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.
Since 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, 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, 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 to capture in such type systems. In recent years, advances in higher order model checking (HOMC) have been made (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 the λ-Y calculus with the intersection-type system described by ? and formally proving certain key properties of the system.
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 project 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 split into roughly two main parts, witht he first part exploring and evaluating different formalizations of the simply-typed λ-Y calculus together with the proof of the Church Rosser Theorem. The second part focuses on implemnting the interestion-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

- 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. For example, 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:

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

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

The function e takes two additional parameters, k and m. k keeps track of the scope from the root of the term and m is a map from bound variables to the levels they were bound at. In the variable case, if x_n appears in m, it is a bound variable, and it's index can be calculated by taking

the difference between the current index and the index $m(x_k)$, at which the variable was bound. If x_n is not in m, then the variable is encoded by adding the current level k to n.
In the abstraction case, x_n is added to m with the current level k, possibly overshadowing a previous binding of the same variable at a different level (like in $\lambda x_1.(\lambda x_1.x_1)$) and k is incremented, going into the body of the abstraction.

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

We need two auxiliary functions to define g:

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

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

The function g is then defined as $g(t) \equiv h_0^{\Diamond_0(t)+1}(t)$. As mentioned above, the complicated definition has to do with avoiding free variable capture. A term like $\lambda(\lambda\ 2)$ intuitively represents a named 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 example, the substitution lemma, which in the named setting would be stated as:

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

becomes the following statement in the nameless formalization:

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

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 λ-Y calculus

?Tie in λ-Y calculus to HOMC?

2.2.1 Definitions

The first part of this project focuses on formalizing the simply typed λ-Y calculus and the proof of confluence for this calculus. The usual/informal definition of the λ-Y terms and the simple types are given below:

Definition 2.1 (λ-Y types and terms).
Types:

$$\sigma ::= \varphi \mid \sigma \to \sigma$$

Terms:

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

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 $(A \to A) \to A$. The usual definition of β-reduction is then augmented with the (Y) rule (this is the typed version of the rule):

$$(Y)\ \frac{\Gamma \vdash M : \sigma \to \sigma}{\Gamma \vdash Y_\sigma M \Rightarrow M(Y_\sigma M) : \sigma}$$

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

2.2.2 Church-Rosser Theorem

The Church-Rosser Theroem 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. $\mathbf{dp}(\Rightarrow^*)$, where:

Definition 2.2 ($\mathbf{dp}(R)$). A relation R has the *diamond property*, i.e. $\mathbf{dp}(R)$, iff

$$\forall a, b, c.\ aRb \wedge aRc \implies \exists d.\ bRd \wedge cRd$$

The proof of confluence of $\Rightarrow_Y$, the 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)).

This proof proceeds by first defining a relation called the parallel beta reduction $\gg$, which is a reflexive and parallel βY-reduction, in that it allows simultaneous reduction of multiple parts of a term:

Definition 2.3 ($\gg$).

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

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

Since the transitive closure of $\gg$ is the same as the transitive closure of $\Rightarrow_Y$, i.e. $\gg^* = \Rightarrow_Y^*$, it is enough to show $\mathbf{dp}(\gg^*)$. The proof therefore proceeds in roughly three main steps:

1. Prove $\mathbf{dp}(\gg)$
2. Show $\mathbf{dp}(\gg) \implies \mathbf{dp}(\gg^*)$
3. Show $\gg^* = \Rightarrow_Y^*$

The most interesting step is to show $\mathbf{dp}(\gg)$, as 2. and 3. follow fairly straightforwardly. To prove $\mathbf{dp}(\gg)$, the formalizations in this project follow the aforementioned proof by Takahashi (1995). The main idea in this proof is to define another relation called the maximum parallel reduction $\ggg$, which contracts all redexes in a given term with a single step:

Definition 2.4 ($\ggg$).

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

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

This relation differs from $\gg$ only in the (app) rule, which can only be applied if M is not a λ or Y 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 ($\exists \ggg$). $\forall M.\ \exists M_{max}.\ M \ggg M_{max}$

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:

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

This triangle can be formulated as the following lemma:

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

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

□

Lemma 2.3. *dp*$(\gg)$.

Proof. We can now prove **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 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 $M'' \gg M_{max}$.

□

3. Methodology

3.1 Comparison of formalizations

The idea of formalizing a functional language in multiple theorem provers and objectively assesing 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).
Whilst 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, there seems to have been no attempt made at analyzing and comparing the different formalizations and drawing any conclusions with regards to the stated aims of the challenge.

