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 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.

Programs as proofs

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:

```
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 are then discharged by the call to `auto`.

In comparison, the Agda proof exposes the proof objects to the user directly. Instead of using commands modifying the proof state, one begins with a definition of the lemma in the following shape:

```
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 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 discharge trivial proof steps, which usually follow form 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 discharged by first unfolding the definition of substitution for `FVar`, where

```
(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)])
...
```

Now, the first goal looks like this:

```
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 weaker than the `simp` tactic. It may therefore seem that Agda will be much more cumbersome to reason in than Isabelle. This, however, turned out not to be the case, due to Agda's type system and the powerful pattern matching as well as direct access to the proof goals.

As was already mentioned, Agda treats proofs as programs, and therefore provides direct access to proof objects. Whereas 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 we need to modify or reason with the assumptions, as was seen in the example above. Here we might rely on tactics such as `drule` which can be used to apply rules to the premises rather than the conclusion. 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 limitations don't arise, 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 inf not needed.
This way of reasoning is also supported in Isabelle via the use of the Isar proof language, where the proof of `subst_fresh` can be expressed in the following way:

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

This representation is more natural (and readable) to humans, as the reasoning steps have been separated out into ‘mini-lemmas’ (the command `have` creates an new separate lemma 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.

Pattern matching

Automatic inference vs.

```
show ?case unfolding 1
using 1(2) apply (cases rule:pbeta.cases)
apply simp
```

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/#comments.

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.