Refactoring with confidence

Creating and proving the correctness of a refactoring to add arguments to functions in a functional language

Kalle Struik[1]

Supervisors: Jesper Cockx[1]**, Luka Miljak**[1]

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

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 30, 2023

Name of the student: Kalle Struik
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Luka Miljak, Koen Langendoen

Abstract

Refactoring tools are an important tool for developers, but their reliability can be questionable at times. In this paper, we show that it is feasible to formally verify refactoring tools using computer-aided proofs. To this end, we create a Haskell-like language and a refactoring operation on this language to add an extra function argument to an arbitrary function in the program. And finally, use the Agda proof assistant to construct a proof of the correctness of this refactoring.

1 Introduction

Refactoring is often used by developers to improve their code quality. This is often aided by automated tools. Some of these tools are built into IDEs, while others are used externally. These tools have one thing in common, however. They are often trusted blindly by the developer. After all these tools are made by professionals. What could go wrong?

In a 2013 study[3] it was found that the refactoring tools provided by the Eclipse IDE failed for up to 7% of cases tested. In some of these cases, the tool produced code that simply did not compile, but in others, the problems were harder to detect. In such cases, we have to rely on the developer or their own test cases to catch the issue. But as stated earlier they have no reason to doubt these tools, so they will not be as vigilant as might be required.

The cost of issues with these refactoring tools, and software in general, scales with user counts. In 2021, JetBrains stated that they have 10.1 million users [1]. If they have an issue in their refactoring tools, even if it only affects a small subset of their users, that is still a large group of developers that has to spend their time finding the issues caused by faulty tools that they did not expect to be fallible in such a way. We can imagine that the costs of these issues have the potential to balloon very quickly.

Most commonly these refactoring tools rely on testing to make sure that they do not contain any issues. But there is another way: formal verification. While it does take significantly more time to formally verify a program than to write test cases for it, it does mathematically guarantee correctness. This can be desirable in certain cases where the cost of failure is high. As explained in section 2 this cost can be quite high for refactoring tools. Therefore, it makes sense to formally verify these programs.

Previous work has been done on creating refactoring tools and formally verifying them. One such work is the implementation of a refactoring tool done by Huiqing Li and Simon Thompson in [5] with the formal verification following later in [6]. This work, however, only proves a small subset of their refactoring tool.

We aim to show that it is possible to create refactoring tools and prove their correctness using computer-aided proofs, such as those created in the Agda proof assistant. To this end, we create and prove the correctness of, a refactoring on a Haskell-like language. We limiting the scope of this paper to a refactoring that adds an argument to a top-level function. Like the paper discussed above we define a custom language to aid in the creation and proving of this refactoring. But we diverge in the way in which we prove the correctness of our refactoring. While they performed a pen and paper proof, we aim to provide a computer-checked proof using the Agda proof assistant.

To this end make the following contributions:

1. We define a Haskell-like language and its semantics written as big-step semantics.
2. We create a refactoring that can be applied to this language.
3. We construct a proof that given a well-typed expression our refactoring will produce a well-typed output.
4. We construct a proof that our refactoring does not change the behaviour of the program being refactored.

This paper is structured as follows. We will start discussing the required background information in section 2. We will then go over the definition of the Haskell-like language we have created and the refactoring we have created for this language in sections 3 and 4 respectively. After that, we will cover the proof of correctness for this refactoring in section 5. We will quickly cover the reproducibility of our research in section 6. We continue with a discussion of the results and their potential limitations in section 7 and finally we close out the paper with a conclusion in section 9.

2 Background

In this section, we go over some knowledge that is required to understand the rest of the paper. First, we cover the Agda programming language, after that, we introduce de Bruijn indices and their benefits, we then go over big-step semantics, and finally we will discuss what intrinsically typed languages are.

2.1 Agda

In this paper, we will be working with the Agda programming language. Because of the Curry-Howard correspondence, which provides a way to translate intuitionistic logic into a type system, and Agda's dependant type system, we can also use Agda as a proof assistant. For a better understanding of how Agda works and its syntax we recommend reading through the Agda documentation[2].