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 and tries apply them in the assessment of the two different mechanizations of binders for the λ-Y calculus (Chapter 4), namely nominal and locally nameless implementations, as well as in the comparison two different theorem provers (Chapter 5), Isabelle and Agda.

3.1.1 Comparison 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 tree 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 mechanizing intersection types for the λ-Y calculus and the associated results. http://www.sharelatex.com...

3.1.1.1 Mechanization/implementation overheads

4. Isabelle vs. Isabelle

aaaa

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).

The proofs of confluence/Church Rosser were formalized using the paper by R. Pollack (1995), which describes a coarser proof of Church Rosser than the one formalized by Mu (2011). This proof uses the notion of a maximal parallel reduction, introduced by Takahashi (1995) to simplify the inductive proof of confluence.

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 previously, 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.
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.

5.1.1 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.

5.1.2 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.

The following lemma states that the parallel-β maximal reduction preserves local closure:

$$t >>> t' \implies \text{term } t \wedge \text{term } t'$$

For simplicity, we will prove a slightly simpler version, namely: $t >>> t' \implies \text{term } t$. 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 $t >>> t'$, 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'`. 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'`.

```
>>>-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

In this section, we will work with both the simple types introduced earlier (definition given again below), as well as intersection types, defined in the following way:

Definition 6.1 (Intersection Types). Note that o and φ are constants. ω is used to denote an empty list of strict intersection types. The following sugar notation will also occasionally be used: $\bigcap \tau \equiv [\tau]$ and $\tau \cap \tau' \equiv \bigcap \tau +\!\!+ \bigcap \tau' \equiv [\tau, \tau']$.

i) Simple types:

$$\sigma ::= \mathsf{o} \mid \sigma \to \sigma$$

ii) Intersection types:

$$\mathcal{T}_s ::= \varphi \mid \mathcal{T} \rightsquigarrow \mathcal{T}$$
$$\mathcal{T} ::= \mathsf{List}\ \mathcal{T}_s$$

The reason why $\mathcal{T}$ is defined as a list of strict types $\mathcal{T}_s$ is due to the requirement that the types in $\mathcal{T}$ be finite. The decision to use lists was taken because the Agda standard library includes a definition of lists along with definitions of list membership $\in$ for lists and other associated lemmas.

Next, we redefine the λ-terms slightly, by annotating the terms with simple types. The reason for this will be clear later on.

Definition 6.2 (Terms). Let σ range over simple types in the following definition:

i) Simply-typed terms:

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

ii) Simply-typed pre-terms:

$$M' ::= x_\sigma \mid i \mid M'M' \mid \lambda_\sigma.M' \mid Y_\sigma \text{ where } x \in Var \text{ and } i \in \mathbb{N}$$

Note that both definitions implicitly assume that in the case of application, a well formed simply-typed term will be of the form st, where s has some simple type $A \to B$ and t is typed with the simple type A. Sometimes the same subscript notation will be used to indicate the simple type of a given pre-/term, for example: $m_{A\to B}$. Also, rather confusingly, the simple type of Y_A is $(A \to A) \to A$, and thus Y_A should not be confused with a constant Y having a simple type A. **Maybe use something like this instead?:** $m_{:A\to B}$ i.e. $Y_{A:(A\to A)\to A}$.
