A formalization of the λ-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

This is a dedication

Acknowledgements

Say thanks to whoever listened to your rants for 2 months

Statement of Originality

This is the statement of originality

Abstract

This is the abstract. For this and the other front-matter options you can either include the text directly on the metadata file or you can use in order to include your text.

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/λ 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 λ-Y calculus with the intersection-type system described by **??which one...should I omit??**, 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 λ-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 λ-Y calculus, as there is little infomration available concerning the merits and disadvantages of different implementation approaches of λ-Y or indeed just the (simply typed) λ-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 λ-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 λ-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.

1.3 Main Achievements

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

- Formalization of the simply typed λ-Y calculus and proofs of confluence in Isabelle, using both Nominal sets and locally nameless encoding of binders.
- Formalization of the simply typed λ-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 λ-Y calculus and proof of subject invariance for intersection-types

2. Background

2.1 Binders

When describing the (untyped) λ-calculus on paper, the terms of the λ-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 λ-terms by structural recursion and prove properties about the λ-terms using structural induction (recursion and induction being two sides of the same coin).
However, whilst the definition above describes valid terms of the λ-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 λ-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 λ. Without the notion of α-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 α-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 α-equivalent λ-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 λ-term. Clearly, such function is well formed for "raw" terms, but does not work (or even make sense) for α-equivalent terms.
Conversely, there are informal definitions over α-equivalent terms, which are not straight-forward to define over raw terms. Take the usual definition of substitution, defined over α-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 λ case, it is assumed that a given λ-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 α-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 α-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 λ-calculus is rarely used in a fully formal setting.

To mitigate the overheads of a fully formal definition of the λ-calculus, we want to have an encoding of the λ-terms, which includes the notion of α-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 λ-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 α-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 λ-terms with a built in notion of α-equivalence, one can instead use nominal sets (**described in the section on nominal sets/Isabelle?**). The nominal package in Isabelle provides tools to automatically define terms with binders, which generate inductive definitions of α-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 α-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 α-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 λ-term. For bound variables, the index indicates which λ it refers to, by encoding the number of λ-binders that are in the scope between the index and the λ-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 λ it corresponds to, and 1 corresponds to x, as there is one λ-binder in scope. To encode free variables, one simply choses an index greater than the number of λ's currently in scope, for example, $\lambda\ 4$.

To see that this representation of λ-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 α-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, $x, y, z, \ldots$ to indexed variables $x_1, x_2, x_3, \ldots$. Then, the mapping from named terms to de Bruijn term is given by f, which we define in terms of an auxiliary function e:

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

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:

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

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

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 λ-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 λ-term, and the notion of α-equivalence is no longer relevant. We thus get around our problem of having an inductive principle and α-equivalent terms, by having a representation of λ-terms where every α-equivalence class of λ-terms has a single representative term in the de Bruijn notation.

In their comparison between named vs. nameless/de Bruijn representations of λ-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 λ-terms, since a `pterm` like `(BVar 1)` does not represent a λ-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 λ-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 λ-calculus.
Using HOAS for encoding the λ-calculus comes down to encoding binders using the meta-language binders. This way, the definitions of capture avoiding substitution or notion of α-equivalence are offloaded onto the meta-language. As an example, take the following definition of terms of the λ-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 λ-term as a Haskell function `(Term -> Term)`, relying on Haskell's internal substitution and notion of

α-equivalence. As with the de Bruijn and locally nameless representations, this encoding gives us inductively defined terms with a built in notion of α-equivalence.
However, using HOAS only works if the notion of α-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 λ-term is a triple (Γ, M, σ) s.t. $\Gamma \vdash M : \sigma$, where Γ is the typing context, M is a term of the untyped λ-calculus and σ 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 λ-term, we construct the following typing tree:

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

In the untyped λ-calculus, simple types and λ-terms are completely separate, brought together only through the typing relation $\vdash$ in the case of simple types *a la Curry*. The definition of λ-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 λ-Y calculus, we tend to conflate the "untyped" λ-Y terms, which are just the terms defined in Definition 2.2, with the "typed" λ-Y terms, which are simply-typed terms *a la Curry* of the form $\Gamma \vdash M : \sigma$, where M is an "untyped" λ-Y term. Thus, results about the λ-Y calculus in this work are in fact results about the "typed" λ-Y calculus.
However, the proofs of the Church Rosser theorem, as presented in the next section, use the untyped definition of β-reduction. Whilst it is possible to define a typed version of β-reduction, it turned out to be much easier to first prove the Church Rosser theorem for the so called "untyped" λ-Y calculus and the additionally restrict this result to only well-types λ-Y terms (see Section 4.2.1 for more details). Thus, the definition of the Church Rosser Theorem, formulated for the λ-Y calculus, is the following one:

Theorem 2.1. *[Church Rosser]*

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

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

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

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

2.3 λ-Y calculus

Originally, the field of higher order model checking mainly involved studying higher order recursion schemes (HORS), but more recently, exploring the λ-Y calculus, which is an extension of the simply typed λ-calculus, in the context of HOMC has gained traction (Clairambault and Murawski (2013)). We therefore present the λ-Y calculus, along with the proofs of the Church Rosser theorem and the formalization of intersection types for the λ-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 λ-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 λ-Y terms and the simple types are given below:

Definition 2.2. [λ-Y types and terms]

The set of simple types σ is built up inductively form the o constant and the arrow type $\rightarrow$. Let Var be a countably infinite set of atoms in the definition of the set of λ-terms M:

$$\sigma ::= \mathsf{o} \mid \sigma \rightarrow \sigma$$
$$M ::= x \mid MM \mid \lambda x.M \mid Y_\sigma \text{ where } x \in Var$$

The λ-Y calculus differs from the simply typed λ-calculus only in the addition of the Y constant family, indexed at every simple type σ, where the (simple) type of a Y_A constant (indexed with the type A) is $(A \rightarrow A) \rightarrow A$. The usual definition of β-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 λ-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 σ. We can therefore perform the following (well-typed) reduction in the λ-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_σ'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 β-reduction of the λ-calculus is confluent, that is, the reflexive-transitive closure of the β-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
>
> $$\forall a, b, c.\ aRb \land aRc \implies \exists d.\ bRd \land cRd$$

The proof of confluence of $\Rightarrow_Y$, the βY-reduction defined as the standard β-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* β-reduction ($\gg$), which, unlike the "plain" β-reduction satisfies the *diamond property* (note that we are talking about the "single step" β-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 β-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 $Q \gg M'$:

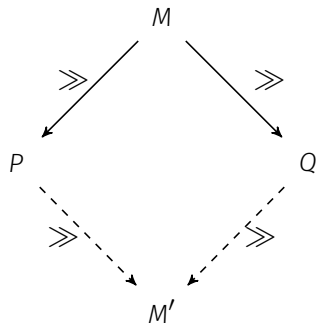

Figure 2.1: The diamond property of $\gg$, visualized

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* β-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}:

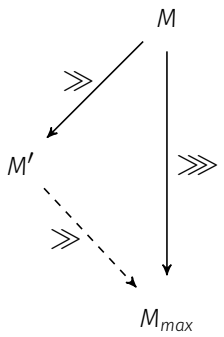

Figure 2.2: The proof of dp$(\gg)$ is split into two instances of this triangle

2.3.2.2 Parallel βY-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* βY*-reduction* $\gg$ formulated for the terms of the λ-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 β-reduction and *parallel* βY-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 βY-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:

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

When we try to construct a similar tree for β-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 (for the rules of normal β-reduction see

Definition 4.1).

Having described the intuition behind the *parallel* β-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 λ 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 λ-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. $dp(\gg)$

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 $\lambda^{BCD}_{\cap}$, extend simple types by adding a conjunction to the definition of types:

Definition 2.6. [$\lambda^{BCD}_{\cap}$ 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 $\lambda^{BCD}_{\cap}$ 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, φ is a constant (analogous to the constant o, introduced for the simple types in Definition 2.2).

