Maybe a List would be better?

Correct by construction Maybe to List refactorings in a Haskell-like language

José Carlos Padilla Cancio[1]

Supervisor(s): Jesper Cockx[1]**, Luka Miljak** [1]

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

Abstract

This paper concerns itself with correct by construction refactoring of `Maybe` values to `List` values in a Haskell-like language (HLL) as a case study on data-oriented refactorings. Our language makes use of intrinsically-typed syntax and de Bruijn indices for variables. Operational semantics are defined using big step semantics. We define a refactoring function which intrinsically verifies its well-typedness due to our intrinsically-typed syntax. The semantic validity of this refactoring is given by a separate proof. We use Agda, a functional language and theorem prover, to define and prove these properties. Techniques and concepts used in our refactoring and proof generalize well to other data-oriented refactorings.

1 Introduction

Refactoring is an important tool for developers. It allows them to improve the maintainability and readability of their codebase while preserving its behaviour. As such many, integration development environments (IDEs) attempt to automate this process [1]. Due to the automated nature of these tools, it is essential that they have a correct implementation and do not introduce hard to detect bugs, which IDEs attempt to guarantee through tests. However as Dijkstra famously said: "testing can be used to show the presence of bugs, but never to show their absence!" [2].

Formal verification on the other hand can guarantee program correctness. In it, we formally define our language using a mathematical model and reason about properties of the language and operations over that language. Additionally, by formally verifying the refactoring itself we guarantee its correctness for an arbitrary scenario and do not need to verify the resulting program after every operation.

Owing to the rigour of mathematical proofs, integrating formal verification into refactoring tools would give programmers guarantees on the correctness of the tool. This could be of particular use in safety-critical systems where a higher degree of certainty is required than in typical software engineering scenarios.

Soares notes in [3] that there is an unfortunate lack of formal verification in modern refactoring tools, owing to them being considered cost prohibitive. We are interested in increasing said formal verification, as there is a notable gap in the literature regarding research into refactoring tools for functional programming languages. Instead, the majority of the research relates to OOP languages and paradigms [4].

This is somewhat surprising, as functional programming languages (e.g. Haskell) lend themselves well to formalisation. They are deterministic, i.e. their output depends only on the input and have direct theoretical ties to lambda calculus, a robust mathematical model of programming languages. As such, many theorem provers are based on lambda calculus and functional programming, with Agda [5] being one of them. In contrast to other theorem provers however, Agda is primarily a programming language [5]. This makes it particularly well suited for the task of verifying properties of Haskell-like code. Additionally, tools like agda2hs [6] can help convert Agda code directly into Haskell code. For a brief overview of Agda and relevant theory, see section 2.

Our paper concerns itself with data-oriented refactorings, which we define as refactorings that change objects of a certain type into objects of another type while encoding the same "intrinsic" information. They are common in Haskell and similar languages, as it is often useful to first convert objects to a different type and maintain the semantics before making use of the new type's additional features. This paper is a case study on the refactoring from Maybe to List types. We argue that this conversion contains no loss of information as a `Nothing` is equivalent to an empty list and a `Just x` is equivalent to a list with only one element, namely `x`.

Most notable in the literature regarding refactoring Haskell programs is the tool HaRe [7]. Unfortunately, it has certain downsides: It is restricted to Haskell 98 while the current stable release is Haskell 2010 and supports a limited number of refactorings. Of these refactorings, few have been formally verified [8]. Accordingly, there have been desires to extend HaRe with more refactorings, such as in [9], which includes data-oriented refactorings.

Additionally, there is literature on the specifics of formally verified refactorings which relates to this thesis. Sultana and Thompson [10] define and formally verify a number of refactorings in typed and untyped lambda calculus using the proof assistant Isabelle/HOL. They make use of substitution and small step reductions in their language formalisation, which complicate the expression of semantics and lemmas. On the other hand, Barwell et al. [11] formalise and verify the renaming refactoring operation in a subset of Haskell 98 using the dependently typed language Idris. However, they only verify well-typedness [1] and not the semantic[2] behaviour. Note that neither of these approaches concern themselves with data oriented refactorings.

Given this gap in the literature, the relevance of data oriented refactorings and the attractive properties of Agda as a metalanguage, we formulate the following contributions of this paper:

- We formally define a Haskell-like language (HLL) in Agda (section 3) using
  - intrinsically typed syntax (subsection 3.1) and
  - big step semantics (subsection 3.2)
- We define a refactoring operation over this HLL converting objects of type `MaybeTy` to objects of type `ListTy` (section 4)
- We verify that this refactoring is well-typed (Due to intrinsic typing section 4)
- Finally, we define how semantics should change by defining the $v_o \mapsto_v v_n$ relation and then verify that this holds for our refactoring. (section 5)

This research sheds light on aspects of defining and verifying Maybe to List conversions that we argue generalize well to other data-oriented refactorings and ultimately contribute

[1] Also known as static semantics

[2] In this paper we refer to dynamic semantics as simply semantics

to the pool of knowledge regarding formally verified correct by construction refactorings. This claim is substantiated and elaborated on in section 6.

2 Theoretical Backgrounds and Agda

Agda is a dependently typed, purely functional, total programming language. It supports interactive programming with typed holes and a rich Emacs plugin. It functions as a proof assistant through the Curry-Howard isomorphism [12]. This section will elaborate on the definitions of these terms, relevant details about Agda and the benefits it provides.

2.1 Dependent Types

Dependently typed languages are a type of functional language with a very rich and extensible type system that allows types to depend on the value of expressions. This allows the programmer to create types that limit input to their functions to only accept values that are deemed to be meaningful or relevant. A common example of this is the `Vec a n` data type in Agda. `Vec a n` is identical to the polymorphic singly linked list `List a` data type with a relevant caveat, namely the argument `n` which is a natural number that specifies the length of the list. Following this logic, one can then write for example a `head` function which only accepts non-empty lists. Thus not having to consider the case of calling `head` on an empty list, as it is impossible by construction. An example of this in our paper can be seen in subsection 3.1. We define terms as a dependent data type $\_\vdash\_$ where the left argument is a context and the right argument is a HLL type. As such, we define terms as being dependently typed on the context they are written in and on the HLL type that they type check to.

