Proving Renaming for Haskell via Dependent Types: A Case-Study in Refactoring Soundness
Extended Abstract

Adam D. Barwell
Department of Computing
Imperial College London
London, UK
a.barwell@imperial.ac.uk

Christopher Brown
School of Computer Science
University of St Andrews
St Andrews, UK
cmb21@st-andrews.ac.uk

Susmit Sarkar
Susmit.Sarkar@st-andrews.ac.uk

We present a formally verified renaming refactoring for a subset of Haskell 98 giving a case-study in proving soundness properties of Haskell refactorings. Our renaming is implemented in the dependently-typed language Idris, which allows us to encode soundness proofs as an integral part of the implementation. We give the formal definition of our static semantics for our Haskell 98 subset, which we encode as part of the AST, ensuring that only well-formed programs may be represented and transformed. This forms a foundation upon which refactorings can be formally specified. We then define soundness of refactoring implementations as conformity to their specification. We demonstrate our approach via renaming, a canonical and well-understood refactoring, giving its implementation alongside its formal specification and soundness proof.

1 Introduction

Refactoring [4] is the process of *changing the structure* of a program *without changing its behaviour* and is a common practice aimed at making a program more understandable, accessible, or amenable to further alterations to a program's design. Refactorings can be applied manually, which is both a tedious and error-prone process [11]. Alternatively, refactorings can be applied semi-automatically via refactoring tools. There are many such tools [8], and they are an important inclusion in the software developer's arsenal. Through automation, they can both simplify the workflow and guard against common human mistakes, such as overlooking one file within hundreds. Unfortunately, automated tools are not immune to error. Given an erroneous refactoring implementation, they may change the behaviour of code silently in unexpected and undesirable ways. The difficulty in constructing refactoring tools is generally a consequence of refactorings being specifically concerned with transformations over *syntax*. Unlike compilers, for instance, which can define and effect transformations over a small desugared core language, refactoring tools must represent and transform both concrete and abstract syntax. Accordingly, the implementation of a refactoring tool necessarily grows in complexity not only with the complexity of its refactorings, but also with the number of syntactic constructs that the target language defines. As refactoring tools grow in power and complexity, and are applied to ever larger code bases, so too increases the opportunity for error. It is therefore crucial that refactoring tools adhere to the key correctness criterion of not changing program behaviour. Typical approaches are reliant on the testing code coverage of their implementation to give confidence that refactorings are performed in the desired way. Yet testing is not infallible; bugs can still occur within well-tested tools (e.g. [5]). Alternative approaches use the methods of formally verified software [7, 10, 9], which may provide soundness guarantees for a refactoring system. However, they often separate proof from implementation, even in cases

Submitted to:
WPTE 2021

```
1  module Sum where
2
3  import Prelude hiding (sum)
4
5  sum n []    = n
6  sum n (h:t) = (+) h rest
7   where
8    rest = (sumRest n t)
9    sumRest n t = sum n t
10
11 main = sum 0 [1..4]
```

(a) Before

```
1  module Sum where
2
3  import Prelude hiding (fold)
4
5  fold n []    = n
6  fold n (h:t) = (+) h rest
7   where
8    rest = (sumRest n t)
9    sumRest n t = fold n t
10
11 main = fold 0 [1..4]
```

(b) After

Figure 1: Renaming `sum`; before and after (with differences highlighted) the application of the rename refactoring.

where the implementation is automatically generated from the proof. Consequently, the relationship of proof to actual implementation is not always clear: *there is an implicit trust that the implementation correctly models the formalism*. We take a different approach: producing a proof of soundness as part of the refactoring itself. In our case, we use Idris [1] to provide a *verified* refactoring framework for a subset of Haskell 98. Dependent types allow us to formally define the semantics of our refactorings and give soundness proofs as part of the refactoring implementations themselves. The inclusion of a refactoring is, by design, only possible when those proof obligations have been met. In other words, the proofs and the implementation are part of the same system such that the implementation directly depends upon the proofs, and vice versa. In our approach, *there is no implicit trust needed that the implementation correctly models the formalism* since the implementation is the formalism itself. This paper presents a dependently-typed representation of the syntax of a subset of Haskell 98, which is intended as a foundation for the implementation of refactorings. In order to facilitate soundness proofs, we encode the static semantics (in this paper we focus purely on the binding structure as our static semantics) of our Haskell 98 subset into its representation. This has the additional benefit of restricting the programs that are representable within the tool, thus preventing the production of ill-formed Haskell programs for free. We further demonstrate our approach by focussing on renaming as an example of a canonical and well-understood refactoring. Renaming refers to the problem of substituting a function or variable name such that all and only those occurrences that are bound by a particular definition are transformed, but that no other occurrences of the name are affected. Renaming must ensure that it introduces no capturing or shadowing violations; specifically, introducing a new name that already exists within the target scope or where the new name conflicts with any binding already present within the scope, resulting in the new name being captured in the result of the renaming; in our implementation, in the case of capturing violations, the refactoring simply returns the original program unchanged. Figure 1 gives an example of renaming, with the program before renaming on the left, and the renamed equivalent, on the right. We present both a formal specification of renaming over our Haskell 98 subset and an implementation of renaming that incorporates a proof of its adherence to its formal specification. In this paper we focus on structural equivalence – rather than functional equivalence – as a preliminary goal. Further work will extend upon the proofs given here to take into account the functional semantics of Haskell programs and their refactoring equivalence.