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

The following conventions for intersection types are adopted throughout this section; ω 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.:

$$\forall\, i \in \underline{n}.\ \tau_i \subseteq \bigcap_{\underline{n}} \tau_i$$
$$\forall\, i \in \underline{n}.\ \tau_i \subseteq \tau \implies \bigcap_{\underline{n}} \tau_i \subseteq \tau$$
$$\rho \subseteq \psi \wedge \tau \subseteq \mu \implies \psi \rightsquigarrow \tau \subseteq \rho \rightsquigarrow \mu$$

(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, λ-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 Γ, M, τ, written as

$\Gamma \vDash M : \tau$, where Γ is the intersection-type context, similar in construction to the simple typing context, M is a λ-Y term and τ 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]

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

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

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

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 Γ is *well-formed*:

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

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

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

3. Methodology

3.1 Comparison of formalizations

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 λ-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 λ-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 λ-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 λ-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.

3.1.1 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

To this, we add another criterion:

- Proof automation

This project focuses mainly on the three criteria of mechanization overheads, technology transparency and automation, since the focus of our comparison is to chose the best mechanization and theorem prover to use for implementing intersection types for the λ-Y calculus (and proving subject invariance). These criteria are described in greater detail below:

3.1.1.1 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. Implementation overheads are usually things deemed too trivial to consider in a paper proof and a good mechanization will leverage automation and other language features to hide as much of these overheads as possible. The best example of a mechanization overhead in this project is the formalization of binders, discussed in Section 2.1 of the previous chapter.

Example 3.1. Another classic example of mechanization overheads, imposed by a fully formal setting of a theorem prover is something as trivial as the proof of commutativity of addition of natural numbers. In order to prove this property formally in Agda, we first have to define what is a natural number, using peano numbers where we have a `Z` zero constructor and a successor function `S`:

```
data ℕ : Set where
  Z : ℕ
  S : ℕ -> ℕ
```

Then we can define the addition operation for natural numbers:

```
_+_ : ℕ -> ℕ -> ℕ
Z + x = x
S x + y = S (x + y)
```

Finally, the proof of commutativity of addition may end up looking something like this:

```
S-inj : ∀ {x y} -> x ≡ y -> S x ≡ S y
S-inj refl = refl

+-comm' : ∀ x -> x + Z ≡ x
+-comm' Z = refl
+-comm' (S x) = S-inj (+-comm' x)

+-comm'' : ∀ x y -> x + S y ≡ S (x + y)
+-comm'' Z y = refl
+-comm'' (S x) y = S-inj (+-comm'' x y)

+-comm : ∀ x y -> (x + y) ≡ (y + x)
+-comm Z y rewrite +-comm' y = refl
+-comm (S x) y rewrite +-comm'' y x = S-inj (+-comm x y)
```

As we see here, this proof is quite long for something seemingly so trivial. Proofs of this type are usually unavoidable in a theorem prover, especially when we want to use such lemmas in a more interesting result we wan to formalize. Having low implementation overheads thus usually depends on the automation of the tool, wherein the tool itself is able to prove these properties automatically. What is more likely, however, is that a "good" tool will include something such a base theory (of natural numbers) in its library, so that he 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 what happened when we used the nominal library, where the theory was conveniently hidden away and managed for us by Isabelle's automatic provers.

For the scope of this project, binders are discussed and used for comparison often, since they are the "weak spot" where mechanization overheads are most apparent.
In this project, we decided to use nominal sets and locally nameless representation for binders, due to several reasons. The choice of nominal sets was tied to the implementation language, namely Isabelle, which has a well developed nominal sets library[1], maintained by Christian Urban. The appeal of using nominal sets is of course the touted minimal overheads in comparison to the informal presentation.
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 (i.e. theorems about locally nameless presentation are much closer in formulation to the informal presentation than theorems formulated for de Bruijn indices).
Both of these choices were guided in part by the initial choice of implementation language, Isabelle, which had good support both mechanizations. Isabelle was also chosen due to previous experience in mechanizing similar proofs.
The comparison between nominal and locally nameless versions of the λ-Y calculus, presented in Chapter 4, tries to highlight the differences in the two approaches in contrast to the usual informal reasoning.