2.2 The Curry-Howard Isomorphism

The Curry-Howard isomorphism is a correspondence between type systems and intuitionistic logic[3] [12], showing that one can be expressed as the other. It states that there is a mapping from propositional statements to types of programs and from proofs of those statements to programs of those types. In other words, if you state your proposition as a type, you can prove it to be valid by giving a type correct implementation of the function. On a surface level, using Agda types, the correspondence is as follows:

- p implies q $\cong$ P $\rightarrow$ Q [4]
- $p \wedge q \cong$ `P × Q`[5]
- $p \vee q \cong$ `P ⊎ Q`[6]
- $True \cong \top$
- $False \cong \bot$

In a way, this correspondence is quite intuitive. If p implies q, then given proof that p holds, we can construct a proof that q holds. If we have evidence that p and q hold, then we obviously have evidence that they both hold and given evidence that p holds or evidence that q holds, then we have evidence that at least one of them holds. The final two isomorphisms listed show the correspondence between true and false literals and types. A proposition being true makes it trivial to prove, as such it corresponds to the unit type, which is a type with a single constructor (named `tt` in Agda). On the other hand, the false literal, which is the empty type in the context of programs as proofs, is un-provable. The empty type cannot be inhabited by any type and has no constructors (thus it cannot be proven true). The primary caveat regarding Curry-Howard is that it refers to intuitionistic logic . This means that we must create an algorithm that constructs direct evidence of a proposition. As such, the law of excluded middle and double negation elimination are not inference rules in intuitionistic logic as they would be in classical logic.

[3]Also sometimes called constructivist logic

[4]Note that the right hand side denotes a function type signature form P to Q

[5]Generalized product type

[6]Generalized sum type

2.3 Agda

This means that we can utilize Agda to formally define our HLL, write a refactoring operation that operates over our HLL and formally verify its implementation in the same language. Additionally, Agda offers a host of features that aid us in these tasks, which will be elaborated on in the following paragraphs.

Agda is a total functional programming language. As such, it requires that all inputs to a function have a defined output, which by extension restricts the set of programs that can be written in Agda to programs that provably terminate.[7] This guarantee (or expectation) of termination is useful in applications where one wishes to formally verify program attributes. To ensure that programs terminate, Agda employs a termination checker, which only accepts structural recursion, i.e. recursion where each call is strictly smaller in size and thus converges to a base case.

Additionally Agda's holes allow us to place a `?` or `{! !}` in our program, which it accepts as a missing chunk of code. When we then load the file, it loads all holes and specifies what type is necessary to fill each hole. Holes allow for a form of interactive programming where one has a better understanding of the state of the current code, and can incrementally supply solutions which will only be accepted if they type check[8].

Like in Haskell, function signatures in Agda make use of currying, meaning that `f : A → B → C` describes a function that takes an argument of type `A` and an argument of type `B` and returns a value of type `C`. However, in addition to this, Agda supports implicit arguments. As an example of this, consider the function in listing 1. It has a number of implicit arguments required to construct the types that it accepts as input and returns. However, as the implementation shows, we can omit the implicit arguments and let Agda automatically infer them for us, thus de-cluttering our code.

Note that Agda is not always able to infer implicit arguments and sometimes needs the programmer to provide them, however this is relatively rare.

[7]With the exception of coinductive data types which are not relevant to this thesis

[8]Remember that as Agda is dependently typed, often type checking is proof of correctness

```
update∋PostMap : ∀ {ty n f} {Γ : Context n} → Γ
↪   ∋ ty → (mapContext Γ f)  ∋ f ty
update∋PostMap Z = Z
update∋PostMap (S l) = S update∋PostMap l
```

Listing 1: Example function for currying and implicit arguments

3 Language Design and Specification

In order to formally reason about operations on a language it is first necessary to define that language. As a baseline, we decided to define our language as an extended simply typed lambda calculus (STLC). Further sections will provide more details on our HLL. The first section will elaborate on our language's type system and syntax, while the latter section specifies the language semantics. Both sub-sections first describe relevant design decisions taken and then give the specification of that aspect of our HLL.

3.1 Type System and Syntax

Type systems help ensure program validity by rejecting expressions which are not meaningful. We say that a program is well-typed if it follows the rules of the relevant type system. In this section, we define our language's type system and syntax. We first give an overview of intrinsic typing and de Bruijn indices as relevant background before giving the full specification of our syntax and type system.

Intrinsically-typed Syntax

Our type system and syntax are defined using the approach of intrinsically-typed syntax as defined and implemented in [13]. Intrinsic typing is a technique in which you define an expression by the type that it has. Should that expression require sub-expressions, then those are also defined with their type. In our code, this is defined as an expression type checking to τ under a typing context Γ by the data type $\Gamma \vdash \tau$. Note however, that following convention our inference rules use the notation $\Gamma \vdash e : \tau$

Listing 2, detailing the syntax for the + operation, shows a useful example to introduce this concept. Its constructor takes two arguments which type check to `IntTy` (i.e. the terms being added) and using those terms constructs a term that type checks to a `IntTy` (i.e. the sum). Should we attempt to construct a + expression where one of the arguments does not type check to `IntTy`, Agda would not accept the expression as we have not provided valid arguments to the constructor.

```
_+_ : Γ ⊢ IntTy → Γ ⊢ IntTy → Γ ⊢ IntTy
```

Listing 2: Constructor for + operation

This follows from the primary benefit of intrinsic typing, namely that as the syntax and type system are defined simultaneously, it is impossible to construct expressions that are not well-typed. Crucially, this in turn means that any function that operates on expressions will simultaneously be a proof that the outcome is also well-typed (as otherwise it would not be able to be constructed), thus sparing us from having to define a separate well-typedness proof.