2.2 De Bruijn Indices

De Bruijn indices (named after and originally introduced by de Bruijn in his paper [2]) are used to have variables without giving them explicit names. Instead, we refer to them using a number. Each variable gets the next available number. So the first variable would get index 0, the second index 1, and so on. This allows us to not worry about difficult-to-solve issues such as name shadowing so we can instead focus on writing the main logic of our refactoring.

[1] https://blog.jetbrains.com/blog/2021/03/03/jetbrains-2020-21-annual-highlights-10-million-users-30-tools-and-more/

[2] https://agda.readthedocs.io/en/latest/overview.html

As an example look at the Haskell program in listing 1. In this code snippet, the variable a would have de Bruijn index 0, b index 1, and c index 2.

```
f :: Int -> Int -> Int -> Int
f a b c = ...
```

Listing 1: Simple Haskell program to show de Bruijn indices.

2.3 Big-Step Semantics

Big-Step semantics, or natural semantics as they were originally called in Kahn's paper [4], are used to formalize the behaviour of a programming language. Compared to other ways to formalize the semantics of a programming language, big-step semantics are intended to be quite close to the way programming languages would be implemented.

2.4 Intrinsic Typing

The Haskell-like language we create in this paper is intrinsically typed as described in the *deBruijn* chapter of the *Programming Language Foundations in Agda*[8] book. This means that all of our language terms will have a type associated with them. As an example, for our addition operator, we have to specify that it will return a number type. We use this approach because it gives us guarantees about the well-typedness of any expression in our language as it is impossible to represent non-well-typed terms in an intrinsically typed language.

3 Defining the Language

In this section, we will go over the process of defining our language constructs and their behaviour. First, we will cover what constructs we decided to include and why those decisions were made. After that, we will cover the definition of the semantics of our language.

For the definition of our language and its behaviour, we made heavy use of the book *Programming Language Foundations in Agda*[8].

3.1 Constructs

We decided to include three base types in our language. These are natural numbers (hereafter referred to as just numbers), booleans, and functions. We decided to include both numbers and booleans to allow the possibility for our refactoring to create arguments of the wrong type in case it was written wrong. The inclusion of functions was an obvious requirement for our language since we are trying to create a refactoring that deals with functions.

Other than constructors for these base types we also include some operations on these types. For numbers, we have implemented addition, multiplication, and less-than operations. For booleans, we decided on implementing and, or, and negation.

The constructs in our language are defined as a data type in Agda. Each construct takes a function context (named Δ), a variable context (named Γ), and a returned type. Different constructors of this type then take their own arguments and return this type. For example the constructor for the addition (plus) language construct in listing 2. It takes two other language constructs that return a number and returns a language construct that also returns a number.

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

Listing 2: Agda definition of the plus constructor for our language.

The number and boolean constructors simply take their respective Agda types and return constructs in our language. An example definition of the number type can be seen in listing 3.

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

Listing 3: Agda definition of the natural number constructor for our language.

The multiplication, and, or, and negation constructors follow a similar pattern as the addition constructor in listing 2. The only difference is that the boolean operators work with booleans instead of numbers. The less-than constructor is slightly more complex taking two numbers and returning a boolean as seen in listing 4.

```
lt : δ × γ ⊢ tyNat
   → δ × γ ⊢ tyNat
   --------------
   → δ × γ ⊢ tyBool
```

Listing 4: Agda definition of the less than constructor for our language.

As stated earlier our language also includes constructs for function definition. We do this in the form of a lambda construct as given in listing 5. The argument it takes is a language construct that returns a type B if we append a variable of type A to the variable context. The variable of type A is the function argument in this case and the given construct is the function body. We can use this to create a multi-argument function by simply nesting lambda expressions until we have the desired number of arguments.

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

Listing 5: Agda definition of the lambda constructor for our language.

To retrieve values from the variable context we have the var construct. The constructor (given in listing 6) takes a lookup proof for the variable in the variable context.

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

Listing 6: Agda definition of the var constructor for our language.

We also need the application of functions as defined in listing 7. Function application takes a construct that returns a function from type A to type B and a construct that returns something of type A. It then returns the value of type B that is obtained by evaluating the function with the given argument.