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

3.1.1.2 Technology transparency

Technology transparency, as discussed here, is usually concerned with the presentation of the theory inside a proof assistant, such as Isabelle or Agda. As the snippets from Example 3.1 demonstrate, Agda includes features, such as Unicode support for mathematical symbols like $\mathbb{N}$, to make the implementation look more "natural". Other features, like Isabelle's proof representation language Isar (discussed in Section 5.2), are built into theorem provers to try to make the proofs and definitions look as close to conventional notation as possible.

Example 3.2. 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[2]:

> **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.

This criterion of technology transparency is discussed mainly in Chapter 5, which deals with the

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

comparison of Isabelle and Agda. The choice of the two theorem provers, but especially of Isabelle, was largely subjective. Having had previous experience with Isabelle, it was natural to use it initially, to lower the cost of entry. Initially only using Isabelle for both formalizations of binders also allowed for a more uniform comparison of the mechanization overheads.
The choice of Agda as the second implementation language was motivated by Agda having a dependent-type system. As a result, the style of proofs in Agda seems quite different to Isabelle, since the distinction between proofs and programs is largely erased. Agda was chosen over Coq, which is also a dependently-typed language, because it is more "bare-bones" and thus seemed more accessible to a novice in dependently-typed languages. Agda also has a higher "cool"-factor than Coq, being a newer language.

3.1.1.3 Proof automation

Proof automation ties into both the mechanization overheads and transparency aspects of a formalization, since high degree of automation can often result in a more natural/transparent looking proof where the "menial" reasoning steps are taken care of by the theorem prover, and the user only sees the higher-level reasoning of informal proofs.

maybe link to Lemma 5.1

Both following chapters discuss the automation features of Isabelle and Agda and try to draw comparisons by analyzing the same/equivalent lemmas in different mechanizations and theorem provers, in terms of automation. Whilst on paper, Isabelle includes a lot more automation, in the form of several tactics and automated theorem provers, whereas Agda comes with only very simple proof search tactics, Agda's more sophisticated type-system takes on and replicates at least some of the automation seen in Isabelle.

4. Nominal vs. Locally nameless

This chapter looks at the two different mechanizations of the λ-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.

4.1 Capture-avoiding substitution and β-reduction

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

4.1.1 Nominal sets representation

As was shown already, nominal set representation of terms is largely identical with the informal definitions, which is the main reason why this representation was chosen. This section will examine the implementation of λ-Y calculus in Isabelle, using the Nominal package.
We start, by examining the definition of untyped β-reduction, defined for the λ-Y calculus (referred to as βY-reduction due to the addition of the (Y) reduction rule):

Definition 4.1. [βY-reduction]

$$(red_L)\ \frac{M \Rightarrow_Y M'}{MN \Rightarrow_Y M'N} \qquad (red_R)\ \frac{N \Rightarrow_Y N'}{MN \Rightarrow_Y M'N}$$

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

This definition, with the exception of the added (Y) rule is the standard definition of the untyped β-reduction found in literature (**link?**). The $\sharp$ symbol is used to denote the *freshness* relation in nominal set theory. The side-condition $x \sharp N$ in the (β) rule can be read as "x is fresh in N", namely, the atom x does not appear in N. For a λ-term M, we have $x \sharp M$ iff $x \notin \mathbf{FV}(M)$, where we take the usual definition of **FV**:

Definition 4.2. The inductively defined FV is the set of *free variables* of a λ-term M.

$$\begin{aligned} \mathsf{FV}(x) &= \{x\} \\ \mathsf{FV}(MN) &= \mathsf{FV}(M) \cup \mathsf{FV}(N) \\ \mathsf{FV}(\lambda x.M) &= \mathsf{FV}(M) \setminus \{x\} \\ \mathsf{FV}(Y_\sigma) &= \varnothing \end{aligned}$$

