Formally proving the correctness of the (un)currying refactoring

Using Agda with a simple Haskell-like programming language

Michał Jóźwik[1]

Supervisor(s): 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 25, 2023

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

Abstract

When designing critical software, great care must be taken to guarantee its correctness. Refactoring is one of the techniques used to improve code readability, maintainability, and other factors without changing functionality. Thus, to ensure that it is properly applied, automated tools are used to perform refactoring. To ensure that the code remained correct, we verified the correctness of the tool actions using formal proof. This is assisted by a dependently typed language Agda, which, when used as a proof assistant, can directly support us by ensuring the soundness of the steps taken. We consider the refactoring of currying and its counterpart, uncurrying, using a small proof-of-concept functional programming language with intrinsically typed terms and de Bruijn indices. Furthermore, we proved the retained properties of well-typedness, termination, and value relation of the input program. Finally, we argue that this research could be extended to the full implementation of Haskell, as well as applied to other languages and similar refactorings.

1 Introduction

The development of secure, performant, and readable code is a challenging problem in computer science. Every day, we use more code for further aspects of our daily lives. The continuous development of critical systems makes it necessary to be able to efficiently check the correctness of their functionality. As these systems grow, changes are necessary and more cognitive work is required to retain the validity of the design. To aid with that, automated refactoring tools are used to optimize the code structure. However, most developers are still reluctant to use these tools because they do not trust these guarantees of safety [1].

The motivation behind this study is to develop a tool for automated refactoring and to prove its correctness. We want to contribute to building trust in developers so that they can use these refactoring tools without any issues. Second, we want to further demonstrate the importance of formal methods, which directly allow us to verify the correctness of the code with regard to the design and find logic bugs early in the coding process, leading to reliable and safe software. This has also been shown in action in real industrial processes [3], as such the demand for such tools still exists in the industry [5].

The primary focus of this project is to prove the correctness of code refactoring. Specifically, we aim to address the operation of currying, which involves transforming a function that takes multiple arguments into a sequence of functions, each with a single argument (see Listing 1). Additionally, we investigate its inverse operation known as uncurrying.

Our objective is to formally prove that the refactoring process has no unforeseen side effects, ensuring that the code retains its original functionality. To achieve this, we will employ Agda, a functional and dependently typed language known for its ability to serve as a proof assistant, allowing for the translation of mathematical proofs into computer programs and vice versa. The use of this assistant makes it easier and more efficient compared to manual methods, as it can directly accompany the development process, verify that the steps taken follow the rules of formal logic, and support the automated completion of parts of the proof.

This study explores an important topic that allows us to improve code readability and performance while ensuring its functional correctness. Although the topic of formal correctness (see Section 8) and the usage of Agda for proving program properties have been extensively researched, the refactoring of programs in functional languages, unlike object-oriented languages such as Java, has received less attention [9; 13]. Existing work provides valuable insights and an overview of the topic's usefulness. However, there is a lack of in-depth research on the formal proofs of refactoring operations, particularly focusing on currying and uncurrying operations.

```
1  -- Original function
2  f1 : (Int, Int) → Int
3  f1 (x, y) = x + y
4
5  -- Function after currying
6  f2 : Int → Int → Int
7  f2 = λ x → (λ y → x + y)
```

Listing 1: Example of currying and uncurrying (transformation back from f2 to f1) operation.

Therefore, the main objective of this study is to *investigate the feasibility of formal proof methods to check the correctness of the currying (and uncurrying) refactoring in a simple Haskell-like language*[1] *using Agda.*

We make the following contributions, to explore this problem:

- We create an intrinsically well-typed language, with de Bruijn indices, that mimics a small subset of Haskell's functionality.
- A set of big-step semantics rules is provided to show the relation between the terms and the output values.
- The definitions of the functions for performing currying and uncurrying transformation are given.
- We prove the retained well-typedness and termination, and value relation of the refactoring.

In this research paper, firstly in Section 2, the background of this research and all the important information is provided. In Section 3, we describe the development of the special programming language used for this research. The main proof and its properties will be explained in Section 4. Limitations follow this in Section 5. Furthermore, Section 6 reflects on the ethics and overall reproducibility of the research outcomes, followed by a general discussion of this research in Section 7. Lastly, we consider the related work in Section

[1] The generalization of this choice is discussed in Section 5.1

8 and finally derive conclusions about this work and identify potential future steps in Section 9.

2 Background

This section describes all the relevant information that might be useful or required to properly understand the content of this study. We provide detailed information about the refactoring, and Agda programming language and give general advice on the development process.