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

Listing 7: Agda definition of the function application constructor for our language.

To store a function in the function context we can use the *fdef* construct. It takes a construct that returns a function and a construct that given that a function of this type is added to the function context returns a type C. It then evaluates this second construct and returns its returned value. Its constructor can be seen in listing 8.

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

Listing 8: Agda definition of the *fdef* constructor for our language.

To retrieve a value from the function context we can use the *fvar* construct. This construct works in much the same way as the *var* construct, but instead of taking a value from the variable context, it takes a function from the function context.

Finally, there is the *newCtx* construct. This construct simply takes a construct and evaluates it with empty function and variable contexts. This is used by the refactoring to place the default value of the argument in later since the context it can be placed in could be vastly different depending on where in the program it is used.

To help get a better grasp of the language we will go through a small example program defined in listing 9. First, we use an *fdef* constructor to place a function into the function context. This function is defined by the *lambda* constructor with a body that takes the first argument (signified by the *# 0*) and the number 1 and adds them together. The rest of the program then uses an *appl* constructor to apply the first function in the function context to the number 41.

```
(fdef
    (lambda
        (plus (# 0) (nat 1))
    )
    (appl (fvar Zᶠ) (nat 41))
)
```

Listing 9: Example program in our language that defines a function to add one to a given number and calls that function.

3.2 Semantics

Our language uses big-step semantics to define its behaviour. The semantics define rules we can use to determine what value our program will take. Since we know that our program will always terminate (it does not contain looping or recursion), we can chain these rules together to create a complete evaluation of our program from start to finish.

For these semantics to make sense we have to first define what values our program can take. To this end we define three different types of values as seen in listing 10. We also associate each value with a matching type from the previous subsection.

```
data Val : Type → Set where
    natV  : ℕ → Val tyNat
    boolV : Bool → Val tyBool
    funcV : Env γ
          → FEnv δ
          → δ × γ , argT ⊢ retT
          → Val (argT ⇒ retT)
```

Listing 10: Agda definition of the values expressions in our language can evaluate to.

The semantics for most of our constructs are quite simple. They tend to take the same shape as the constructors defined earlier. Take, for example, the addition operator (from listing 2) and its semantics (given in listing 11). Instead of taking two constructs we now take two reductions to numbers and instead of returning a number type we now return a concrete number value based on the two earlier reductions.

```
↓plus : ∀ {v₁ v₂} {e₁ e₂ : δ × γ ⊢ tyNat}
      → Δ × Γ ⊢ e₁ ↓ (natV v₁)
      → Δ × Γ ⊢ e₂ ↓ (natV v₂)
      → Δ × Γ ⊢ (plus e₁ e₂) ↓ (natV (v₁ + v₂))
```

Listing 11: Agda definition of the semantics of the addition operator in our language.

Most other constructs follow a very similar pattern to the one above. There are, however, a couple of exceptions. Namely, the reductions for function definition, application, and variable lookup are a bit more interesting.

The reduction for function definitions (or lambdas) is fairly straightforward as can be seen in listing 12. It takes the current environments and the given function body and stores them into a *funcV* or closure for use later.

```
↓lambda : ∀ {e : δ × γ , A ⊢ B}
        → Δ × Γ ⊢ (lambda e) ↓ (funcV Γ Δ e)
```

Listing 12: Agda definition of the semantics of function definition in our language.

The *var* and *fvar* constructs are also simpler than their constructor counterparts. They simply use the already given lookup proof to retrieve a variable from their respective environments. As an example, the semantics for *var* are given in listing 13.

```
↓var : ∀ {l : γ ∋ A}
     → Δ × Γ ⊢ (var l) ↓ (env-lookup Γ l)
```

Listing 13: Agda definition of the semantics of *var* in our language.

The semantics for function application are too cumbersome to include in full, but in essence, it takes care of the evaluation of the function provided to it in full next to also resolving all of its normal arguments.