The definition of substitution, used in the (β) rule is also unchanged with regards to the usual definition (except for the addition of the Y case, which is trivial):

Definition 4.3. [Capture-avoiding substitution]

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

4.1.1.1 Nominal Isabelle implementation

Whilst on paper, all these definitions are unchanged from the usual presentation, there are a few caveats when it comes to actually implementing these definitions in Isabelle, using the Nominal package. The declaration of the terms and types 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 **?equated up to α-equivalence?**, where a term $\lambda x.x$ and $\lambda y.y$ are considered equivalent, because both x and y are bound in the two respective terms, and can both be α-converted to the same term, for example $\lambda z.z$. In fact, proving such a lemma in 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.)

Next, we 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"
```

Whilst using **nominal_datatype** is automatic and requires no user input, the declaration of a function in Nominal Isabelle is less straightforward. Unlike using the usual "**fun**" declaration of a recursive function in Isabelle, where the theorem prover can automatically prove properties like termination or pattern exhaustiveness, there are several goals (13 in the case of the `subst` definition) which the user has to manually prove for any function using nominal data types, such as the λ-Y terms. 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)
```

do i need to explain what this property is? or is it ok for illustrative purposes?

Whilst most of the goals were trivial, proving cases involving λ-terms involved a substantial understanding 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. The proof script for the definition of substitution was actually lifted/copied? from the sample document, found in the Nominal package documentation, which had a definition of substitution for the untyped λ-calculus similar enough to be adaptable for the λ-Y calculus. Whilst this formalization required only a handful of other recursive function definitions, most of which could be copied from the sample document, 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.

4.1.2 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:

4.1.2.1 Pre-terms

Definition 4.4. [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 λ-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 4.4, is a superset of λ-Y terms and includes terms which are not well formed λ-Y terms.

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

Since we don't want to work with terms that do not correspond to λ-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 λ-Y terms (i.e. this **?inductive definition?** ensures that there are no "out of bounds" indices in a given pre-term):

Definition 4.5. [Well-formed terms]

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

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

Already, we see that this formalization introduces some overheads with respect to the informal/nominal encoding of the λ-Y calculus.
The upside of this definition of λ-Y terms becomes apparent when we start thinking about α-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 α-equivalence classes of named λ-Y terms collapse into a singleton α-equivalence class in a LN representation, the notion of α-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 4.6. [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:

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

ii) Substitution:

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

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

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:

$$\dfrac{\dfrac{\dfrac{\dfrac{(fvar)\ \dfrac{}{\mathbf{term}(y)} \qquad (app)\ \dfrac{(fvar)\ \dfrac{}{\mathbf{term}(z)} \qquad (fvar)\ \dfrac{}{\mathbf{term}(x)}}{\mathbf{term}(z\,x)}}{\mathbf{term}((0(z\,x))^y)}\,(lam)}{\mathbf{term}((\lambda 0(z\,1))^x)}\,(lam)}{\mathbf{term}(\lambda\lambda 0(z\,1))}$$

We assumed that $x \not\equiv y \not\equiv z$ in the proof tree above and thus omitted the $x \notin \mathbf{FV}\ldots$ 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 ($\mathbf{term}(2)$):

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

4.1.2.2 β-reduction for LN terms

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

Definition 4.7. [β-reduction (LN)]

$$(red_L)\ \frac{M \Rightarrow_Y M' \qquad \mathbf{term}(N)}{MN \Rightarrow_Y M'N} \qquad (red_R)\ \frac{\mathbf{term}(M) \qquad N \Rightarrow_Y N'}{MN \Rightarrow_Y MN'}$$

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

$$(Y)\ \frac{\mathbf{term}(M)}{Y_\sigma M \Rightarrow_Y M(Y_\sigma M)}$$

As expected, the *open* operation is now used instead of substitution in the (β) 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 λ-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 λ) with x.
While this definition is equivalent to Definition 4.1, 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 picking fresh/free variables) using *cofinite quantification*:

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

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

4.1.2.3 Implementation details

Unlike using the nominal package, the implementation of all the definitions and functions listed for the LN representation is very straightforward. To demonstrate this, we present the definition of the β-reduction in the LN mechanization:

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

4.2 Untyped Church Rosser Theorem

Having described the implementations of the two binder representations along with some basic definitions, such as capture-avoiding substitution or the *open* operation, 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 are part of the bigger result. The general outline of the proof has been described in Section 2.3.2.