2.1 Refactoring

The process of refactoring is one of the main subjects of this study and is a well-known topic in literature. The term "refactoring" was introduced in the 1990s by Smalltalk[2] programmers, however, one of the more well-established definitions was provided by Martin Fowler, who defines it as "A series of small steps, each of which changes the program's internal structure without changing its external behavior" [6].

According to Fowler, the general benefits of these changes include better software maintainability, simpler code, the easier process of finding bugs, and the removal of code smell. Furthermore, he provides guidelines for refactoring tools. The most relevant rule for this research is that tools should be based on provable transformations with mathematical proof that the semantics do not change. In practice, these guidelines are often not adhered to; thus, some automated refactoring may introduce unforeseen changes that create bugs in the software [7]. In other cases, refactoring may not be formally proven; thus, program testing is still an important and relevant part of the process [6].

2.2 Agda Programming Language

Agda is a functional programming language with a dependent type system. It is based on the type theory, which concerns both programming and logic, to express syntactic correctness. For this reason, it is a total language, which means that if any program with type T is executed, then it will always result in the value of type T. This program must terminate in finite time, and no runtime error can occur unless specified explicitly. The main reason for using Agda in this project is that it can be used as a proof assistant and allows one to ensure the correctness of specific programs at compile time. This is possible because of the Curry-Howard correspondence, which shows a direct relationship between the logical specification and the dependent types [11].

Dependent types

The main feature of Agda used in this paper is dependent types. They are introduced by having hierarchies of types indexed by objects of another type. An often-used example is of a `Vec A n` type (see Listing 2).

In this case, the `Vec A n` type corresponds to a family of vectors of a given type A with an explicitly given size n. This allows us to better formalize the constraints required by the program to avoid runtime errors. This exact dependent type

```
1 -- Vec type definition
2 data Vec (A : Set) : Nat -> Set where
3     []   : Vec a~zero
4     _::_ : {n : Nat}
5          -> a~
6          -> Vec a~n
7          -> Vec a~(suc n)
```

Listing 2: Example of a dependent type definition.

can be interpreted as a list of a given length. The `[]` constructor is used to create an empty `Vec` of a given type and size of 0. The other constructor is interpreted as a prepend operator[3], where an element of type A is provided on the left and the existing `Vec` of the same type. This results in a new `Vec` that now includes the new element, and its type directly represents the updated size, which is a successor of the previous size.

2.3 Development process

Most of the fundamental knowledge of this study has been taken from the PLFA book (see [12]). It goes deeply into the usage of Agda, from simple to very advanced tasks. It has been used as a guide throughout development and as the main source of code examples. Within the book content, there have also been countless embedded exercises, which allow for further development of a better understanding of the formal methods. For a keen reader who would like to start research in a given area, it is definitely necessary to read. To further aid in obtaining a proper introduction to the field of programming languages, we received additional exercise material directly related to the creation of a small functional language, which was later used as a base for the HLL, as described in the following section.

3 Programming language development

To perform refactoring on a program, we have to begin with a programming language to perform this action. The main research area of this paper is the formal proofs of refactoring; thus, we do not require the language to have a separate parser, as we can only focus on the abstract syntax tree of this language. With this approach, we avoid working on irrelevant elements for this research and focus more on the main subject of this study.

The language created during this research resembles the syntax of other functional programming languages, particularly Haskell. Throughout this paper, we refer to this created language as Haskell-Like Language (HLL). The capabilities of the possible programs are much less extensive, as the implementation of all features is not feasible within the research period. However, all the relevant parts of the refactoring were implemented and are described in the following subsections.

[2]Smalltalk is an Object Oriented Language created in 1970s mostly for educational purposes, which later inspired new programming languages such as Java.

[3]Underscores can be interpreted as places for arguments. Agda calls these mixfix operators - they allow for a more convenient syntax that allows mixing arguments within the defined syntax.

3.1 Type system

To start with the HLL definition we have to define the available types (`Ty`) as well as type context (`Ctx`) explained in detail further in the text (see Listing 3).

```
data Ty : Set where
  ...
  numT  : Ty                      -- Natural number
  _⇒_   : Ty → Ty → Ty            -- 1-arg functions
  /_/_⇒_ : Ty → Ty → Ty → Ty      -- 2-arg functions

Ctx = List Ty
```

Listing 3: The definition of the main types and type context used in the HLL.

The HLL uses intrinsic typing for its syntax. It works in such a way that only well-typed programs can be expressed, which makes it much easier to show the preservation of type after refactoring. We do not have to write separate type-checking proofs, as they have already been checked by Agda.