We will continue building on the example provided in the last section in listing 9 and provide the semantic evaluation of the program in listing 14. As stated earlier our semantics follow mostly the same structure as the constructors in the original program. First, we reduce the *fdef* by providing it with a reduction for both the function definition side (in this case a simple *lambda* reduction) and a reduction for the rest of the program. The function application reduction takes a reduction for its first and second arguments (the *fvar* and *nat* reductions) and a reduction for the evaluation of the body of the given function. Which in this case is an addition with reductions for its two arguments.

```
↓fdef
    ↓lambda
    (↓appl
        ↓fvar
        ↓nat
        (↓plus ↓var ↓nat)
    )
```

Listing 14: Semantic evaluation of the example program given in listing 9.

The *fixFVar* function adds a function application with the default argument around a *fvar* construct if it refers to the function being refactored. If it refers to any other function it is left untouched.

4 Creating the Refactoring

The goal of the refactoring operations we are trying to create is to add an extra argument to a function and adding a default to the call sites of the function. As an example, we want to rewrite the program from listing 15 to the program in listing 16.

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

...

add 1 2
```

Listing 15: A Haskell program defining a function add that takes two arguments.

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

...

add 1 2 0
```

Listing 16: A Haskell program defining a function add that takes three arguments of which the last one is unused.

As stated in the previous section we use a separate environment for functions (referred to as Δ). We do this because it makes the refactoring easier to implement and prove. Without the two environments we would have to also provide proof to Agda that the variable we would be modifying indeed resolves to a function value, but since the environment being targeted by the refactoring only contains functions this is always true.

Each program that is refactored should start with one or more *fdef* constructs for the refactoring to be effective. These constructs are comparable to the top-level function definitions in Haskell code. If these constructs are not present the refactoring will simply exit without performing any modifications.

Our refactoring takes the original program, the index of the function to modify, and the default expression to be inserted. Its code signature can be seen in listing 17. The implementation recursively descends the *fdef* expressions at the start of the program until it has found the target function definition. If it encounters any other expression it simply returns the original program and exits.

```
afa : δ × γ ⊢ t → ℕ → ∅ᶠ × ∅ ⊢ argT → δ × γ ⊢ t
```

Listing 17: Agda function signature of our refactoring.

When the refactoring finds the correct function definition it performs two actions. First, it wraps the function into another lambda expression and updates all references inside the function to correspond to the same values in the environment as before. After that, it updates the remainder of the program at every point the function is called to add an extra function application using the default argument provided. The code for this is provided in listing 18.

```
afa (fdef e e₁) zero default = fdef
    (lambda (fixRefs e zero))
    (fixCalls e₁ zero default)
```

Listing 18: Agda code for our refactoring in the case that it has found the function to refactor.

The *fixRefs* helper function (as seen in listing 19) recursively walks the program constructs until it finds a variable lookup and updates that variable lookup to point to the right place in the environment. To do this it keeps track of how many lambda expressions it has encountered, because it only has to change variable lookups that refer to variables defined before the function definition.

```
fixRefs : δ × γ ⊢ t → (n : ℕ)
        → δ × (insertType γ n A) ⊢ t

fixRefs (var x) n = var (fixVar x n)
fixRefs (lambda e) n = lambda (fixRefs e (suc n))
```

Listing 19: Agda code for the *fixRefs* helper function. Trivial cases omitted for brevity.

The *fixVar* helper function moves lookups to values defined outside of the function one element deeper into the environment since we have inserted a new value in between. If the lookup is to a variable defined within the function it simply returns the original lookup.

The *fixCalls* helper works on the other argument of the *fdef* construct to update all references to the function being refactored to contain the newly added argument. It works mostly the same as the *fixrefs* helper, but instead of acting on *var* and *lambda* constructs it works on *fvar* and *fdef* constructs. The code for it can be seen in listing 20.

```
fixCalls : δ × γ ⊢ t → (n : ℕ) → ∅ᶠ × ∅ ⊢ argT
         → (replaceFunction δ n argT) × γ ⊢ t

fixCalls (fdef e e₁) n default
    = fdef (fixCalls e n default)
           (fixCalls e₁ (suc n) default)

fixCalls (fvar l) n default
    = fixFvar l n default
```

Listing 20: Agda code for the *fixCalls* helper function. Trivial cases omitted for brevity.

5 Proving Correctness

To prove the correctness of our refactoring we will have to prove two separate conditions. The first is that our refactoring outputs a well-typed program and the second is that it does not change the behaviour in unexpected ways.