De Bruijn Indices

Concerning variables, as our refactoring does not have strong ties to variable naming, we opted to instead make use of de Bruijn indices as defined in [14] and implemented in [13]. Conceptually, these indices are simply numerical indices stating the position in the typing context/environment which contains the type/value that a given variable is referring to. Note that in our current definition of HLL, there are only two expressions which change the context: functions which add a single type to the context (i.e. the argument) and case statements which add the types and values that the pattern match makes available to the programmer to the context [9]. This approach simplifies our language and proof, as it is not necessary to deal with name shadowing or seeing if a name is contained within a given context/environment. More specifically, we implement contexts and de Bruijn indices as shown in listing 3.

```
data Context : Set where
    ∅ : Context
    _,_ : Context → Type → Context

data _∋_ :Context → Type → Set where
    Z : Γ , A ∋ A
    S_ : Γ ∋ A → Γ , B ∋ A
```

Listing 3: Definitions in Agda for the typing context and lookup judgement

For starters, typing `Context`s are defined inductively with constructors for an empty context ∅ and for extending a context with a type _,_. As such, contexts are similar to lists of types (i.e. empty list, :: operator). Lookup judgements are defined using the dependently typed _∋_ [10] data type. _∋_ takes as (explicit) arguments a typing `Context` and the `Type` we wish to find in said context. There are two constructors for lookup judgements, the `Z` and the `S_` operator. `Z` simply states that the first element in the `Context` is of the type we want. `S_` on the other hand, takes evidence that sub-`Context` Γ contains type `A` and then returns evidence that the extended context Γ, `B` contains `A`. On the one hand, we can interpret ∋ as a natural number index of a list-like object, with `Z` $\cong$ 0 and `S l` $\cong$ `suc n`. More pertinent to our work however, is the interpretation of ∋ as evidence that a `Context` contains a certain type at a given position (and later on when specifying our semantics a given value at a certain position).

Specification of Intrinsically-typed Syntax

As the typing rules in Figure 1 indicate, the HLL supports `IntTy` as a base type with built in operations for addition +, subtraction - and multiplication *. Additionally, we support the algebraic data types `MaybeTy` and `ListTy`, as they are necessary for our proof. Note that although there is a single base type, these types are indexed by types so may be nested and in the future contain other base types.

Given that we have parametric types, we also need to support some Haskell-like mechanism to extract the values

[9]See next section for specification of syntax
[10]pronounced "ni"

$$\frac{x : \mathbb{Z}}{\Gamma \vdash x : \texttt{IntTy}} \qquad \frac{\Gamma \vdash x : \texttt{IntTy} \quad \Gamma \vdash y : \texttt{IntTy}}{\Gamma \vdash x \texttt{ + } y\texttt{, } x \texttt{ - } y\texttt{, } x \texttt{ * } y : \texttt{IntTy}}$$

$$\frac{}{\Gamma \vdash \texttt{Nothing} : \texttt{MaybeTy } \tau} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \texttt{Just } x : \texttt{MaybeTy } \tau}$$

$$\frac{}{\Gamma \vdash \texttt{[]} : \texttt{ListTy } \tau} \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash xs : \texttt{ListTy } \tau}{\Gamma \vdash x\texttt{::}xs : \texttt{ListTy } \tau}$$

Figure 1: Typing rules for data types and operation on integers

$$\frac{\Gamma \vdash m : \texttt{MaybeTy } \tau_i \quad \Gamma \vdash e_n : \tau_r \quad \Gamma, \tau_i \vdash e_j : \tau_r}{\Gamma \vdash \texttt{caseM } m \texttt{ of nothingP to } e_n \texttt{ or justP to } e_j : \tau_r}$$

$$\frac{\Gamma \vdash m : \texttt{ListTy } \tau_i \quad \Gamma \vdash e_{[]} : \tau_r \quad \Gamma, \tau_i, \texttt{ListTy } \tau_i \vdash e_{::} : \tau_r}{\Gamma \vdash \texttt{caseL } m \texttt{ of []P to } e_{[]} \texttt{ or ::P to } e_{::} : \tau_r}$$

Figure 2: Typing rules for case statements.

within them and thus decided to support case statements for `Lists` and `Maybes`. However, as seen in Figure 2 pattern matching is hard coded. The first branch of a case statement corresponds to the "empty" case (`Nothing` and `[]`) while the second branch corresponds to the "nonempty" case (`Just x` and `x::xs`).

$$\frac{\Gamma, \tau_a \vdash b : \tau_r}{\Gamma \vdash \lambda b : \tau_a \Rightarrow \tau_r} \qquad \frac{\Gamma \vdash f : \tau_a \Rightarrow \tau_r \quad \Gamma \vdash a : \tau_a}{\Gamma \vdash f \cdot a : \tau_r}$$

Figure 3: Typing rules for lambda functions and function application.

Syntax and type rules for functions λ and function applications $\_ \cdot \_$ are given in Figure 3. Note that due to our usage of de Bruijn indices, functions are defined by their bodies. We do not support recursion [11] as we did not consider it relevant enough to our refactoring to include in the scope of this project.

The typing rule for variables given in Figure 4 stands out as it is the first one that does not (technically) depend on subexpressions. Instead, it takes a lookup judgement as defined in the previous section, which serves as evidence that the type is in the `Context` and returns a term of that type (i.e. it refers to that position in the context/environment)

3.2 Operational Semantics

Now that we have defined how to construct well-typed (i.e. meaningful) expressions, we need some mechanism to assign meaning to them. In order to do this, we have to define what values our language can return as well as what the inference rules are for mapping well-typed expressions to values. These rules are defined by our operational semantics (more specifically big step semantics).