The typed versions of substitution and the open and close operations are virtually identical to the untyped versions.

6.1 Type refinement

Next, we introduce the notion of type refinement by defining the refinement relation ::, between simple types and intersection types.

Definition 6.3 (::). Since intersection types are defined in terms of strict ($\mathcal{T}_s$) and non-strict ($\mathcal{T}$) intersection types, for correct typing, the definition of :: 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:

$$(base)\ \frac{}{\varphi ::_s \mathsf{o}} \qquad (arr)\ \frac{\tau_i :: A \qquad \tau_j :: B}{\tau_i \rightsquigarrow \tau_j ::_s A \longrightarrow B}$$

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

Having a notion of refinement, we define a restricted version of a subset relation on intersection types, which is defined only for pairs of intersection types, which refine the same simple type.

Definition 6.4 ($\subseteq^A$). 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:

$$(base)\ \frac{}{\varphi \subseteq_s^{\mathsf{o}} \varphi} \qquad (arr)\ \frac{\tau_i \subseteq^A \tau_j \qquad \tau_m \subseteq^B \tau_n}{\tau_j \rightsquigarrow \tau_m \subseteq_s^{A \to B} \tau_i \rightsquigarrow \tau_n}$$

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

$$(\rightsquigarrow\cap)\ \frac{(\tau_i \rightsquigarrow (\tau_j +\!\!+ \tau_k),\ \tau_m) :: A \longrightarrow B}{(\tau_i \rightsquigarrow (\tau_j +\!\!+ \tau_k),\ \tau_m) \subseteq^{A \to B} (\tau_i \rightsquigarrow \tau_j,\ \tau_i \rightsquigarrow \tau_k,\ \tau_m)}$$

$$(trans)\ \frac{\tau_i \subseteq^A \tau_j \qquad \tau_j \subseteq^A \tau_k}{\tau_i \subseteq^A \tau_k}$$

It's easy to show the following properties hold for the $\subseteq^A$ and :: relations:

Lemma 6.1 ($\subseteq \implies ::$). *i)* $\tau \subseteq_s^A \delta \implies \tau ::_s A \wedge \delta ::_s A$
ii) $\tau_i \subseteq^A \delta_i \implies \tau_i :: A \wedge \delta_i :: A$

Proof. By **?mutual?** induction on the relations $\subseteq_s^A$ and $\subseteq^A$.

□

Lemma ($\subseteq$ admissible) The following rules are admissible in $\subseteq_s^A$ / $\subseteq^A$:

i) $(refl_s)\ \frac{\tau ::_s A}{\tau \subseteq_s^A \tau} \qquad (refl)\ \frac{\tau_i :: A}{\tau_i \subseteq^A \tau_i} \qquad (trans_s)\ \frac{\tau \subseteq_s^A \tau' \qquad \tau' \subseteq_s^A \tau''}{\tau \subseteq_s^A \tau''} \qquad (\subseteq)\ \frac{\tau_i \subseteq \tau_j}{\tau_i \subseteq^A \tau_j} (\tau_j :: A)$

ii) $(\mathbin{+\!\!+}_L)\,\dfrac{\tau_i :: A \qquad \tau_j \subseteq^A \tau_{j'}}{\tau_i \mathbin{+\!\!+} \tau_j \subseteq^A \tau_i \mathbin{+\!\!+} \tau_{j'}} \qquad (\mathbin{+\!\!+}_R)\,\dfrac{\tau_i \subseteq^A \tau_{i'} \qquad \tau_j :: A}{\tau_i \mathbin{+\!\!+} \tau_j \subseteq^A \tau_{i'} \mathbin{+\!\!+} \tau_j} \qquad (glb)\,\dfrac{\tau_i \subseteq^A \tau_k \qquad \tau_j \subseteq^A \tau_k}{\tau_i \mathbin{+\!\!+} \tau_j \subseteq^A \tau_k}$

iii) $(mon)\,\dfrac{\tau_i \subseteq^A \tau_j \qquad \tau_{i'} \subseteq^A \tau_{j'}}{\tau_i \mathbin{+\!\!+} \tau_{i'} \subseteq^A \tau_j \mathbin{+\!\!+} \tau_{j'}}$

iv) $(\rightsquigarrow\cap')\,\dfrac{\tau_i :: A \qquad \tau_j :: A}{\bigcap((\tau_i \mathbin{+\!\!+} \tau_j) \rightsquigarrow (\tau_i \mathbin{+\!\!+} \tau_j)) \subseteq^{A \to B} \tau_i \rightsquigarrow \tau_i \cap \tau_j \rightsquigarrow \tau_j}$