The first condition is easily proven in the case of our refactoring since we use an intrinsically typed language. As described earlier it is impossible to represent a non-well-typed program in such a language. This means that our refactoring program is itself a proof that it will produce a well-typed program.

The second condition of behaviour preservation is harder to prove. First, we have to define what we want our refactoring to do to the behaviour of our program. In essence, we do not want it to change at all, but this constraint is too strict for our refactoring. This is because our refactoring has to modify the environment of parts of our code. And since closure values capture their environment as part of themselves, they would not be identical to their pre-refactoring counterparts.

Because of this, we define a relation between two values that we will call *refactoring equivalence*. This relation is represented by the $\equiv v_r$ symbol. For natural numbers and booleans, it is defined as normal equality, but for closure values, it is defined such that two closure values are considered refactoring equivalent if and only if given refactoring equivalent arguments they would evaluate to refactoring equivalent values. This definition allows us to ignore the environment captured by the closures as long as we can prove that they would evaluate to refactoring equivalent values. The Agda definition of the relation can be seen in listing 21.

```
_≡vr_ : ∀ {ty} → Val ty → Val ty → Set
natV xₒ ≡vr natV xₙ = xₒ ≡ xₙ
boolV xₒ ≡vr boolV xₙ = xₒ ≡ xₙ
funcV Γₒ Δₒ bₒ ≡vr funcV Γₙ Δₙ bₙ =
    ∀ {argVₒ : Val argTy} {argVₙ : Val argTy}
      {argVₒ≡vrargVₙ : argVₒ ≡vr argVₙ}
      {retVₒ : Val retTy} {retVₙ : Val retTy}
    → Δₒ × (Γₒ ,' argVₒ) ⊢ bₒ ↓ retVₒ
    → Δₙ × (Γₙ ,' argVₙ) ⊢ bₙ ↓ retVₙ
    → retVₒ ≡vr retVₙ
```

Listing 21: Agda definition of the refactoring equivalence relation.

Using this definition of refactoring equivalence we will also define the notion of *refactoring equivalent environments*. These are a list of refactoring equivalence proofs that can be used to construct an environment with different, but refactoring equivalent, values. They are defined for both normal and function environments, but their definitions are almost identical so in the interest of brevity we will only show the one for normal environments in listing 22.

```
data EquivEnv : Env γ → Set where
    ee-root : EquivEnv ∅'
    ee-elem : ∀ {vₒ vₙ : Val t}
            → EquivEnv Γ
            → (vₒ≡vrvₙ : vₒ ≡vr vₙ)
            → EquivEnv (Γ ,' vₒ)
```

Listing 22: Agda definition of refactoring equivalent environments defined as a data type.

For each type of environment, we also define a helper that takes a refactoring equivalent environment with a lookup into the environment and returns the proof of refactoring equiv-

alence of the values stored at the lookups location. These helpers are called `var≡v`$_r$ and `fvar≡v`$_r$ for normal and function environments respectively.

Using these definitions we can construct a proof that the same expression evaluated in two refactoring equivalent environments will produce refactoring equivalent values. The signature of this proof can be seen in listing 23.

```
equiv-env≡ : ∀ {v_o v_n : Val t} {e : δ × γ ⊢ t}
           → (efe : EquivFEnv Δ)
           → (ee : EquivEnv Γ)
           → Δ × Γ ⊢ e ↓ v_o
           → (constructEquivFEnv efe)
             × (constructEquivEnv ee) ⊢ e ↓ v_n
           → v_o ≡v_r v_n
```

Listing 23: Signature of the Agda function that produces a proof that the same expression evaluated in refactoring equivalent environments produce refactoring equivalent values.

