Lambda Calculus Domain Reasoner

Satoshi Takimoto
takimoto.s.ab@m.titech.ac.jp

May 20, 2024

1 Features

The domain reasoner is for the Untyped Lambda Calculus with the named representation of variables. The goal of an exercise is to reduce a (possibly open) lambda term into its β-normal form. Students do not have to make it $\beta\eta$-normal, but η-reduction is still a valid step.

In the domain reasoner, α-conversion is made explicit as a rewrite step. The purpose is to teach the notion of variable capture. The Lambda Calculus is all about β-reduction, and thus substitution. However, variable capture is one of the subtle points where students are likely to struggle while learning. By having α-conversion as an explicit rewrite step, students can ask for a hint of which variables to rename, for example, helping them understand variable capture.

I could have chosen the Lambda Calculus with explicit substitutions[2] as the domain instead but did not because writing out substitutions would be a lengthy process.

2 Implementation

In this section, I will only explain interesting parts of the domain reasoner in detail: a strategy for capture avoidance and an equivalence relation that captures the "unhelpfulness" of α-conversion.

2.1 A strategy for capture avoidance

The domain reasoner is implemented using the DSL provided by the `Ideas` Library[1][1]. In the DSL, valid steps of rewrite rules are described as a strategy, which is roughly a context-free language whose terminal symbols are rewrite rules[3].

The following is a strategy for β-reduction preceded by enough α-conversions for capture avoidance. In other words, β-reduction gets "blocked" until sufficient α-conversions are applied.

```
1  captureAvoidingBeta =
2    repeatS (
3      -- If the current term is a redex (\x. t) u, save the substitution x -> u
4      ruleSaveSubst
5      -- Go down to t
6      .*. ruleDown .*. ruleDownLast
7      -- Apply alpha-conversion to an appropriate subterm
8      .*. traverse [traversalFilter notShadowed] ruleAlpha
9      -- Go back up to the redex
10     .*. ruleUp .*. ruleUp)
11   -- Apply beta reduction
12   .*. liftToContext ruleBeta
```

Basically, it greedily `repeatS`s applying the α-conversion rewrite rule `ruleAlpha` to some subterm that needs to be renamed, then applies the β-reduction rewrite rule `ruleBeta`. `ruleSaveSubst` in line 4 is an administrative rule that saves to the context the substitution to apply. The saved substitution is used in `ruleAlpha` to calculate a fresh variable, as well as in `notShadowed` not to wrongly apply `ruleAlpha` to subterms where the variable to be substituted is shadowed.

[1] I used the following branch to develop with a more recent version of GHC: `https://github.com/ideas-edu/ideas/tree/ideas-bastiaan`

The following is an example of the sequences of rewrite rules valid in `captureAvoidingBeta`:

$$(\lambda x.\underline{\lambda y.\lambda z.x})(yz) \xRightarrow[v/y]{\texttt{ruleAlpha}} (\lambda x.\lambda v.\underline{\lambda z.x})(yz) \xRightarrow[w/z]{\texttt{ruleAlpha}} \underline{(\lambda x.\lambda v.\lambda w.x)(yz)} \xRightarrow{\texttt{ruleBeta}} \lambda v.\lambda w.yz$$

2.2 An equivalence relation that captures the "unhelpfulness" of α-conversion

In the `Ideas` library, we have to define two equivalences on terms for checking if a student's step is semantically/syntactically equivalent to an expected step, respectively. I specified $\alpha\beta\eta$-equivalence (up to a finite amount of reductions as a term can be non-terminating) for the former. For the latter, because we want to make α-conversion explicit as a rewrite step, α-equivalence is too flexible as α-conversion is a no-op under the equivalence, while the syntactic equality is too strict. So, I defined a new one which I call *ρ-equivalence*. Two terms are ρ-equivalent if they are α-equivalent and have the same places of binders that need to be renamed for capture avoidance (to "unblock" the subsequent β-reduction step in the `captureAvoidingBeta` strategy). The following showcases how ρ-equivalence behaves.

- $(\lambda x.\underline{\lambda y.}\underline{\lambda z.}\lambda x)(yzw) \neq_\rho (\lambda x.\lambda a.\underline{\lambda z.}\lambda x)(yzw) =_\rho (\lambda x.\lambda b.\underline{\lambda z.}\lambda x)(yzw)$
- $(\lambda x.\underline{\lambda y.}\underline{\lambda z.}\lambda x)(yzw) =_\rho (\lambda x.\underline{\lambda z.}\underline{\lambda y.}\lambda x)(yzw) =_\rho (\lambda x.\underline{\lambda y.}\underline{\lambda w.}\lambda x)(yzw)$
- $(\lambda x.\lambda a.\underline{\lambda z.}\lambda x)(yzw) \neq_\rho (\lambda x.\underline{\lambda y.}\lambda b.\lambda x)(yzw)$

References

[1] The `Ideas` library package. `https://hackage.haskell.org/package/ideas`.

[2] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. `doi:10.1017/S0956796800000186`.

[3] Bastiaan Heeren, Johan Jeuring, and Alex Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3:349–370, 05 2010. `doi:10.1007/s11786-010-0027-4`.