⟨proof⟩
end
```