This typing system is based directly on the notion of inference rules, which, given a list of premises, draw a conclusion. An example of this would be the following derivation: given two expressions n_1 and n_2 in context Γ of a type $numT$, the result of add to these two expressions has a type $numT$ as well. This can be formally written, as shown in Figure 1.

$$\frac{\Gamma \vdash n_1 : numT \quad [\mathbb{N}] \qquad \Gamma \vdash n_2 : numT \quad [\mathbb{N}]}{\Gamma \vdash (\texttt{add}\ \ n_1\ \ n_2) : numT}$$

Figure 1: Inference rules for the add instruction.

The previously described inference rules can be directly translated to Agda to define HLL syntax as shown in Listing 4.

```
-- Following implicit arguments used on the code
↪  have types as defined after the colon
private
  variable
    Γ : Ctx

data _⊢_ : Ctx → Ty → Set where
  num : ℕ
      → Γ ⊢ numT
  add : Γ ⊢ numT
      → Γ ⊢ numT
      → Γ ⊢ numT
```

Listing 4: Example of syntax definition using intrinsic typing.

To further get to the point where currying refactoring is possible, we need to define the language constructs for anonymous functions. With the current approach, we mostly require functions that accept a single argument and two arguments to perform the currying as shown in the example provided in Listing 1. In Section 5, we define the generalization for multi-argument functions with proper reasoning. Section 7.1 describes the use of special constructs instead of regular tuples.

Currently, the most important parts include function definitions, applications, and variables, which can be directly seen in Listing 5.

```
private
  variable
    t u v : Ty
    Γ : Ctx

data _⊢_ : Ctx → Ty → Set where
  ...
  -- Variable lookup
  var  : t ∈ Γ
       → Γ ⊢ t
  -- Single argument function
  fun  : (t :: Γ) ⊢ u
       → Γ ⊢ (t ⇒ u)
  -- Two argument function
  fun₂ : (t :: u :: Γ) ⊢ v
       → Γ ⊢ (/ t / u ⇒ v)
  -- Single argument function application
  app  : Γ ⊢ (t ⇒ u)
       → Γ ⊢ t
       → Γ ⊢ u
  -- Two argument function application
  app₂ : Γ ⊢ (/ t / u ⇒ v)
       → Γ ⊢ t
       → Γ ⊢ u
       → Γ ⊢ v
```

Listing 5: Syntax definition of functions in HLL.

3.2 De Bruijn indices

In the previous subsection, as one of the language constructs, we used a variable lookup requiring $t \in \Gamma$ as a premise for its resulting type. In fact, it does not use a name to identify variables but uses a concept known as the *de Bruijn indices* [4]. They are represented similarly to natural numbers, with the most recently bound variable and its successor (see the definition in Listing 6). Thus, to perform a variable lookup, we must provide a witness of the existence of the provided type in the type context. The definition used here is based on the Chapter "DeBruijn" from the PLFA book [12].

```
data _∈_ : Ty → Ctx → Set where
  here  : ∀ {a as} → a ∈ (a :: as)
  there : ∀ {a₁ a₂ as}
        → (a₁ ∈ as)
        → a₁ ∈ (a₂ :: as)
```

Listing 6: Definition of the de Bruijn lookup.

3.3 Big-step semantics

The next important step when defining a language is the actual evaluation of the given program. We would want to end up with a useful result after performing each instruction in our code. In this case, the list of the possible output values has to be defined, which for the HLL case can be seen in Listing 7. Furthermore, to be able to evaluate functions properly, we need to use the environment (`Env`), which stores a corresponding value to each element in the type context, which can be later used to evaluate functions.

```
private
  variable
    t u v : Ty
    Γ : Ctx
    γ : Env Γ

data Env : Ctx → Set
  []  : Env []
  _::_ : Val t → Env Γ → Env (t :: Γ)

lookup-val : Env Γ → t ∈ Γ → Val t
lookup-val (v :: γ) here = v
lookup-val (v :: γ) (there x) = lookup-val γ x

data Val : Ty → Set where
  ...
  numV   : ℕ → Val
  closV  : Env Γ → (t :: Γ) ⊢ u → Val
  closV₂ : Env Γ → (t :: u :: Γ) ⊢ v → Val
```

Listing 7: Definition of all the available values in HLL, which represent the possible outputs of the program. It also defines the environment of the bound variables, as well as the function used to lookup a given variable based on the witness from the previous chapter.