4.2.1 Typed vs. untyped proofs

As mentioned previously, when talking about the terms of the λ-Y calculus, we generally refer to simply typed terms, such as $\Gamma \vdash \lambda x.Y_\sigma : \tau \to (\sigma \to \sigma) \to \sigma$. However, the definitions of reduction seen so far and the consecutive proofs using these definitions don't use simply typed λ-Y terms, operating instead on untyped terms. The simplest reason why this is the case is one of convenience and simplicity.
As is the case in most proofs of the Church Rosser Theorem, the result is usually proved for untyped terms of the λ-calculus and then extended to simply typed terms by simply restricting the terms we want to reason about. The CR theorem holds in the restricted setting of simply typed terms due to subject reduction, which says that if a term M can be given a simple type σ and β-reduces to another term M', the new term can still be typed with the same type σ.
Another reason, besides convention is convenience, specifically succinctness of code, or the lack thereof, when including simple types in the definition of β-reduction and all the subsequent lemmas and theorems. Indeed, the choice of excluding typing information wherever possible has been an engineering choice to a large degree, as it is generally not good practice to keep and pass around variables/objects which are not needed (in classical programming). The same should also apply to theorem proving, especially since notation can easily become bloated and difficult to present in a "natural" way (i.e. using the notation a mathematician would write).
Whilst it is true that the implementation of some of the proofs of Church Rosser might have been shorter, if the typing information was included directly in the definition of β-reduction, the downside to this would have been an increased complexity of proofs, resulting in potentially less understandable and maintainable code. **This then also ties into automation? + ex'le??**

4.2.2 Lemma 2.1

The first major result in both implementations is Lemma 2.1, which states that for every λ-Y term M, there is a term M', s.t. $M \ggg M'$. This is trivial for $\gg$, as we can easily prove the derived rule $(refl^*)$:

> **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 λ 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 β-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:

```
1  lemma pbeta_max_ex:
2    fixes M
3    shows "∃M'. M >>> M'"
4  apply (induct M rule:trm.induct)
5  apply auto
6  apply (case_tac "not_abst trm1")
7  apply (case_tac "not_Y trm1")
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(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`, 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:

```
goal (1 subgoal):
 1. ⋀trm1 trm2 M' M'a.
       trm1 >>> M' ⟹ trm2 >>> M'a ⟹ ∃M'. App trm1 trm2 >>> M'
```

Lines 6 and 7 correspond to doing a case analysis on `trm1` (where `M = App trm1 trm2`). We end up with 3 goals, corresponding to `trm1` either being a λ-term, Y-term or neither (shown below in reverse order):

```
1. ... not_abst trm1 ⟹ not_Y trm1 ⟹ ∃M'. App trm1 trm2 >>> M'
2. ... not_abst trm1 ⟹ ¬ not_Y trm1 ⟹ ∃M'. App trm1 trm2 >>> M'
3. ... ¬ not_abst trm1 ⟹ ∃M'. App trm1 trm2 >>> 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 `trm1` is either a λ-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 λ-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 λ case in both versions of the proof, we clearly see the differences in the treatment of binders:

Nominal:

```
∧x M. ∃M'. M >>> M' ⇒ ∃M'. Lam [x]. M >>> M'
```

Locally nameless:

```
∧L M. finite L ⇒
        (∧x. x ∉ L ⇒ trm M^FVar x) ⇒
        (∧x. x ∉ L ⇒ ∃M''. M^FVar x >>> M'') ⇒ ∃M'. Lam M >>> M'
```

