FORMALLY PROVING THE CORRECTNESS OF THE (UN)CURRYING REFACTORING

MICHAŁ JÓŹWIK - M.G.JOZWIK@STUDENT.TUDELFT.NL

TU Delft

1. INTRODUCTION TO AGDA

- Dependently typed functional programming language (see Figure 1)
- Used as a proof assistant
- Total language:
  - Always terminates
  - No runtime errors

```
data Vec (A : Set) : Nat → Set where
    []   : Vec A zero
    _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

Figure 1: Example of dependent type

2. MOTIVATION

- Formal proofs are important for critical software
- Refactoring allows for more readable or better performing code
- Bigger adoption of formal proofs over tests
- Extension of the refactoring topic to less researched functional programming [1]

3. OBJECTIVE

- Develop a simple functional programming language with anonymous functions
- Implement a refactoring function to perform currying (transformation from f1 to f2 as seen in Figure 2) and uncurrying, which is the inverse operation
- Create a formal proof using Agda as a proof assistant

```
-- Original function
f1 : (Int, Int) → Int
f1 (x, y) = x + y

-- Function after currying
f2 : Int → Int → Int
f2 = λ x → (λ y → x + y)
```

Figure 2: Uncurried (f1) and curried (f2) function

4. HASKELL-LIKE LANGUAGE (HLL)

For this project a small Haskell-like language is created to show this idea in a controlled and more basic environment, than a full functional programming language. A subset of functionality is seen in Figure 3.
Main properties of it include:

- Intrinsic typing
- Big-step semantics
- de Bruijn indices [2]

```
data _⊢_ : Ctx → Ty → Set where
  ...
  var  : t ∈ Γ
         -------
       → Γ ⊢ t

  fun  : (t :: Γ) ⊢ u
         ------------
       → Γ ⊢ (t ⇒ u)

  app  : Γ ⊢ (t ⇒ u)
       → Γ ⊢ t
         -------
       → Γ ⊢ u
```

```
data _⊢_ : Ctx → Ty → Set where
  ...
  fun₂ : (t :: u :: Γ) ⊢ v
         ------------------
       → Γ ⊢ (/ t / u ⇒ v)

  app₂ : Γ ⊢ (/ t / u ⇒ v)
       → Γ ⊢ t
       → Γ ⊢ u
         -------
       → Γ ⊢ v
```

Figure 3: Constructs related to anonymous functions in HLL

5. REFACTORING

The refactoring functions directly take the input programs and transform their internal representation to use alternative constructs (see Figure 4). The currying refactoring is more generalized with changes to functions and applications separately. The uncurrying refactoring uses a more restrictive approach and only changes the combination of function application and definition without changing the actual values.

```
ref-curry : Γ ⊢ p → (ref-ctx Γ) ⊢ (ref-type p)
...
ref-curry (fun₂ x)     = fun (fun (ref-curry x))
ref-curry (app₂ x arg1 arg2) =
    app (app (ref-curry x) (ref-curry arg2)) (ref-curry arg1)

ref-uncurry : Γ ⊢ p → Γ ⊢ p
...
ref-uncurry (app (app (fun (fun x)) arg2) arg1) =
    app₂ (fun₂ (ref-uncurry x)) (ref-uncurry arg1) (ref-uncurry arg2)
```

Figure 4: Implementation of refactoring functions

6. PROOF

It is shown that before and after refactoring, the following holds:

- values are changed in a controlled manner
- program is well-typed
- program terminates

To prove the change in values, a function is used to describe the mapping and used in an explicit proof (full proof available in the repository linked in QR code seen in Figure 5).

Figure 5: Repository link

HLL by definition only allows well-typed programs to be constructed, thus any valid transformation preserves well-typedness by design. Termination is preserved due to the previously mentioned value relation and due to the totality of Agda, which makes sure that the refactoring always finishes.

7. DISCUSSION

- The definition of currying is different from the usual case - it uses particular language constructs instead of tuples
- The approach is limited to two arguments
- Agda proof has remaining holes that were proven manually in the research paper

8. FUTURE WORK

- Use this research as a baseline to develop a tool for other languages
- Expand the language to use n-argument functions
- Change the approach to use tuples for multiple arguments
- Make the tool more customizable with user interaction

9. REFERENCES

[1] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On preserving the behavior in software refactoring: A systematic mapping study. Information and Software Technology, 140:106675, 2021.
[2] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. Indagationes Mathematicae, 75(5):381–392, 1972.

SUPERVISED BY: JESPER COCKX, LUKA MILJAK