Finally, to obtain the result of the given program, we use *big-step semantics*, which is a relation between the input terms and output values [12]. Suppose we have a term P that evaluates to a value V in the given environment γ, which is represented by $\gamma \vdash P \Downarrow V$. We must define all these relations for each possible instruction in our program to represent all the possible outcomes that could be calculated using the provided syntax, which for the HLL is shown in Listing 8. This formalization uses a call-by-value strategy as it directly evaluates all arguments, even in cases where they might not be directly needed. This is a different approach from that of Haskell, which uses a call-by-need[4] strategy.

4 Refactoring proof

After defining the language syntax and its semantics, we can continue with the main subject of the paper, which is about performing the actual refactoring.

4.1 Refactoring definition

The definition of the refactoring functions is provided in Listing 9. It uses a simple approach of pattern matching on all

[4]https://wiki.haskell.org/Lazy_evaluation

```
private
  variable
    Γ Δ : Ctx
    γ : Env Γ
    δ : Env Δ
    n : ℕ
    v1 : Val t
    v2 : Val u
    t u v : Ty

data _⊢_⇓_ : Env Γ → Γ ⊢ t → Val t → Set where
  ...
  ⇓var  : (x : t ∈ Γ)
        → γ ⊢ (var x) ⇓ lookup-val γ x
  ⇓num  :
        γ ⊢ (num n) ⇓ (numV n)
  ⇓add  : ∀ {e1 e2 n1 n2}
        → γ ⊢ e1 ⇓ (numV n1)
        → γ ⊢ e2 ⇓ (numV n2)
        → γ ⊢ (add e1 e2) ⇓ numV (n1 + n2)
  ⇓fun  : ∀ {b}
        → γ ⊢ (fun {t} {Γ} {u} b) ⇓ (closV γ b)
  ⇓fun₂ : ∀ {b}
  → γ ⊢ (fun₂ {t} {u} {Γ} {v} b) ⇓ (closV₂ γ b)
  ⇓app  : ∀ {f : Γ ⊢ t ⇒ u}
            {b : (t :: Δ) ⊢ u}
            {arg}
        → γ ⊢ f ⇓ (closV δ b)
        → γ ⊢ arg ⇓ v1
        → (v1 :: δ) ⊢ b ⇓ v2
        → γ ⊢ (app f arg) ⇓ v2
  ⇓app₂ : ∀ {f : Γ ⊢ / t / u ⇒ v}
            {b : (t :: u :: Δ) ⊢ v}
            {arg1 arg2 v3}
        → γ ⊢ f ⇓ (closV₂ δ b)
        → γ ⊢ arg1 ⇓ v1
        → γ ⊢ arg2 ⇓ v2
        → (v1 :: v2 :: δ) ⊢ b ⇓ v3
        → γ ⊢ (app₂ f arg1 arg2) ⇓ v3
```

Listing 8: Rules for evaluating all the relevant constructs in HLL.

possible constructs in the language, followed by either a recursive call to the function or by transforming the relevant construct. Some of the constructs were omitted to make the listing more concise, as they only called the function recursively. In this case, the most important construct is the function application, as when we encounter two applications following each other, we can directly translate them into a single two-argument function application. The reverse also holds: when we encounter a two-argument function application, we can unwrap it into two separate function applications. The type signature for both refactorings also provides us with a well-typedness proof, as the type of program before and after must match each other.

With this approach, there are some limitations to the actions that can be performed. In Section 5 we discuss the issue that remains with the current approach to uncurrying, as it is more restrictive in its application than currying. Another remark that could be said about refactoring is that it applies to all possible cases. Currently, owing to language limitations

and time constraints, it is not possible to perform refactoring only on a subset of selected functions; thus, it is applied to all possible locations in the provided code.

```
private
  variable
    p : Ty
    Γ : Ctx

ref-type : Ty → Ty
...
ref-type numT = numT
ref-type (t ⇒ u) =
    (ref-type t) ⇒ (ref-type u)
ref-type (/ t / u ⇒ v) =
    (ref-type u) ⇒ (ref-type t) ⇒ (ref-type v)

ref-ctx : Ctx → Ctx
ref-ctx [] = []
ref-ctx (x :: xs) = (ref-type x) :: (ref-ctx xs)

ref-curry : Γ ⊢ p → (ref-ctx Γ) ⊢ (ref-type p)
...
ref-curry (fun₂ x) = fun (fun (ref-curry x))
ref-curry (app₂ x y z) =
    app
    (app (ref-curry x) (ref-curry z))
    (ref-curry y)

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)
ref-uncurry (app f arg) =
    app (ref-uncurry f) (ref-uncurry arg)
```

Listing 9: Definition of the curry and uncurry refactoring with the most relevant parts. The missing parts include pattern matching on all other instructions from the language and recursively calling the same refactoring.