Unlike in the nominal proof, where from `M >>> M'` we get `Lam [x]. M >>> Lam [x]. M'` by (abs) immediately, the proof of `∃M'. Lam M >>> M'` in the LN mechanization is not as trivial. The difficulty arises with the precondition $\forall x \notin L.\ M^x \Rightarrow_\gamma M'^x$ 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 $\exists M'.\ \forall x \notin L.\ M^x \ggg M'^x$

[1] While the original goal is `∃M'. Lam M >>> M'`, since there is only one possible "shape" for the right-hands side term, namely `M'` must be a λ-term, we can easily rewrite this goal as `∃M'. Lam M >>> Lam M'`.

[2] The nominal version was approximately 770 lines of code vs. 1180 for the LN version, making it about 50% longer.

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 4.8. [Close operation]

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

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

Example 4.3. To demonstrate the close operation, take the term λxy. 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. $term(M) \implies \{k \to y\}\{k \leftarrow x\}M = M[y/x]$

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

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

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

Starting from the goal (4.1), we expand the definitions of *open*, *close* and substitution for the

> λ case in (4.2). (4.3) holds by injectivity of λ. 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* $\{k+1 \to y\}\{k+1 \leftarrow x\}M^z = (M^z)[y/x]$.
>
> □

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 $\forall y \notin L.\ M^y \ggg (^{\backslash x}M')^y$.

Should I go into more detail here or just wrap it up by saying how much more code was necessary over the nominal version??

4.2.3 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:

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

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

4.2.3.1 (β) 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 **??reduction trees??** are valid:

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

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 $M' \gg P_{max}[Q_{max}/x]$:

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

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* $\gg$*:*
>
> $$(||_{subst})\ \frac{M \gg M' \qquad N \gg N'}{M[N/x] \gg M'[N'/x]}$$

4.2.3.2 Nominal implementation

The code below shows the proof of the (β) 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.

4.2.3.3 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 λ-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.

4.3 Subject reduction

This chapter is already quite long, so this section might end up being quite brief, as the main differences between the mechanizations have already been illustrated...I think...Or?

4.4 Verdict

Having given an overview of the main technical points of the λ-Y calculus mechanization, we concluded that on the whole, neither mechanization proved to be significantly better than the other. Whilst the LN mechanization proved to have significantly higher "obvious" mechanization overheads[3] in therms 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.
Continuing with the next round of comparison between the two theorem provers, Isabelle and

[3] The nominal version was approximately 770 lines of code vs. 1180 for the LN version, making it about 50% longer.

Agda, this point was the main reason to chose LN over nominal sets, as implementing the LN version of the calculus requires a lot less “background” theory, which was especially important in Agda, where nominal set support is a lot less mature than in Isabelle.

5. Isabelle vs. Agda

The formalization of the terms and reduction rules of the λ-Y calculus presented here is a locally nameless presentation due to B. Aydemir et al. (2008). The basic definitions of λ-terms and β-reduction were borrowed from an implementation of the λ-calculus with the associated Church Rosser proof in Agda, by Mu (2011).

One of the most obvious 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 apparent 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 another command that invokes the automatic solver, which tries to prove all the goals in the given context.

In comparison, 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 needs to fill in, to construct the proof. Using the emacs/atom 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.

5.1 Automation

As seen in the first example, Isabelle includes several automatic provers of varying complexity, including `simp`, `auto`, `blast`, `metis` and others. These are 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 common 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 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.

However, 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 wihtout 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 longer 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
```

5.2 Proofs-as-programs

As was already mentioned, Agda treats proofs as programs, and therefore provides direct access to proof objects. In Isabelle, the proof goal is 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 the example above. In the 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 functional 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 previous 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. Since all functions in Agda terminate, this operation on types is safe **(not sure this is clear enough... im not entirely sure why... found here: http://people.inf.elte.hu/divip/AgdaTutorial/Functions.Equality_Proofs.html#automatic-reduction-of-types)**.

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 tranform 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
```

We can thus see that using Isar style proofs and Agda reasoning ends up being rather similar in practice.