The first section deals with which values our HLL supports and how variables are assigned to values. Afterwards, we define big-step semantics and motivate our decision to utilize it. Finally, we conclude with the formal specification of our language's operational semantics.

[11]Fixpoints or letrec-like constructs in the context of STLC

$$\frac{l : \Gamma \ni \tau}{\Gamma \vdash \texttt{var } l : \tau}$$

Figure 4: Typing rule for variables.

Environments and Values

Listing 4, based on the work in [13], contains the Agda code defining both our `Env` and `Value` data types. Note that `Values` are indexed over types, which allows us to place restraints on the values that expressions can evaluate to.

We can generally separate our values into three broad categories: base `Values`, inductive `Values` and closures (`ClosV`). Base `Values` return a value immediately [12] and inductive values depend on some inner value such as the head and tail of a list in `ConsV` which eventually terminate in some base `Values`. Closures are a bit more tricky and correspond to the value that functions evaluates to. They function by "closing off" and storing the `Env` where the function was defined along with the function body. As such, in order to apply functions, we can simply extend the "closure environment" with the argument to the function and evaluate the body.

```
data Env : {n : ℕ} → Context n → Set

data Value : Type → Set where
    IntV : ℤ → Value IntTy
    -- MaybeTy
    NothingV : Value (MaybeTy ty)
    JustV : Value ty → Value (MaybeTy ty)
    -- ListTy
    NilV : Value (ListTy ty)
    ConsV : Value ty → Value (ListTy ty) → Value (ListTy ty)
    -- Closures
    ClosV : Env Γ →  Γ , argTy ⊢ retTy → Value (argTy ⟶ retTy)
data Env where
    ∅' : Env ∅
    _,'_ : Env Γ → (v : Value ty) → Env (Γ , ty)
```

Listing 4: Definitions in Agda for values and environment

As aforementioned, we use environments to associate variables to values, which in the context of this paper will be denoted using lowercase gamma γ. As listing 4 shows, `Envs` are indexed over `Context`. As such, its constructors strike a close resemblance to those of `Contexts`. $\emptyset$ corresponds to $\emptyset'$ while `_,_` corresponds to `_,'_`. Note that when extending environments, the given `Value` must be of the same type as in the context. This correspondence, in turn, allows us to use our previous de Bruijn indices to locate values in the environment.

Big Step Semantics

Big-step semantics are a form of operational semantics described originally by Kahn in [15] [13]. Its semantic rules are defined as a sequence of general preconditions (i.e. big steps)

[12]or in the case of `IntV` depend on Agda's $\mathbb{Z}$ specification

[13]Kahn referred to them as natural semantics

which need to be fulfilled in order to infer the evaluation of an expression. This allows us to abstract away the reductions described in small step semantics and only define what is essential. We construct our big step semantics as a dependent data type, inspired by the work of [13], in listing 5. Notationally $\gamma \vdash_e e \downarrow v$ indicates that an expression e in the environment γ evaluates to the value v.

```
data _⊢e_↓_ : Env Γ → (Γ ⊢ ty) → Value ty → Set
```

Listing 5: Type signature of big step semantics data type

As a simple example of big step semantics and how it is implemented in our language, consider listing 6 detailing the big step semantics for the "+" operation. Given a left and right expression that both evaluate to `IntV` values, we define addition to be an `IntV` value wrapping the sum of the left and right values. As it is not relevant to our refactoring and for brevity's sake, we use the underlying Agda specifications for operations over integers, which we denote as +z or $+_z$.

```
↓+ : -- Eval lhs
    → γ ⊢e l ↓ IntV i
    -- Eval rhs
    → γ ⊢e r ↓ IntV j
    -- Eval sum
    → γ ⊢e l + r ↓ IntV (i +z j)
```

Listing 6: Semantics for + operation

Big step semantics define inference rules which more closely resemble the function of interpreters, which give programs meaning in the real world. However, it is important to note that this is not an interpreter; we are simply defining what certain terms should evaluate to given that certain preconditions hold.

Specification of Big Step Semantics

$$\frac{}{\gamma \vdash_e \texttt{Int}\ i \downarrow \texttt{IntV}\ i} \downarrow \texttt{Int} \qquad \frac{\gamma \vdash_e x \downarrow \texttt{IntV}\ i \quad \gamma \vdash_e y \downarrow \texttt{IntV}\ j}{\gamma \vdash_e x\ \texttt{+}\ y \downarrow \texttt{IntV}\ i +_z j} \downarrow \texttt{+}$$

$$\frac{}{\gamma \vdash_e \texttt{Nothing} \downarrow \texttt{NothingV}} \downarrow \texttt{Nothing} \qquad \frac{\gamma \vdash_e x \downarrow v}{\gamma \vdash_e \texttt{Just}\ x \downarrow \texttt{JustV}\ v} \downarrow \texttt{Just}$$

$$\frac{}{\gamma \vdash_e \texttt{[]} \downarrow \texttt{NilV}} \downarrow \texttt{[]} \qquad \frac{\gamma \vdash_e x \downarrow h \quad \gamma \vdash_e xs \downarrow t}{\gamma \vdash_e x\ \texttt{::}\ y \downarrow \texttt{ConsV}\ h\ t} \downarrow \texttt{::}$$

Figure 5: Semantics for data types and operations on integers

Following the structure of the specification in subsection 3.1, we begin our specification of the big step semantics of our HLL with the data type constructors and operations over integers given in Figure 5. Regarding integer operations, note that we only give the semantic rule for addition, as the rules for subtraction and multiplication are identical with updated operations.