The above proof is split on the different types of expressions that exist in our language. The natural number and boolean expressions are trivially proven because our function signature states that the expression does not change and Agda can understand that because of that the values produced by them also does not change. For operators such as addition, multiplication, etc it is slightly more complicated. For these, we use a helper function in the form shown in listing 24 (shown is the one for addition). Using the helper function we provide a proof that $i+j = h+k$ if $i = h$ and $j = k$. The other helper functions follow the same pattern, just replacing the addition operator with others such as multiplication. For lambda expressions, we return a function that takes a proof that the arguments are refactoring equivalent and the evaluation of the old and new bodies (listing 25). For function application expressions we then call this returned function with a proof that the arguments are refactoring equivalent and the evaluation of the bodies to produce a proof that the application of the functions is indeed refactoring equivalent (listing 26). The argument refactoring equivalence proof is provided by a recursive call to the `equiv-env≡` function.

```
i+j≡h+k : ∀ {i j h k} → i ≡ h → j ≡ k
        → i + j ≡ h + k
i+j≡h+k refl refl = refl
```

Listing 24: Helper function used to prove that the equality of values implies that their addition is also equal.

```
equiv-env≡ efe ee ↓lambda ↓lambda
    {argV_o≡v_rargV_n = argV_o≡v_rargV_n} ↓b_o ↓b_n
        = equiv-env≡ efe
          (ee-elem ee argV_o≡v_rargV_n) ↓b_o ↓b_n
```

Listing 25: Case of the `equiv-env≡` function that handles lambda expressions.

```
equiv-env≡ efe ee (↓appl e_o e_o1 e_o2)
    (↓appl e_n e_n1 e_n2)
    = equiv-env≡ efe ee e_o e_n
      {argV_o≡v_rargV_n = equiv-env≡
        efe ee e_o1 e_n1}
      e_o2 e_n2
```

Listing 26: Case of the `equiv-env≡` function that handles function application expressions.

The `constructEquivFEnv` and `constructEquivEnv` helpers used above create the equivalent function and normal environments respectively. They allow us to express to Agda that an expression will be evaluated under refactoring equivalent environments. This works by extracting the right hand side values of each refactoring equivalence relation stored in the constructs to build up the new environment recursively.

We use these same constructs to define a function to prove the refactoring equivalence of the values pre and post our refactoring function (listing 27). This function mostly calls the earlier defined `equiv-env≡` except for when the expression is of the type *fdef*. This is because our refactoring does not touch any expression except for *fdef* at the top level.

```
correct-afa : ∀ {Δ : FEnv δ} {e : δ × ∅ ⊢ t}
                {v_o v_n : Val t}
            → (efe : EquivFEnv Δ)
            → (ee : EquivEnv ∅')
            → Δ × ∅' ⊢ e ↓ v_o
            → (constructEquivFEnv efe)
              × (constructEquivEnv ee)
              ⊢ (afa e n default) ↓ v_n
            → v_o ≡v_r v_n
correct-afa {n = zero} efe ee
    (↓fdef e_o e_o1) (↓fdef ↓lam@↓lambda e_n)
        = correct-fix-calls ee
          (efer-repl efe ee e_o ↓lam) e_o1 e_n

correct-afa {n = suc n} efe ee
    (↓fdef e_o e_o1) (↓fdef e_n e_n1)
        = correct-afa (efe-elem efe
          (equiv-env≡ efe ee e_o e_n)) ee e_o1 e_n1
```

Listing 27: Signature and non-trivial cases of the Agda function that produces a proof that the pre and post-refactoring expressions, evaluated in refactoring equivalent environments, produce refactoring equivalent values.

Just as our refactoring program, the proof splits into two cases when it comes to *fdef* expressions. One case for $n = zero$ and one for $n = sucm$. The latter simply does a recursive call to the `correct-afa` function. In the zero case we call another helper function called `correct-fix-calls` with a modified version of our `EquivFEnv` to also include that we have replaced a function of type $A \Rightarrow B$ with one of type $C \Rightarrow A \Rightarrow B$.