5.3 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-β 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 λ-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 λ-Y calculus along with the proof of subject invariance for intersection types.
Whilst the proof is not novel, there is, to our knowledge, no known fully formal version of it for the λ-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 that were necessary for the implementation.
The chapter is presented in sections, each explaining implementation details that had to be considered and any tweaks to the definitions, presented in Section 2.4, that needed to be made. Some of the definitions presented early on in this chapter, thus undergo several revisions, as we discuss the necessities for these changes in a pseudo-chronological manner in which they occurred throughout the mechanization.

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 (`ITypeI` $/\mathcal{T}$) types:

```
1  data ITypeI : Set
2  data IType : Set
3
4  data IType where
5    ψ : IType
6    _~>_ : (s : ITypeI) -> (t : IType) -> IType
7
8  data ITypeI where
9    ∩ : List IType -> ITypeI
```

The reason why the intersection `ITypeI` 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 `ITypeI` are somewhat redundant, in that `ITypeI` only has one constructor $\cap$ and therefore, any instance of `IType` in the definition of `ITypeS` 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 λ-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 σ:

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

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, τ ranges over strict intersection types $\mathcal{T}_s$, with τ_i, τ_j ranging over non-strict intersection types $\mathcal{T}$, and A, B range over simple types σ:

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

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

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, τ, τ' range over $\mathcal{T}_s$, $\tau_i, \ldots, \tau_n$ range over $\mathcal{T}$ and A, B range over σ:

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

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

The typing relation **add link to future def** was also modified to include type refinement **... didn't want to include it here because it will differ significantly from the initial**

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 **?verbally/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:

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

To give a motivation as to why we chose the formulation of these rules, 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 now tried to turn the first property above into a rule, such as:

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

Where τ is a strict type and τ_i is an intersection, we would immediately get an error because the type signature of $\subseteq$ (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 τ 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. For example, 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:

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

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:

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

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 rules clearly doesn't capture the properties, outlined in Definition 2.8.

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 complete the previously open branch in the example above. 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`'s. This iterated style of rules was adopted throughout this chapter for all definitions involving `List IType`'s, 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 thus $[] +\!\!+ \tau_j ::_\ell A \equiv \tau_j ::_\ell A$, which 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:

$$(cons)\ \frac{\tau ::_A \qquad \tau_s +\!\!+ \tau_j ::_\ell A}{\tau : \tau_s +\!\!+ \tau_j ::_\ell A}$$

□

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

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[1].

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[2].

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[3].

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[4].

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[5].

———. 2013. "Model Checking Higher-Order Programs." *J. ACM* 60 (3). New York, NY, USA: ACM: 20:1–20:62. doi:10.1145/2487241.2487246[6].

Mu, Shin-Cheng. 2011. "Proving the Church-Rosser Theorem Using a Locally Nameless Representation." Blog. http://www.iis.sinica.edu.tw/~scm/2011/proving-the-church-rosser-theorem.

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[7].

Pfenning, F., and C. Elliott. 1988. "Higher-Order Abstract Syntax." In *Proceedings of the Acm Sigplan 1988 Con-*

[1] https://doi.org/10.1007/11541868_4
[2] https://doi.org/10.1145/1328438.1328443
[3] https://doi.org/10.4230/LIPIcs.FSTTCS.2013.91
[4] https://doi.org/10.1145/138027.138060
[5] https://doi.org/10.1145/1480881.1480933
[6] https://doi.org/10.1145/2487241.2487246
[7] https://doi.org/10.1109/LICS.2006.38

ference on Programming Language Design and Implementation, 199–208. PLDI '88. New York, NY, USA: ACM. doi:10.1145/53990.54010[8].

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[9].

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[10].

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[11].

[8] https://doi.org/10.1145/53990.54010
[9] https://doi.org/10.1007/3-540-48660-7_14
[10] https://doi.org/10.1145/2578855.2535873
[11] https://doi.org/10.1145/2603088.2603133