4.2 Proof of correctness

With the definition of refactoring, we can perform proofs of the properties previously mentioned in this paper. In practice, all of the work done beforehand in the previous sections when defining the language made this process much easier. Two of the properties, which are well-typedness of the program after refactoring as well as the proof of termination, have already been provided to us.

We are sure that the created programs are always well-typed because the HLL only allows well-typed expressions to be represented with their syntax. The other property of termination is ensured automatically thanks to Agda's design of being a total language, which makes it such that if the program written in the first place is accepted by Agda and definitely terminates, then the refactoring function cannot change that fact and will perform this transformation in finite time.

Finally, we must discuss the remaining properties of the retained semantics according to the specific mapping. This requires explicit proof. In Listing 10 we provide the proof for currying with the required lemma and helper functions.

```
private
  variable
    Γ : Ctx
    γ : Env Γ
    t : Ty
    v : Val t
    q : (Γ ⊢ t)

ref-env : Env Γ → Env (ref-ctx Γ)
ref-val : Val t → Val (ref-type t)

ref-env []       = []
ref-env (v :: vs) = (ref-val v) :: (ref-env vs)

-- Function which shows the mapping of values
-- after refactoring
ref-val (numV x)      = numV x
ref-val (closV x x₁)  =
    closV (ref-env x) (ref-curry x₁)
ref-val (closV₂ x x₁) =
    closV (ref-env x) (fun (ref-curry x₁))

-- Small lemma used to prove that
-- the refactored lookups stay the same
-- Used in the proof for the ⇓var
lp : (γ : Env Γ) → (x : t ∈ Γ) →
    lookup-val (ref-env γ) (ref-lookup x) ≡
↪   ref-val (lookup-val γ x)
lp (x :: xs) here = refl
lp (x :: xs) (there y) = lp xs y

curry-proof : γ ⊢ q ⇓ v
    → (ref-env γ) ⊢ (ref-curry q) ⇓ (ref-val v)
...
curry-proof {γ = γ} v@(⇓var x) =
    subst
    (λ x₁ → _ ⊢ _ ⇓ x₁)
    (lp γ x)
    (⇓var (ref-lookup x))
curry-proof ⇓fun = ⇓fun
curry-proof ⇓fun₂ = ⇓fun
curry-proof (⇓app x x₁ x₂) =
    ⇓app
    (curry-proof x)
    (curry-proof x₁)
    (curry-proof x₂)
curry-proof (⇓app₂ clos arg1 arg2 eval) =
    ⇓app
    (⇓app
        (curry-proof clos)
        (curry-proof arg2)
        ⇓fun)
    (curry-proof arg1)
    (curry-proof eval)
```

Listing 10: Definition of the semantics proof of the currying refactoring. Less relevant constructs have again been commented out, as they just included recursive calls to the proof function.

The proof is structured as a type of mapping from the input semantic steps to the refactored ones. We use the `ref-val` function to show how each value changes with refactoring. The most important one is the $\texttt{closV}_2$ value, which changes to `closV` with the modified body containing the previous function body wrapped with `fun`. Some readers might also find it interesting that this function internally uses `ref-curry`, which might lead to some confusion about it being a circular proof. However, this is required to actually work correctly as the defined closures directly take the body of the function and store it within; thus, after the refactoring, the value should also hold the refactored body.

The remaining uncurrying proof uses a similar structure (see Listing 11) to that of the previous proof. It again uses a function to show the mapping between the input and output values; however, in this case, it does not transform the closure types. This is done because of the different approach taken in the refactoring function, as we only perform the transformation when the functions are directly coupled with the application. This allowed us to change only the internal representation and retain the same output values.

With all of these assumptions taken, we still could not finish the proof in Agda because of the refactoring structure, and these limitations also made it harder to close some of the holes. The holes in the proof are indicated by $\{!n!\}$, where n is used to index them in this paper. To properly complete this formal proof, we provide a handwritten proof for the remaining holes.

First, we consider the hole with an index of 1. The pattern match of `(⇓app clos arg eval)` is supposed to capture all remaining cases of applications that were not followed by another application. It is placed below the more specific pattern, which is used to find the direct construct used in refactoring. This hole could not be trivially filled because Agda still considers the possibility of `clos` being in a form that could be refactored, even though the previous matches would already satisfy that. The most common way to solve this problem is to match all the remaining options for the `clos` to satisfy all the cases; however, Agda considers an option where the closure might be contained in a variable; thus, it cannot solve the proper constraints. We assume that, in this case, refactoring is not performed, which means that the proof should simply be called recursively without any modifications. Thus, the hole must be filled with `⇓app (uncurry-proof clos) (uncurry-proof arg) (uncurry-proof eval)`.