This `correct-fix-calls` (Agda code in listing 28) helper gets its name from the `fixCalls` helper used by the refactoring. As an astute reader might have guessed from

the name correct-fix-calls is the behaviour preservation proof for fixCalls. Just as the other proof helpers, it is mostly the same as equiv-env≡. The exception to this is the code that handles the *fvar* expressions. This calls the correct-fix-fvar helper function.

```
correct-fix-calls : ∀ {Δ : FEnv δ} {Γ : Env γ}
    {e : δ × γ ⊢ t} {v_o v_n : Val t}
          → (ee : EquivEnv Γ)
          → (efer : EquivFEnvReplaced Δ n argT)
          → Δ × Γ ⊢ e ↓ v_o
          → (constructEquivFEnvReplaced efer)
            × (constructEquivEnv ee)
            ⊢ (fixCalls e n default) ↓ v_n
          → v_o ≡v_r v_n

correct-fix-calls ee efer e_o@↓fvar e_n
    = correct-fix-fvar ee efer e_o e_n
```

Listing 28: Signature and the non-trivial case of the Agda function that produces a proof that the pre and post-refactoring expressions, in refactoring equivalent environments, produce refactoring equivalent values (fixCalls helper function).

The correct-fix-fvar helper continues the trend of being the behaviour preservation proof of the similarly named fixFvar. It follows the same pattern as the fixFvar of using a with abstraction to split the possible cases into two groups with a proof that a certain case belongs to that group. The first group is where the lookup is referring to the refactored function. This case is delegated to the correct-fix-lookup^y. The rest of the cases are delegated to the correct-fix-lookup^n.

First, we will cover the correct-fix-lookup^n helper. It uses the proof generated in the correct-fix-fvar function to recurse down on the lookup into the replaced refactoring equivalent function environment until it either hits $n = zero$ or runs out of replaced function environment. When it hits $zero$ it uses the proof stored in the refactoring equivalent function environment as a proof that both lookups produce refactoring equivalent values. If it does not hit $zero$ it will use the fvar≡v$_r$ helper to provide the proof that both lookups provide refactoring equivalent values.

The correct-fix-lookup^y helper is used in the case that the lookup does point to the refactored function. In this case, we have to prove that a lookup into the old function environment is refactoring equivalent to the following expression in the new function environment: (appl (fvar l) (newCtx default)). This new expression wraps the lookup with an application to provide the default value for our new function argument. To prove this equivalence we need to prove that the evaluation of the returned clojures is still equivalent. To this end, we use the correct-fix-refs helper function.

Within our new function, we have a new value bound into the environment (the new function argument). Because of the extra function argument, some variables will be moved over by one in the environment. This is represented by a modified version of EquivEnv named EquivEnvPadded. The name is chosen because the new environment has one entry of "padding" in it compared to the original, but other than that all values are still refactoring equivalent. As long as the "padded" value is never accessed this will not cause issues.

The correct-fix-refs helper proves that the fixRefs function correctly updated the references within the body of the function to reference the same values in the new environment. This function is almost identical to the equiv-env≡ helper, except for the case involving *var* expressions. In this case, we use one last helper function called correct-fix-var to provide a proof that the two variables are equivalent. The code for the correct-fix-var helper can be found in listing 29.

```
correct-fix-var : ∀ {Γ : Env γ}
    → (l : γ ∋ t)
    → (n : ℕ)
    → (eep : EquivEnvPadded Γ n A)
    → (env-lookup Γ l) ≡v_r (env-lookup
        (constructEquivEnvPadded eep)
        (fixVar l n))
correct-fix-var l zero (eep-pad ee v_n)
    = var≡v_r ee l
correct-fix-var Z (suc n) (eep-elem eep v_o≡v_r v_n)
    = v_o≡v_r v_n
correct-fix-var (S l) (suc n)
    (eep-elem eep v_o≡v_r v_n)
    = correct-fix-var l n eep
```

Listing 29: Agda function that produces a proof that the variable lookups within the body of the refactored function point to refactoring equivalent values pre and post-refactoring.

6 Responsible Research

Our work should be easy to reproduce from the source code provided at our GitHub repository[3]. The source code in the repository is provided under the MIT License. The repository also contains documentation that explains how to run the code and what every part of it does. Together this allows any other researchers to reproduce, and build on top of, this work.

7 Discussion

Our provided proofs contain some potential issues that could stop them from generalizing. For one, our language is a very small subset of what most functional programming languages offer. It has no user-defined types, no branching constructs, and most importantly, it is not capable of recursion. This raises potential issues as we do not know how our refactoring will be affected by the introduction of these constructs. It might be that our proofs are trivially adapted, or they could completely fall apart.