In Figure 6 we define the semantics of case statements. Each inference rule has two preconditions: the evaluation of the term being pattern matched on and the evaluation of the clause which corresponds to that pattern. Note that as such

`Pattern matching on a Maybe`

$$\frac{\gamma \vdash_e m \downarrow \texttt{JustV}\ u \quad \gamma, u \vdash_e e_j \downarrow v}{\gamma \vdash_e \texttt{caseM}\ m\ \texttt{of nothingP to}\ e_n\ \texttt{or justP to}\ e_j \downarrow v} \downarrow \texttt{caseMJ}$$

$$\frac{\gamma \vdash_e m \downarrow \texttt{NothingV} \quad \gamma \vdash_e e_n \downarrow v}{\gamma \vdash_e \texttt{caseM}\ m\ \texttt{of nothingP to}\ e_n\ \texttt{or justP to}\ e_j \downarrow v} \downarrow \texttt{caseMN}$$

`Pattern matching on a List`

$$\frac{\gamma \vdash_e m \downarrow \texttt{Consv}\ h\ t \quad \gamma, h, t \vdash_e e_{::} \downarrow v}{\gamma \vdash_e \texttt{caseL}\ m\ \texttt{of []P to}\ e_{[]}\ \texttt{or ::P to}\ e_{::} \downarrow v} \downarrow \texttt{caseL::}$$

$$\frac{\gamma \vdash_e m \downarrow \texttt{NilV} \quad \gamma \vdash_e e_{[]} \downarrow v}{\gamma \vdash_e \texttt{caseL}\ m\ \texttt{of []P to}\ e_{[]}\ \texttt{or ::P to}\ e_{::} \downarrow v} \downarrow \texttt{caseL[]}$$

Figure 6: Semantics for case expressions

we need an inference rule for each case: one for the empty case and one for the non-empty case.

$$\frac{}{\gamma \vdash_e \lambda b \downarrow \texttt{ClosV}\ \gamma\ b} \downarrow \lambda$$

$$\frac{\gamma \vdash_e f \downarrow \texttt{ClosV}\ \gamma_c\ b \quad \gamma \vdash_e g \downarrow a \quad \gamma_c, a \vdash_e b \downarrow v}{\gamma \vdash_e f \cdot g \downarrow v} \downarrow \cdot$$

Figure 7: Semantics for functions and application

Figure 7 deals with the semantics of functions and their applications. Functions have an axiomatic inference rule (i.e. no preconditions) namely a `ClosV`. Applications, on the other hand, have three steps: evaluating a function to a closure, evaluating the argument and evaluating the body of the function under the closure environment extended by the argument value.

$$\frac{l : \Gamma \ni \texttt{A}}{\gamma \vdash_e \texttt{var}\ l \downarrow \texttt{v-lookup}\ \gamma\ l} \downarrow \texttt{var}$$

Figure 8: Sematic rule for variables.

Unlike in subsection 3.1, defining variable semantics in Figure 8 is not quite as trivial. As variables only contain a pointer to the context, we need some mechanism to extract that value from the environment at the given lookup. We call this function `v-lookup` and give its definition in Agda in listing 7.

4 Refactoring Function

Now that we have defined our language and its semantics, we can construct our refactoring function. First of all, our refactoring operates at the top level (i.e. with an empty context). This follows from the fact that the operation changes all type signatures from `MaybeTy` to `ListTy`. If we were to refactor only a sub-expression of the program $\Gamma \vdash ty$, the return type of some sub-expression may depend on Γ which we have not refactored and thus may be a `MaybeTy`. Therefore, our function starts from an empty context and then calls a recursive

```
v-lookup : Env Γ → Γ ∋ ty → Value ty
v-lookup (γ ,' v) Z = v
v-lookup (γ ,' v) (S l) = v-lookup γ l
```

Listing 7: Definition of v-lookup in Agda

```
1  refactorListH : Γ ⊢ ty → (ev : Extend Γ Under MaybeTy→ListTy) → (constructRefContext ev) ⊢ MaybeTy→ListTy ty
2  -- Variables
3  refactorListH (var x) ev = var (update∋PostRef x)
4  -- MaybeTy terms to ListTy
5  refactorListH Nothing ev = []
6  refactorListH (Just e) ev = refactorListH e ev :: []
7  -- caseM to caseL
8  refactorListJH (caseM m of nothingP to nC or justP to jC) ev = -- construct caseL term. Recurse on all subterms but jC with ev. Recurse on jC with (eo-pad (ListTy _) (eo-elem ev))
9  -- All other cases
10 refactorListH t ev = -- Return if base term, else construct term while recursing on sub-terms adding variables to context with eo-elem where necessary
11
12 refactorList : Γ ⊢ ty → (ev : Extend Γ Under MaybeTy→ListTy) → (constructRefContext ev) ⊢ MaybeTy→ListTy ty
13 refactorList = refactorListH
```

Listing 8: Pseudo-agda implementation of refactor and helper function

helper function that performs the actual refactoring. We define the functions in psuedo-Agda in listing 8.

Keen-eyed readers will note that we use functions to define and enforce properties of our refactored expression, namely `constructRefContext` and `MaybeTy→ListTy`. `MaybeTy→ListTy` defines how types should change post refactoring, i.e. it swaps out `MaybeTy` for `ListTy` [14]. `constructRefContext` constructs the context that the refactored expression would operate under by using the `Extend_Under_` data type.

This data structure as defined in listing 9 is dependently typed on the original `Context` and the relevant swap function (in our case `MaybeTy→ListTy`), and is conceptually the list of operations necessary to construct the new context. Its three constructors indicate constructing an empty `Context` (eo-root), appending a type to the context (eo-elem) and appending a type to the context as a side effect of the refactoring operation (eo-pad). Using this information, the `constructRefContext` function can construct the new context.