Second, we consider the hole with an index of 0. To formally show how this can be solved manually, we provide the following proof.

Proof. To fill in the hole with an index of 0 we need to satisfy the goal provided by the type-checker.

The labels on the left directly correspond to the variables provided in the specific case of pattern matching (see line 29 in Listing 11), whereas the expression after the label is the type signature of the given variable.

```
1  private
2    variable
3      Γ : Ctx
4      γ : Env Γ
5      t : Ty
6      v : Val t
7      q : (Γ ⊢ t)
8
9  -- Implementation the same as in the previous
   ↪  listing
10 ref-env : Env Γ → Env Γ
11 ref-val : Val t → Val t
12 ...
13 ref-val (closV x x₁)  = closV (ref-env x)
   ↪  (ref-uncurry x₁)
14 ref-val (closV₂ x x₁) = closV₂ (ref-env x)
   ↪  (ref-uncurry x₁)
15
16 uncurry-proof : γ ⊢ q ⇓ v
17     → (ref-env γ) ⊢ (ref-uncurry q) ⇓ (ref-val v)
18 ...
19 uncurry-proof ⇓fun = ⇓fun
20 uncurry-proof ⇓fun₂ = ⇓fun₂
21 uncurry-proof (⇓app (⇓app {f = var x} func arg2
   ↪  clos) arg1 eval) =
22     ⇓app
23     (⇓app
24         (uncurry-proof func)
25         (uncurry-proof arg2)
26         (uncurry-proof clos))
27     (uncurry-proof arg1)
28     (uncurry-proof eval)
29 uncurry-proof (⇓app (⇓app {f = fun (fun f)} func
   ↪  arg2 clos) arg1 eval) =
30     ⇓app₂
31     ⇓fun₂
32     (uncurry-proof arg1)
33     (uncurry-proof arg2)
34     (uncurry-proof {!0!})
35 uncurry-proof (⇓app clos arg eval) = {!1!}
36
```

Listing 11: Definition of the semantics proof of the uncurrying refactoring. Uses the same approach as the previous listing and omits the less relevant constructs.

Goal: $(v1 :: (v2 :: \gamma)) \vdash \texttt{f} \Downarrow v$

$$
\begin{aligned}
\texttt{func:}\quad & \gamma \vdash \texttt{fun (fun f)} \Downarrow \texttt{closV}\ \delta_1\ b_1 && (1)\\
& = \gamma \vdash \texttt{fun (fun f)} \Downarrow \texttt{closV}\ \gamma\ \texttt{(fun f)} && (2)\\
& \Leftrightarrow \delta_1 = \gamma;\ \ b_1 = \texttt{fun f} && (3)\\
\texttt{arg2:}\quad & \gamma \vdash arg_2 \Downarrow v2 && (4)\\
\texttt{clos:}\quad & (v2 :: \delta_1) \vdash b_1 \Downarrow \texttt{closV}\ \delta\ b && (5)\\
& = (v2 :: \gamma) \vdash \texttt{fun f} \Downarrow \texttt{closV}\ \delta\ b && (6)\\
& = (v2 :: \gamma) \vdash \texttt{fun f} \Downarrow \texttt{closV}\ (v2 :: \gamma)\ \texttt{f} && (7)\\
& \Leftrightarrow \delta = (v2 :: \gamma);\ \ b = \texttt{f} && (8)\\
\texttt{arg1:}\quad & \gamma \vdash arg_1 \Downarrow v1 && (9)\\
\texttt{eval:}\quad & (v1 :: \delta) \vdash b \Downarrow v && (10)\\
& = (v1 :: (v2 :: \gamma)) \vdash \texttt{f} \Downarrow v && (11)
\end{aligned}
$$

The step from lines 1 to 2 is valid because we directly see the `fun` that needs to be reduced. Thus, in this case, the environment in the closure is directly equivalent to the current execution environment. Similarly, we can use this knowledge to demonstrate that the body of this closure is more detailed.

In steps from lines 6 to 7 we perform the same exact reasoning as before owing to the direct existence of the `fun` construct.

$\therefore$ `eval` satisfies our goal, which we can use to fill in the hole □

Finally, with the second hole filled, we can conclude that the proof for both currying and uncurrying is finished.

5 Limitations

In this section, all possible limitations encountered during the research process are described to show the potential issues encountered and their consequences.