A potential point of contention is our definition of *refactoring equivalence*. An argument could be made that this notion of equality is not strong enough and would leave room for unintended side effects of the refactoring, but since closures capture their environments we can not simply use proper equality between values.

[3] https://github.com/MetaBorgCube/brp-agda-refactoring-khstruik

Another potential issue is that our proofs do not prove what we say they do. This is partly mitigated by the use of the Agda proof assistant, but it could still be possible, albeit unlikely, that there is an issue within Agda itself allowing us to prove something that is false. There could also be subtle issues with the way our goals have been translated into Agda code, such as the above *refactoring equivalence*. This would lead to us proving a different constraint than what has been described in this paper.

On a positive note, during the process of writing our language and the refactoring operation, we learned that intrinsically typed language has some big advantages during the creation of the refactoring tool. Initially, we started with an extrinsically typed language, but eventually, we switched to an intrinsically typed language. This switch happened when we got stuck with writing our refactoring. It turns out that since our refactoring is now also a proof of well-typedness, Agda can give a lot more assistance during the creation of the refactoring itself. Agda was able to reject wrong ideas earlier than before and in that way steered us in the direction of a correct refactoring.

8 Related Work

While some work has been done on the formal verification of refactoring operations, the use of computer-aided proving techniques is somewhat rare. We would argue that computer-aided proofs are a better fit for the formal verification of refactoring operations, since could easily be kept next to the refactoring program itself ensuring that it never goes out of date as changes are made to the refactoring program itself.

The same refactoring as discussed in this paper is also present in the work by Huiqing Li and Simon Thompson[5], but in their later formalization of some of these refactoring operations it is excluded[6]. On top of that, they choose to use pen and paper proofs instead of computer-aided proofs as described in this paper.

A similar method of proving correctness, as described in this paper, is used in [1] by Barwell et al. In their work, they focus on creating a renaming refactoring and proving its correctness. They find that for the correctness of a renaming refactoring, they do not need to prove behaviour equivalence so long as they can prove structural equivalence between the pre and post-refactoring program. This is distinctly different from our approach in this paper because for our refactoring it is required to prove behavioural equivalence.

The work by Nik Sultana and Simon Thompson in [7] again follows a similar approach as we described in this paper, except that they are working with source-to-source refactoring operations. While in this paper we decided to not worry about the textual representation of the code being refactored, they chose to also verify that their refactoring did not make unnecessary changes to the textual representation. They constructed proofs for many different refactoring operations, but adding function arguments is notably absent from their work.

9 Conclusions and Future Work

In conclusion, we have created a Haskell-like language with corresponding big-step semantics and a refactoring operation that adds an argument to top-level functions in this language. We have also been able to use Agda to construct a proof of behaviour preservation. Since our language is intrinsically typed, the refactoring also serves as a proof of well-typedness.

This, combined with the work done by others in this space, shows that it is feasible to construct refactoring operations and prove their correctness when applied to functional languages using computer-aided proofs.

There are plenty of ideas left worth exploring related to this work. One such idea is to expand our language with some important missing constructs such as branching and recursion and to expand our proofs to fit these new conditions. Another idea to explore would be to take the work done by the rest of the group and combining these different refactoring operations into one tool.

References

[1] Adam David Barwell, Christopher Mark Brown, and Susmit Sarkar. Proving renaming for haskell via dependent types: a case-study in refactoring soundness. In *8th International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2021)*, 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.

[3] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic testing of refactoring engines on real software projects. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, pages 629–653, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Gilles Kahn. Natural semantics. In *STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science Passau, Federal Republic of Germany, February 19–21, 1987 Proceedings 4*, pages 22–39. Springer, 1987.

[5] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, New York, NY, USA, 2008. Association for Computing Machinery.

[6] Huiqing Li and Simon J Thompson. Formalisation of haskell refactorings. *Trends in Functional Programming*, pages 95–110, 2005.

[7] Nik Sultana and Simon Thompson. Mechanical verification of refactorings. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 51–60, 2008.

[8] Philip Wadler, Wen Kokke, and Jeremy G. Siek. Programming language foundations in Agda, August 2022.