Proof:

i) By induction on τ and τ_i.

ii) By induction on $\tau_i \subseteq^A \tau_{i'}$.

iii) $$(glb)\,\dfrac{(trans)\,\dfrac{\tau_i \subseteq^A \tau_j \qquad (\subseteq)\,\dfrac{\dfrac{}{\tau_j \subseteq \tau_j \mathbin{+\!\!+} \tau_{j'}}}{\tau_j \subseteq^A \tau_j \mathbin{+\!\!+} \tau_{j'}}}{\tau_i \subseteq^A \tau_j \mathbin{+\!\!+} \tau_{j'}} \qquad (trans)\,\dfrac{\tau_{i'} \subseteq^A \tau_{j'} \qquad (\subseteq)\,\dfrac{\dfrac{}{\tau_{j'} \subseteq \tau_j \mathbin{+\!\!+} \tau_{j'}}}{\tau_{j'} \subseteq^A \tau_j \mathbin{+\!\!+} \tau_{j'}}}{\tau_{i'} \subseteq^A \tau_j \mathbin{+\!\!+} \tau_{j'}}}{\tau_i \mathbin{+\!\!+} \tau_{i'} \subseteq^A \tau_j \mathbin{+\!\!+} \tau_{j'}}$$

iv) Follows from $(\rightsquigarrow\cap)$, $(cons)$ and $(trans)$.

6.2 Intersection-type assignment

Having annotated the λ-terms with simple types, the following type assignment only permits the typing of simply-typed λ-terms with an intersection type, which refines the simple type of the λ-term:

Definition (Intersection-type assignment)

$$(var)\,\dfrac{\exists (x, \tau_i, A) \in \Gamma.\ \bigcap \tau \subseteq^A \tau_i}{\Gamma \Vdash_s x_A : \tau} \qquad (app)\,\dfrac{\Gamma \Vdash_s u_{A \to B} : \tau_i \rightsquigarrow \tau_j \qquad \Gamma \Vdash v_A : \tau_i}{\Gamma \Vdash_s uv_B : \tau}\,(\bigcap \tau \subseteq^B \tau_j)$$

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

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

$$(nil)\,\dfrac{}{\Gamma \Vdash m : \omega} \qquad (cons)\,\dfrac{\Gamma \Vdash_s m : \tau \qquad \Gamma \Vdash m : \tau_i}{\Gamma \Vdash m : \tau, \tau_i}$$

In the definition above, Γ is the typing context, consisting of triples of the variable name and the corresponding intersection and simple types. Γ is defined as a list of these triples in the Agda implementation. It is assumed in the typing system, that Γ is well-formed. Formally, this can be expressed in the following way:

Definition (Well-formed intersection-type context)

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

6.2.1 Subtyping

In the typing system, the rules (Y) and $(\rightsquigarrow\bigcap)$ are defined in a slightly more complicated way than might be necessary. For example, one might assume, the (Y) rule could simply be:

$$(Y)\ \frac{}{\Gamma \Vdash_s Y_A : \bigcap(\tau_x \rightsquigarrow \tau_x) \rightsquigarrow \tau_x}$$

The reason why the more complicated forms of both rules were introduced was purely an engineering one, namely to make the proof of sub-typing/weakening possible, as the sub-typing rule is required in multiple further proofs:

Lemma (Sub-typing) The following rule(s) are admissible in $\Vdash_s / \Vdash$:

$$(\supseteq_s)\ \frac{\Gamma \Vdash_s m_A : \tau}{\Gamma' \Vdash_s m_A : \tau'}\ (\Gamma' \subseteq_\Gamma \Gamma, \tau \supseteq^A_s \tau') \qquad (\supseteq)\ \frac{\Gamma \Vdash m_A : \tau_i}{\Gamma' \Vdash m_A : \tau_j}\ (\Gamma' \subseteq_\Gamma \Gamma, \tau_i \supseteq^A_s \tau_j)$$

Proof: Ommited.

The relation $\Gamma \subseteq_\Gamma \Gamma'$ is defined for any well-formed contexts Γ, Γ', where for each triple $(x, \tau_i, A) \in \Gamma$, there is a corresponding triple $(x, \tau_j, A) \in \Gamma'$ s.t. $\tau_i \subseteq^A \tau_j$.

6.2.2 Inversion lemmas

The shape of the derivation tree is not always unique for arbitrary typed term $\Gamma \Vdash_s m : \tau$. For example, given a typed term $\Gamma \Vdash_s \lambda_A.m : \tau_i \rightsquigarrow \tau_j$, either of the following two derivation trees, could be valid:

$$(abs)\ \frac{\dfrac{\vdots}{\forall x \notin L.\ (x, \tau_i, A), \Gamma \Vdash m^x : \tau_j}}{\Gamma \Vdash_s \lambda_A.m : \tau_i \rightsquigarrow \tau_j}$$

$$(\rightsquigarrow\bigcap)\ \frac{\dfrac{\vdots}{\Gamma \Vdash_s \lambda_A.m_B : \tau_i \rightsquigarrow \tau_p} \qquad \dfrac{\vdots}{\Gamma \Vdash_s \lambda_A.m_B : \tau_i \rightsquigarrow \tau_q}}{\Gamma \Vdash_s \lambda_A.m_B : \tau_i \rightsquigarrow \tau_j}\ (\tau_j \subseteq^B \tau_p +\!\!+\ \tau_q)$$

However, it is obvious that the second tree will always necessarily have to have an application of (abs) in all its branches. Because it will be necessary to reason about the shape of the typing derivation trees, it is useful to prove the following inversion lemmas:

Lemma (Y-inv, abs-inv)

i) $\Gamma \Vdash_s Y_A : \tau_i \rightsquigarrow \tau_j \implies \exists \tau_x.\ \bigcap(\tau_x \rightsquigarrow \tau_x) \subseteq^{A \to A} \tau_i \wedge \tau_j \subseteq^A \tau_x$

ii) $\Gamma \Vdash_s \lambda_A.m : \tau_i \rightsquigarrow \tau_j \implies \exists L.\ \forall x \notin L.\ (x, \tau_i, A), \Gamma \Vdash m^x : \tau_j$

Proof:

i) There are two cases to consider, one, where the last rule in the derivation tree of $\Gamma \Vdash_s Y_A : \tau_i \rightsquigarrow \tau_j$ was (Y). Otherwise, the last rule was $(\rightsquigarrow\bigcap)$:

(Y): Follows immediately.
$(\rightsquigarrow\bigcap)$: We must have a derivation tree of the shape:

$$(\rightsquigarrow\bigcap)\ \dfrac{\dfrac{\vdots}{\Gamma \Vdash_s Y_A : \tau_i \rightsquigarrow \tau_p} \qquad \dfrac{\vdots}{\Gamma \Vdash_s Y_A : \tau_i \rightsquigarrow \tau_q}}{\Gamma \Vdash_s Y_A : \tau_i \rightsquigarrow \tau_j}\ (\tau_j \subseteq^B \tau_p \mathbin{+\!\!+} \tau_q)$$

Then by IH, we have:

- $\exists \tau_{xp}.\ \bigcap(\tau_{xp} \rightsquigarrow \tau_{xp}) \subseteq^{A \longrightarrow A} \tau_i \wedge \tau_p \subseteq^A \tau_{xp}$ and
- $\exists \tau_{xq}.\ \bigcap(\tau_{xq} \rightsquigarrow \tau_{xq}) \subseteq^{A \longrightarrow A} \tau_i \wedge \tau_q \subseteq^A \tau_{xq}$

We then take $\tau_x \equiv \tau_{xp} \mathbin{+\!\!+} \tau_{xq}$:

$$(trans)\ \dfrac{(trans)\ \dfrac{(\rightsquigarrow\cap')\ \dfrac{}{\bigcap(\tau_x \rightsquigarrow \tau_x) \subseteq^{A \longrightarrow A} \tau_{xp} \rightsquigarrow \tau_{xp} \cap \tau_{xq} \rightsquigarrow \tau_{xq}} \quad (mon)\ \dfrac{(IH)\ \dfrac{}{\tau_{xp} \rightsquigarrow \tau_{xp} \subseteq^{A \longrightarrow A} \tau_i} \quad (IH)\ \dfrac{}{\tau_{xq} \rightsquigarrow \tau_{xq} \subseteq^{A \longrightarrow A} \tau_i}}{\tau_{xp} \rightsquigarrow \tau_{xp} \cap \tau_{xq} \rightsquigarrow \tau_{xq} \subseteq^{A \longrightarrow A} \tau_i \mathbin{+\!\!+} \tau_i}}{\bigcap(\tau_x \rightsquigarrow \tau_x) \subseteq^{A \longrightarrow A} \tau_i \mathbin{+\!\!+} \tau_i} \quad (\subseteq)\ \dfrac{\dfrac{}{\tau_i \mathbin{+\!\!+} \tau_i \subseteq \tau_i}}{\tau_i \mathbin{+\!\!+} \tau_i \subseteq^{A \longrightarrow A} \tau_i}}{\bigcap(\tau_x \rightsquigarrow \tau_x) \subseteq^{A \longrightarrow A} \tau_i}$$

$$(trans)\ \dfrac{\dfrac{}{\tau_j \subseteq^A \tau_p \mathbin{+\!\!+} \tau_q} \quad (mon)\ \dfrac{(IH)\ \dfrac{}{\tau_p \mathbin{+\!\!+} \ \subseteq^A \tau_{xp}} \quad (IH)\ \dfrac{}{\tau_q \mathbin{+\!\!+} \ \subseteq^A \tau_{xq}}}{\tau_p \mathbin{+\!\!+} \tau_q \subseteq^A \tau_x}}{\tau_j \subseteq^A \tau_x}$$

ii) Follows in a similar fashion.

6.3 Proofs of subject expansion and reduction

An interesting property of the intersection types, is the fact that they admit both subject expansion and subject reduction, namely $\Vdash$ is closed under β-equality. Subject expansion and reduction are proved in two separate lemmas:

Theorem ($\Vdash$ closed under $=_\beta$)

i) $\Gamma \Vdash_s m : \tau \implies m \Rightarrow_\beta m' \implies \Gamma \Vdash_s m' : \tau$
ii) $\Gamma \Vdash m : \tau_i \implies m \Rightarrow_\beta m' \implies \Gamma \Vdash m' : \tau_i$
iii) $\Gamma \Vdash_s m' : \tau \implies m \Rightarrow_\beta m' \implies \Gamma \Vdash_s m : \tau$
iv) $\Gamma \Vdash m' : \tau_i \implies m \Rightarrow_\beta m' \implies \Gamma \Vdash m : \tau_i$

Proof: By induction on $\Rightarrow_\beta$. The proofs in both directions follow by straightforward induction for all the rules except for (Y) and $(beta)$. Note that the (Y) rule here is not the typing rule, but rather the reduction rule $Y_A m \Rightarrow_\beta m(Y_A m)$.

i) (Y): By assumption, we have $Y_A m \Rightarrow_\beta m(Y_A m)$ and $\Gamma \Vdash_s Y_A m : \tau$. By case analysis of the last rule applied in the derivation tree of $\Gamma \Vdash_s Y_A m : \tau$, we have two cases:
   - (app) We have:

$$(app)\ \frac{\dfrac{\vdots}{\Gamma \Vdash_s Y_A : \tau_i \rightsquigarrow \tau_j} \qquad \dfrac{\vdots}{\Gamma \Vdash m_{A\to A} : \tau_i}}{\Gamma \Vdash_s Y_A m : \tau}\ \left(\bigcap \tau \subseteq^A \tau_j\right)$$

Then, by (Y-inv) we have some τ_x s.t $\bigcap(\tau_x \rightsquigarrow \tau_x) \subseteq^{A\to A} \tau_i \wedge \tau_j \subseteq^A \tau_x$.

- $(\rightsquigarrow\bigcap)$ Then we have:

$$(\rightsquigarrow\bigcap)\ \frac{\dfrac{\vdots}{\Gamma \Vdash_s Y_{B\to C} m : \tau_i \rightsquigarrow \tau_j} \qquad \dfrac{\vdots}{\Gamma \Vdash_s Y_{B\to C} m : \tau_i \rightsquigarrow \tau_k}}{\Gamma \Vdash_s Y_{B\to C} m : \tau_i \rightsquigarrow \tau_{jk}}\ \left(\tau_{jk} \subseteq^C \tau_j \mathbin{+\!\!+} \tau_k\right)$$

  Where $A \equiv B \to C$.
  By IH, we get $\Gamma \Vdash_s m(Y_{B\to C} m) : \tau_i \rightsquigarrow \tau_j$ and $\Gamma \Vdash_s m(Y_{B\to C} m) : \tau_i \rightsquigarrow \tau_k$, thus from $(\rightsquigarrow\bigcap)$ it follows that $\Gamma \Vdash_s m(Y_{B\to C} m) : \tau_i \rightsquigarrow \tau_{jk}$

Test reference to lemma 6.1.

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].

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.

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[3].

Kobayashi, Naoki. 2013. "Model Checking Higher-Order Programs." *J. ACM* 60 (3). New York, NY, USA: ACM: 20:1–20:62. doi:10.1145/2487241.2487246[4].

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.

Pfenning, F., and C. Elliott. 1988. "Higher-Order Abstract Syntax." In *Proceedings of the Acm Sigplan 1988 Conference on Programming Language Design and Implementation*, 199–208. PLDI '88. New York, NY, USA: ACM. doi:10.1145/53990.54010[5].

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[6].

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[7].

Takahashi, M. 1995. "Parallel Reductions in λ-Calculus." *Information and Computation* 118 (1): 120–27. doi:http://dx.doi.org/10.1006/inco.1995.1057[8].

Tsukada, Takeshi, and C.-H. Luke Ong. 2014. "Compositional Higher-Order Model Checking via $$-Regular Games over

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

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

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

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

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

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

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

[8] https://doi.org/http://dx.doi.org/10.1006/inco.1995.1057

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[9].

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