Crucially, `Extend_Under_` helps us deal with the fact that when converting from `caseM` to `caseL`, our refactoring doesn't preserve the context's structure. To demonstrate

```
data Extend_Under_ : Context → (Type → Type) → Set where
    eo-root : Extend ∅ Under f
    eo-elem : Extend Γ Under f → Extend (Γ , ty) Under f
    eo-pad : Type → Extend Γ Under f → Extend Γ Under f
```

Listing 9: Agda implementation of Extend_Under_ data structure

this, consider listing 10. Specifically note that while applying our refactoring, we convert `e_j` to `e_::`, the clause for the just pattern to the clause for the nonempty list pattern. When we pattern match on a `Just`, the inner type of the `MaybeTy` is made available (Γ, a), whereas when pattern matching on a nonempty list, both the head and tail are made available $(\Gamma_r, b,$ `ListTy` $b)$. As this step changes the structure of the context, we keep track of the insertions made (with `Extend_Under_`). Additionally, by virtue of this entry being new, it is never used by `e_::` and thus we can also safely "instruct" all variables to ignore it (i.e. with the `update∋PostRef` function).

```
-- Original expression
caseM matchOnM of
    NothingP to e_n
    or
    JustP to e_j

-- Post refactor
caseL matchOnL of
    []P to e_[]
    or
    ::P to e_::
```

Listing 10: Refactoring of caseM statement

`vars` and `caseMs` aside, the majority of the function is fairly straightforward. In the cases detailed in lines 8-9, we swap `MaybeTy` terms for their corresponding `ListTy` equivalents. `Nothing` becomes `[]` and `Just e` becomes `refactorListH e ev :: []`. For all other terms, base terms remain unchanged, while for polymorphic terms we construct the same parent term and recurse into its sub-terms.

5 Verifying Semantic Behaviour

Many properties of our refactoring have already been verified by construction. Intrinsically-typed syntax guarantees well-typedness and dependently typed values exclude `MaybeTy` terms from our syntax . This leaves only verifying the semantic behaviour of our refactoring as an explicit proof. In this section we will first define relevant relations necessary for our proof and then give an overview of the proof itself.

5.1 Relations

In order to do this, we need to define what the desired behaviour of our refactoring even is. As such, we introduce the relation $v_o \mapsto_r v_n$ to express that if an expression evaluated to v_o, then after refactoring we expect it to evaluate to v_n .

[14]This function is applied recursively on types with sub-types

The inference rules for this relation are defined in figures 9 and 10. Our definition is inductive and ensures that `MaybeTy` values have been replaced according to what our refactoring should produce while also ensuring that the portions of values which are not `MaybeTy` remain untouched.

$$\frac{x = y}{\texttt{Int}\ x \mapsto_v \texttt{IntV}\ y}$$

$$\frac{}{\texttt{NothingV} \mapsto_v \texttt{NilV}} \qquad \frac{x \mapsto_v h \quad t = \texttt{NilV}}{\texttt{JustV}\ x \mapsto_v \texttt{ConsV}\ h\ t}$$

$$\frac{h_o \mapsto_v h_n \quad t_o \mapsto_v t_n}{\texttt{ConsV}\ h_o\ t_o \mapsto_v \texttt{ConsV}\ h_n\ t_n} \qquad \frac{}{\texttt{NilV} \mapsto_v \texttt{NilV}}$$

Figure 9: Inference rules for $\mapsto_v$ except closures

As is evident by the complexity of the inference rule, closures are somewhat of an exception. Our refactoring can change the length and content of a closure environment and by definition changes its body. As these changes are context dependent and to avoid circularity, we present the definition in Figure 10. This rule is inspired by the concept of extensional equivalence [16] and states that two functions are equivalent if for all $a_o \mapsto_v a_n$ arguments they produce $r_o \mapsto_v r_n$ results.

$$\frac{\forall a_o \mapsto_v a_n \quad \gamma_o, a_o \vdash_e b_o \downarrow r_o \quad \gamma_n, a_n \vdash_e b_n \downarrow r_n \quad r_o \mapsto_v r_n}{\texttt{ClosV}\ \gamma_o\ b_o \mapsto_v \texttt{ClosV}\ \gamma_n\ b_n}$$

Figure 10: Inference rules for $\mapsto_v$ of closures

On top of this, we define another relation $e, \gamma_o \mapsto_e \gamma_n$ for environments. It is defined in Figure 11 and conceptually states that, given a list of changes made to the context c (i.e. the `Extend_Under_` data type), all non padded entries to the environment a, b must be $a \mapsto_v b$.

$$\frac{}{\texttt{eo-root}, \emptyset' \mapsto_e \emptyset'} \qquad \frac{c, \gamma_o \mapsto_e \gamma_n}{(\texttt{eo-pad}\ x\ c), \gamma_o \mapsto_e \gamma_n, b} \qquad \frac{c, \gamma_o \mapsto_e \gamma_n \quad a \mapsto_v b}{(\texttt{eo-elem}\ c), \gamma_o, a \mapsto_e \gamma_n, b}$$

Figure 11: Inference rules for , $\mapsto_e$

5.2 Proof of Semantics

We have written our proof in Agda but for brevity will give a non-rigorous overview of the essence of the structure of our proof. In Agda our proof is structured similarly to our refactoring in that it is a top level function that calls a recursive helper. We will focus on the helper function in our overview.

Central Theorem. *Given a semantic derivation $d_o : \gamma_o \vdash_e e \downarrow v_o$ an environment γ_n such that $c, \gamma_o \mapsto_e \gamma_n$ and the semantic derivation $d_n : \gamma_n \vdash_e$ `refactorListH` $c\ e \downarrow v_n$ then $v_o \mapsto_v v_n$*

Proof. We will prove this statement by case analysis on (d_o, d_n), showing that all valid combinations of (d_o, d_n) preserve the central theorem. Most cases function by induction on the fact that all other cases hold.