5.1 Generalization of results

The research conducted in this study directly proves the refactoring of a specially crafted language for this purpose. This might lead to the conclusion that this study is relevant only to this specific language. In practice, HLL has been crafted to resemble the syntax of the well-known programming language Haskell. The minimal subset presented in HLL contains all the most relevant constructs for the (un)currying refactoring, thus all the non-implemented parts should not be influenced directly by this refactoring. However, this has not been tested, as the proper extension of the HLL to resemble its parent was not possible within the research timeframe, but definitely is a valid point for future research.

5.2 Assumptions

Refactoring restrictions

Throughout this research, there have been a handful of assumptions that were made to aid the process of constructing this proof. The main issue relates directly to the subject of this research - the refactoring is done only on the two-argument functions. In practice, the extension to three or four-argument builds up using the same rule of converting the n-argument function to n subsequent calls of the given sub-functions. The ability to prove it for other exact numbers of arguments was possible, but the best approach would be to create a generalizable language construct for n-argument functions. However, the implementation of it did not fit within the provided time frame thus this argument remains.

Uncurrying limitation

This definition considers only programs that have direct function definition instructions. This implies that for the uncurrying case, the code must be directly constructed as two functions, followed by two applications, to be transformed back to the corresponding two-argument function and application. There is a possibility of constructing an example in which one of the functions is actually a variable, which after proper substitution would result in the same pattern of two functions and applications. However, to get to that point, we need to go through the actual semantics of the code, as refactoring only directly changes the instruction of the program in the static context. We have considered this edge case and determined it to be out of the scope of this project, but might be an interesting problem for any future work on this topic.

5.3 Agda as a proof-assistant

The verification of the formal proofs in this study is mostly based on Agda. This provides us with another dependency in our proof that Agda actually checks everything correctly. Its capabilities are limited mostly to the language context; thus, correctness is checked based on the Curry-Howard correspondence. This does not consider the problem that there might have been a logical error in the design itself, and as such, Agda would not help with such cases. When this process is performed for a more elaborate project, the code and design of the tool must be formally specified as well as peer-reviewed by multiple people. This would significantly decrease the possibility of general errors and make the process more robust.

6 Responsible Research

This section has examined all the considerations taken into account while doing the research to ensure that all of the core principles of good research standards have been met.

6.1 Ethics of this research

The subject discussed in this paper does not involve any ethical issues, as it only revolves around logic and mathematical methods. No external data has been used, as every result in this paper has been directly derived from the created code. Furthermore, the whole codebase is available for a public audit to be verified that it has no malicious elements.

6.2 Reproducibility

The research done in this paper had been completely documented. All of the code listings describing the valid Agda code can be directly used to prove the main subject of this paper. As mentioned previously the code has been published in the public repository on Github[5], which allows for direct code download, manual testing of the implementation, and verification of the results.

7 Discussion

This section goes more in-depth into the process and analyzes the steps taken to reach a conclusion. We reflect on the choices made during this study.

7.1 Currying definition

Some readers may find the definition of currying in this paper unusual. Instead of using tuples directly, our language presents a special construct for functions using two arguments. Using tuples, we can directly use them and nest them to have access to n-argument functions without additional language constructs. However, this caused a problem during the coding of the refactoring function. Because tuples

[5] See: https://github.com/MetaBorgCube/brp-agda-refactoring-mjozwik

are considered to be a single element in the environment, after refactoring, they are unwrapped into multiple elements, thus changing the size of the environment. This has caused problems with nested definitions. The approach with special language constructs has the advantage of retaining the environment size and order; thus, only the interpretation of the values is changed.

7.2 Use of manual proof

As one of the steps in formalizing the relationship between the retained values, we had a proof that contained two holes. The inability to finish this proof using Agda directly was most likely due to the convoluted pattern matching in the refactoring function. With some proper changes to the code structure and some other adaptations, it should be possible to conclude this proof directly in code; however, because of time constraints, this was not done. a similar problem with these kinds of holes was encountered during the development of currying and was solved by generalizing the case for currying; however, with uncurrying being more restrictive, it was not possible.

7.3 Choice of intrinsically-typed terms

During the early stages of development, a decision was made regarding the type of system design of the HLL. The choice was to employ intrinsic typing rather than having distinct terms and type rules. This, of course, affected the development of refactoring as well as the proof. The main benefit of this decision is that it has automatically provided us with a well-typedness proof of the refactored program because we can only generate well-typed programs. This has also helped identify potential issues earlier in the process. When considering these disadvantages, we must consider the extra complexity of this approach. We must follow stricter rules when constructing the proof and refactoring, which could potentially extend the time spent on the development. In practice, this approach would require the creation of an external well-typedness proof, which could be longer than the more complex approach. Therefore, intrinsic typing was selected despite its potential for additional complexity.

