# Isabelle vs. Agda

The formalization of the terms and reduction rules of the $\lambda$-Y calculus presented here is a locally nameless presentation due to Aydemir et al. (2008). The basic definitions of $\lambda$-terms and $\beta$-reduction were borrowed from an implementation of the $\lambda$-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.

## 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.

### 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.

### 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-$\beta$ 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')
```

# References


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.

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.

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

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