Case 1 ($\downarrow$ `var`)**.** *Valid semantic derivations: ($\downarrow$ `var`, $\downarrow$ `var`)*

Proof. Since $c, \gamma_o \mapsto_e \gamma_n$ and our refactored expression cannot refer to padded entries, the central theorem holds by definition □

We now consider "base" derivations for simple expressions with no sub-expressions

Case 2 ($\downarrow$ `Int`)**.** *Valid semantic derivations: ($\downarrow$ `Int`, $\downarrow$ `Int`)*

Proof. `IntV` $i \mapsto_v$ `IntV` j if $i = j$. Our refactoring does not affect any underlying Agda integers in `Int` terms, thus $i = j$ and the theorem holds for this case. □

Case 3 ($\downarrow$ `Nothing`)**.** *Valid semantic derivations: ($\downarrow$ `Nothing`, $\downarrow$ `[]`)*

Proof. As both derivations are base derivations and `NothingV` $\mapsto_v$ `NilV`, this holds. □

Case 4 ($\downarrow$ `[]`)**.** *Valid semantic derivations: ($\downarrow$ `[]`, $\downarrow$ `[]`)*

Proof. As both derivations are base derivations and `NilV` $\mapsto_v$ `NilV`, this holds. □

This concludes the set of proofs for simple base derivations.

Case 5 ($\downarrow$ +, $\downarrow$ -, $\downarrow$ *)**.** *Valid semantic derivations: ($\downarrow$ +, $\downarrow$ +), ($\downarrow$ -, $\downarrow$ -), ($\downarrow$ *, $\downarrow$ *)*

Proof. The same logic applies to all 3 operations: All operations result in integers, thus we need to show that $i_o +_z j_o \mapsto_v i_n +_z j_n$. If $i_o = i_n$ and $j_o = j_n$ hold, then it would follow trivially. For terms of adding two `Int` terms, this holds by case 2 as a base case. By induction, this also holds for complex terms. □

Case 6 ($\downarrow$ `Just`)**.** *Valid semantic derivations: ($\downarrow$ `Just`, $\downarrow$ `::`)*

Proof. `JustV` $v \mapsto_v$ `ConsV` $h\ t$ if $v \mapsto_v h$ and $NilV = t$. The latter holds by the definition of our refactoring. The former holds by induction. □

Case 7 ($\downarrow$ `::`)**.** *Valid semantic derivations: ($\downarrow$ `::`, $\downarrow$ `::`)*

Proof. `ConsV` $h_o\ t_o \mapsto_v$ `ConsV` $h_n\ t_n$ if $h_o \mapsto_v h_n$ and $t_o \mapsto_v t_n$. This holds by induction □

Case 8 (Valid case statements)**.** *Valid semantic derivations: ($\downarrow$ `caseMJ`, $\downarrow$ `caseL::`), ($\downarrow$ `caseMN`, $\downarrow$ `caseL[]`), ($\downarrow$ `caseL::`, $\downarrow$ `caseL::`), ($\downarrow$ `caseL[]`, $\downarrow$ `caseL[]`)*

Proof. For all these cases, we show that by induction the respective clauses of the case are $e_o \mapsto_v e_n$. We extend the environment according to inner values for all non padded entries. The relation holds for these extensions since the values being matched on are $m_o \mapsto_v m_n$ (e.g. the head and tail of the lists will have this property in the `caseL` `caseL` case) □

Case 9 (Absurd case statements)**.** *As we defined the inputs to the Agda function very broadly, we must consider the following: ($\downarrow$ `caseL::`, $\downarrow$ `caseL[]`), ($\downarrow$ `caseL[]`, $\downarrow$ `caseL::`), ($\downarrow$ `caseMJ`, $\downarrow$ `caseL[]`), ($\downarrow$ `caseMN`, $\downarrow$ `caseL::`)*

Proof. All of these combinations are absurd as the terms they match on are not $m_o \mapsto_v m_n$. As such, these rule combinations are impossible and the central theorem holds by principle of explosion. □

Case 10 ($\downarrow \lambda$). *Valid semantic derivations: ($\downarrow \lambda$, $\downarrow \lambda$)*

Proof. Take an arbitrary a_o and a_n such that $a_o \mapsto_v a_n$ and extend the bodies of the functions to the semantic rules $\gamma_{co}, a_o \vdash_e b_o \downarrow r_o$ $\gamma_{cn}, a_n \vdash_e b_n \downarrow r_n$. By induction, as we have shown that all other cases result in $r_o \mapsto_v r_n$, this holds. □

Case 11 ($\downarrow \cdot$). *Valid semantic derivations: ($\downarrow \cdot$, $\downarrow \cdot$)*

Proof. As we have shown in case 10, the closures are $clos_o \mapsto_v clos_n$, which by definition means that for all arguments $a_o \mapsto_v a_n$, the results are $r_o \mapsto_v r_n . r_o = v_o$ and $r_n = v_n$ so the central theorem holds. □

As the central theorem holds for all possible cases, the theorem holds. □

6 Conclusions, Limitations and Future Work

6.1 Contribution

This paper makes use of Agda to define a HLL and a refactoring operation (swapping `Maybe` values to `List` values). We successfully verify well-typedness and valid semantic behaviour fully with the use of Agda.

The benefits of dependent types were of great assistance in this task, as they allow one to not only limit input to the cases which are relevant, but also to enforce desirable properties of the result, e.g. making `Value`'s indexed on types and restricting the values one can have post refactoring to exclude any `MaybeTy` value.

We discovered techniques for constructing our refactoring and proof that were instrumental and we believe generalize well to formally verifying refactorings.

