Refactoring with Confidence: Correct-by-Construction Refactoring of Functional Code

Kalle Struik[1]
Supervisors: **Jesper Cockx**[1] and **Luka Miljak**[1]

Background

- Haskell is a popular functional programming language (See figure 1 for example code).

```
add :: Int -> Int -> Int
add a b = a + b
```

Figure 1. A Haskell function that adds two numbers

- Agda is a proof assistant and functional programming language. It can be used to create programs and proof their correctness using dependant types.

The Goals

- Show that it is feasible to prove the correctness of a refactoring operation using computer aided proving techniques.
- Define a Haskell-like language for our refactoring to be applied to.
- Create a refactoring operation that adds an argument to a function in our Haskell-like language (See figure 2).

```
add :: Int -> Int -> Int
add a b = a + b

...

add 1 2 -- Return 3
```

→

```
add :: Int -> Int -> Int -> Int
add a b c = a + b

...

add 1 2 0 -- Still return 3
```

Figure 2. Adding an argument to a function

- Construct a proof that, given a well-typed input, our refactoring produces a well-typed output.
- Construct a proof that our refactoring does not change the behaviour of the refactored program.

The Language

Our Haskell-like language consists of three base types, natural numbers, booleans, and functions. We also define some operations on these types such as addition, multiplication, and less-than. Of course, for our language to be useful, we also need variable lookup and function application. A partial definition of the language constructs can be seen in figure 3. The method used to define the language and its semantics is based on the PLFA book[1].

```
data _⊢_ : FContext → Context → Type → Set where
    bool   : Bool
             ---------------
           → δ × γ ⊢ tyBool

    nat    : ℕ
             ---------------
           → δ × γ ⊢ tyNat

    plus   : δ × γ ⊢ tyNat
           → δ × γ ⊢ tyNat
             ---------------
           → δ × γ ⊢ tyNat

    neg    : δ × γ ⊢ tyBool
             ---------------
           → δ × γ ⊢ tyBool

    var    : γ ∋ A
             ---------------
           → δ × γ ⊢ A
```

```
lambda : δ × γ , A ⊢ B
         ---------------
       → δ × γ ⊢ (A ⇒ B)

appl   : δ × γ ⊢ (A ⇒ B)
       → δ × γ ⊢ A
         ---------------
       → δ × γ ⊢ B

fdef   : δ × γ ⊢ (A ⇒ B)
       → (δ ,ᶠ (A ⇒ᶠ B)) × γ ⊢ C
         ---------------
       → δ × γ ⊢ C

fvar   : δ ∋ᶠ ft
         ---------------
       → δ × γ ⊢ (FType-to-Type ft)

newCtx : ∅ᶠ × ∅ ⊢ A
         ---------------
       → δ × γ ⊢ A
```

Figure 3. Definition of our language constructs in Agda

The Refactoring

Our refactoring function takes three arguments, the program to refactor, a natural number n indicating what function should be refactored, and a default value for the newly added argument.

The program to refactor should start with one or more *fdef* constructs representing top level function definitions. The nth of these constructs will be the one targeted by the refactoring. (Example in figure 4)

```
(fdef #| n = 0|#
  (lambda #| Function 1}|#)
  (fdef #| n = 1|#
    (lambda #| Function 2}|#)
    (fdef #| n = 2|#
      (lambda #| Function 3}|#)
      (#| Rest of the program|#)
    )
  )
)
```

```
(fdef
  (lambda (lambda #| Original function body |#))
  #| Somewhere in the remaining program |#
  (appl
   (fvar #| Lookup to function |#)
   (newCtx #| Default argument |#)
  )
)
```

Figure 4. Example of a program targeted by our refactoring and the effects of the refactoring.

The Proof

Since our language is intrinsically typed, our refactoring will always produce well-typed outputs.

To prove that our refactoring preserves behaviour we define the concept of *refactoring equivalence* (Agda definition in figure 5). It is defined as normal equality for natural numbers and booleans, but for function values we do something special. For these we define it as *given an refactoring equivalent argument they produce a refactoring equivalent output*.

```
_≡v_ : Val t → Val t → Set
natV x₀ ≡vₜ natV xₙ = x₀ ≡ xₙ
boolV x₀ ≡vₜ boolV xₙ = x₀ ≡ xₙ
funcV {argT = argT} {retT} Γ₀ Δ₀ b₀ ≡vₜ funcV Γₙ Δₙ bₙ =
    ∀ {argV₀ : Val argT} {argVₙ : Val argT} {argV₀≡vₜargVₙ : argV₀ ≡vₜ argVₙ}
      {retV₀ : Val retT} {retVₙ : Val retT}
    → Δ₀ × (Γ₀ ,' argV₀) ⊢ b₀ ↓ retV₀
    → Δₙ × (Γₙ ,' argVₙ) ⊢ bₙ ↓ retVₙ
    → retV₀ ≡vₜ retVₙ
```

Figure 5. Definition of *refactoring equivalence*.

To construct a proof using this definition we have written a series of functions that take an expression pre and post refactoring and output a proof that the *refactoring equivalence* relations holds between them.

Compromises

To make this project feasible within ten weeks we had to compromise on a few points.

- Our refactoring is not a source-to-source refactoring like most, but instead converts from one AST to another.
- Our Haskell-like language is only a small subset of Haskell. This subset includes
  - Function definition
  - Function calling
  - Natural numbers and basic operations (addition, multiplication) on them.
  - Booleans and basic operations (and, or, not) on them.

Future Work

Every member of our group has created their own refactoring with their own version of a Haskell-like language. It would be interesting to see if our efforts could be combined to create a group of provably correct refactoring tools. This posses a couple challenges the primary one being that all of our proofs depend on our specific choices when defining our respective languages.

Another interesting avenue to explore would be creating a set of smaller composable refactoring operations that are proven to be correct on the entire language which could be used to create larger refactoring operations which would then by extension also be correct.

References

[1] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. August 2022.

[1]EEMCS, Delft University of Technology, The Netherlands

K.H.Struik@student.tudelft.nl