8 Related work

The topic of formal methods in the refactoring of functional programs has not been as widely researched as its object-oriented counterpart. There exists a knowledge gap between these different paradigms, but there are still some relevant examples of work on this subject.

One research study performed a similar case study on renaming refactoring in a subset of Haskell [2]. In their research, they were able to implement a refactoring function to rename the selected variables or functions, as well as prove the soundness of this operation, which directly relates to the work done in this study. It was similarly aided by a dependently typed language, Idris, which was used as a proof assistant.

Several studies have been conducted on the theory of refactoring operations. In [10], the authors described an approach for the formal verification of refactoring. This subject is closely related; however, their work is highly theoretical and more extensible, than the one done in this paper. One of the suggestions made by this study was the use of anonymous syntax for convenience during the refactoring process. This has been directly applied in this research using de Bruijn indices.

Further research on this topic involves creating a generic framework for trustworthy program refactoring [8]. It proposes a general high-level abstraction of the refactoring pipeline that allows it to be used in multiple languages. In their work, as concrete examples, they used Java to show usage in object-oriented paradigms and Erlang for functional paradigms. With this framework in mind, the research conducted in this study might be extended to create a fully functional refactoring tool.

9 Conclusions and Future Work

In the conclusion and future work section, we summarize the key findings and contributions of our study, emphasizing their importance. Additionally, we explore potential paths for further development in this area.

9.1 Conclusions

With this research paper, we achieved the final goal of demonstrating the feasibility of proving the correctness of currying and uncurrying refactoring using formal methods. As for the other contributions, a small functional programming language was created, which we argue resembled Haskell. This language could be used as a foundation for extending this research to other more standard languages. A basic proof-of-concept refactoring tool was created as a function that works directly on the internal representation of the language. The proofs have been formally described using Agda. With these proofs, we were able to show that the property of well-typedness and termination is retained and that the output values relate to each other in a specifically defined way.

9.2 Future work

As a next step in this research, the refactoring function would have to be extended to allow for more customizability and to be used as a proper developer tool. As previously mentioned in the limitations, currently the function refactors all occurrences of the given type, which might not be intended by the programmer. Preferably, we would want to include user interactions to make the tool more useful.

Furthermore, going more into the details of the implementation, the refactoring could be extended to work on the n-argument functions, as mentioned previously in the limitations. An alternative would be to use a more general definition of currying, which directly uses tuples, instead of special multi-argument constructs.

Lastly, the research done in this paper mainly focused on the single refactoring of currying; however, from the given one, we could relate a handful of other examples, which would have a similar approach to that described in this document. During the process of identifying edge cases, a case that included a function as an argument was identified. In practice, this case was an example of a higher-order function that took another function as an argument. This could

be considered separate refactoring on its own, which could be researched in a similar manner. If the same function is used subsequently, it might be beneficial to factor it out, thus making the code more modular and avoiding unnecessary redundancy.

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] 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.

[3] Abderrahmane Brahmi, Marie-Jo Carolus, David Delmas, Mohamed Habib Essoussi, Pascal Lacabanne, Victoria Moya Lamiel, Famantanantsoa Randimbivololona, Jean Souyris, and Airbus Operation SAS. Industrial use of a safe and efficient formal method based software engineering process in avionics. *Embedded Real Time Software and Systems (ERTS 2020)*, 2020.

[4] 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.

[5] Alessio Ferrari and Maurice H Ter Beek. Formal methods in railways: a systematic mapping study. *ACM Computing Surveys*, 55(4):1–37, 2022.

[6] Martin Fowler. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[7] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic testing of refactoring engines on real software projects. In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*, pages 629–653. Springer, 2013.

[8] Dániel Horpácsi, Judit Kőszegi, and Dávid J Németh. Towards a generic framework for trustworthy program refactoring. *Acta Cybernetica*, 25(4):753–779, 2022.

[9] John Paul, Nadya Kuzmina, Ruben Gamboa, and James Caldwell. Toward a formal evaluation of refactorings. In *Proceedings of The Sixth NASA Langley Formal Methods Workshop*, 2008.

[10] 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.

[11] The Agda Team. What is agda? https://agda.readthedocs.io/en/v2.6.3/getting-started/what-is-agda.html, 2023. [Online; accessed 26-May-2023].

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

[13] Xiang Yin, John Knight, and Westley Weimer. Exploiting refactoring in formal verification. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 53–62, 2009.