Firstly, our $\mapsto_v$ relation dictating how we expect program results to change post refactoring helped explicitly express how our refactoring should affect semantic behaviour. Specifically in the case for closures, its use of a weaker “contextual equivalence” helped us define what valid transformations of closures are without having to rely on our refactoring definition to define the body of the closure. Additionally, the definition for closures makes proving function application trivial. We believe that this notion is also relevant to other data-oriented refactorings where results can strictly change but information is still preserved in some sense. The specific definition for closures generalizes to most refactorings, as they understandably may change code.

Our `Extend_Under_` data type was of great help when dealing with changes in the structure of the `Context` and `Env` by accumulating all changes made to the `Context` and labelling whether that addition existed before the refactoring or is simply a side effect of the refactoring. This notion would extend well to other data-oriented refactorings and generally refactorings that change the structure of the environment.

6.2 Limitations and future work

There are a number of limitations to our work as well as general future work that we can directly observe.

To start with, as the $\mapsto_v$ relation relies on “quasi” contextual equivalence in the case of comparing closure values our relation for closures is relatively weak. A refactoring could potentially admit refactorings that radically change the bodies of closures but would still be accepted, as it does technically affect semantic behaviour. It is worth investigating if there is a stronger definition that fits our purposes and is not circular or whether the strong constraints of the other cases of the relation exclude this possibility.

A significant portion of our results and approach is dependent on the fact that we utilize de Bruijn indices instead of named variables, which is not how Haskell or most programming languages approach variables. We argue that this is not a major limitation as the general effect or intent of the individual approaches apply to named variables as well, e.g. changing the structure of the environment as a consequence of the refactoring. Nonetheless, it would be good to adapt our approach to make use of named variables to see if the approach does indeed map and to uncover results which are closer to the real world nature of Haskell and similar programming languages.

Although we intrinsically verify well-typedness through the usage of intrinsically typed syntax, we still need to use an extrinsic proof to verify that the semantic behaviour matches expectations. Future work could investigate the feasibility of attempting to integrate this property into the refactoring function directly, thus ensuring that refactoring functions are truly “correct by construction”.

Another noteworthy limitation is our decision to hard code case statements due to time constraints. It is worth investigating how we can model generic pattern matching, both in regards to number and order of patterns as well as depth of the patterns. Additionally, it is worth investigating if there is a need for additional methods or lemmas for this more generic definition.

Formulating a generic definition of pattern matching in a HLL and verifying its behaviour would also help pave the way for defining a generic data oriented refactoring function. Given that many of the data types and approaches used in this paper are not specific to the conversion of `Maybe` to `List`, it is worth investigating how these techniques could be used to define and verify other data oriented refactorings. Future work could even investigate the possibility of defining a generic function that, given the minimal amount of information, both constructs a function to operate the refactoring and verifies its correctness (in terms of well-typedess, semantics and potentially other properties). As our case statements are hard coded and tied to the type being matched on, this research would depend on uncovering how to model and verify generic pattern matching.

Ackowledgements

First and foremost, I am immensely grateful to my supervisors, Luka Miljak and Jesper Cockx. Their guidance and feedback has been invaluable in this endeavour and I never would have completed this thesis without it. Specifically I would like to thank Jesper Cockx for their wonderful functional programming course, that sparked my interest in this field. Dank je wel.

To my friend William, I thank him for recommending said course to me and for motivating my descent into this rabbit hole. Shukran habibi.

To my incredible partner, Astrid, thank you for standing by my side and dealing with my moments of stress, frustration, long hours spent immersed in research and even longer rants. Additionally I would like to thank her for teaching me the usage of commas in the English language.

I would also like to acknowledge my sister for her constant motivation and belief in my abilities. Your unwavering faith in me even when I doubted myself has been pivotal in my academic journey. Gracias por creer en mi, Isa.

I would also like to thank my group mates for their support and advice, helping me avoid pitfalls they had already dealt with. Specifically I would like to thank Jeroen Bastenhof for inspiring me with the `Extend_Under_` data type.

Lastly, I want to express my deepest appreciation to my parents for their unending support and encouragement to pursue my dreams and happiness in life. Gracias por decir que sigua lo que ame.

References

[1] M. Fowler, *Refactoring*. Addison-Wesley Professional, 2018.

[2] B. Randell and J. Buxton, "Software engineering techniques: Report of a conference sponsored by the nato science committee, rome, italy, 27th-31st october 1969," 1970.

[3] G. Soares, "Making program refactoring safer," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pp. 521–522, 2010.

[4] E. A. AlOmar, M. W. Mkaouer, C. Newman, and A. Ouni, "On preserving the behavior in software refactoring: A systematic mapping study," *Information and Software Technology*, vol. 140, p. 106675, 2021.

[5] A. Bove, P. Dybjer, and U. Norell, "A brief overview of agda-a functional language with dependent types.," in *TPHOLs*, vol. 5674, pp. 73–78, Springer, 2009.

[6] J. Cockx, O. Melkonian, L. Escot, J. Chapman, and U. Norell, "Reasonable agda is correct haskell: writing verified haskell using agda2hs," in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, pp. 108–122, 2022.

[7] H. Li, S. Thompson, and C. Reinke, "The haskell refactorer, hare, and its api," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 29–34, 2005.

[8] H. Li and S. J. Thompson, "Formalisation of haskell refactorings.," *Trends in Functional Programming*, pp. 95–110, 2005.

[9] C. Brown, "A collection of ideas for haskell transformation," 2006.

[10] N. Sultana and S. Thompson, "Mechanical verification of refactorings," in *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp. 51–60, 2008.

[11] A. D. Barwell, C. M. Brown, and S. 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.

[12] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[13] P. Wadler, W. Kokke, and J. G. Siek, *Programming Language Foundations in Agda*. Aug. 2022.

[14] N. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem," *Indagationes Mathematicae*, vol. 75, no. 5, p. 381–392, 1972.

[15] G. 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*, pp. 22–39, Springer, 1987.

[16] J. H. Morris Jr, *Lambda-calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology, 1969.