$$
\begin{array}{rcll}
p \in P & := & x_{\langle l,i \rangle} & \text{(Variable Patterns)} \\
& | & A_i\ \vec{p} & \text{(Constructor Patterns)} \\
e \in E & := & x_{\langle l,i \rangle} & \text{(Variable Exp.)} \\
& | & n & \text{(Integer Literal Exp.)} \\
& | & e_1\ e_2 & \text{(Application Exp.)} \\
& | & \backslash \vec{p}\ \texttt{->}\ e & (\lambda\text{-Exp.}) \\
c \in C & := & \vec{p} = e\ \texttt{where}\ \vec{d} & \text{(Function Clauses)} \\
d \in D & := & \texttt{def}\ f_{\langle l,i \rangle}\ \vec{c} & \text{(Function Decl.)} \\
& | & \texttt{def}\ p = e\ \texttt{where}\ \vec{d} & \text{(Pattern Decl.)} \\
m \in M & := & \texttt{mod}\ \vec{d} & \text{(Modules)}
\end{array}
$$

Figure 2: Hs98: a subset of Haskell 98.

2 A Subset of Haskell 98

We focus on the subset, Hs98, of the Haskell 98 standard given in Figure 2. In Hs98, a valid *program* is comprised of a single module, $m \in M$, where *modules* are defined as a vector of declarations, $\vec{d}$. Modules implicitly import a Prelude, i.e. a set of constructors and variables that are valid in the module. *Declarations* are either pattern or function declarations. Pattern declarations consist of a pattern, p, an expression, e, and a where-block, $\vec{d}$. Function declarations are comprised of the function name, $f_{\langle l,i \rangle}$, and a vector of clauses, $\vec{c}$. Declarations within the same vector are assumed to be defined mutually. A *pattern* is either a variable, $x_{\langle l,i \rangle}$, or a constructor pattern comprising a constructor, A_i, and a vector of patterns, $\vec{p}$. An *expression* may be: a variable, $x_{\langle l,i \rangle}$; an integer literal, n; an application, $e_1\ e_2$; or a λ, $\backslash \vec{p}$ `->` e. Whilst we provide only an abstract presentation here, the full paper will include the full Idris definitions for our syntactic constructs.

2.1 Variables and Constructors

For clarity of presentation we denote variables by $x_{\langle l,i \rangle}$ and constructors by A_i. We assume an infinite set of valid variable names, VarN, and use $f, x, y, \ldots$ to represent arbitrary variable names. A variable, $x_{\langle l,i \rangle}$, is a triple comprising a variable name, $x \in$ VarN, an identifier, $i \in \mathbb{N}$, and a level identifier, $l \in \mathbb{N}$. Identifiers disambiguate variables since variable names may be shadowed by declarations in where-blocks. Similarly, level identifiers group variables in order to enforce linearity (i.e. uniqueness) [6] of variable names that are declared in the same vector of patterns or declarations. As defined in the Haskell 98 Report [6], Haskell 98 has linear patterns, in the sense that variables within a set of patterns must be declared exactly once, which forbids function declarations in the form `f x x = e`. We assume a set of valid constructor names, CtrN, and use A and B to represent arbitrary constructor names. A constructor, A_i, is a constructor name, $A \in$ CtrN, indexed by an identifier, $i \in \mathbb{N}$. Identifiers are assumed to be unique within a given scope. Since type declarations are not included in Hs98, constructors are limited to those defined in the Prelude. All variables and constructors are *unqualified*, i.e. variables are not preceded by a module name.

2.2 Environments

Scoping rules via environments represent a significant part of our static semantics. Intuitively, we define

```
1  data EnvItem : HsNameKind -> Type where
2    MkEnvItem : (id : Nat) -> (na : Name k) -> EnvItem k
3
4  data ScopeLevel : Type where
5    MkScopeLevel : (lvl  : Nat)
6                -> (vars : Vect nvars (EnvItem Variable))
7                -> {lin  : UniqueNames vars}
8                -> ScopeLevel
9
10 data Env : Type where
11   MkEnv : (ctrs : Vect nctrs (EnvItem Constructor))
12        -> (lvls : Vect nlvls ScopeLevel)
13        -> {ok   : UniqueIds ctrs lvls}
14        -> Env
15
16 data EnvAddNewLevel : (env_init  : Env)
17                    -> (vars      : Vect n (HsNameTy Variable))
18                    -> (sameLevel : SameLevel vars)
19                    -> (env_new   : Env)
20                    -> Type where
21 ...
```

Listing 1: Definitions for environments, scope levels, and extending environments in Hs98.

an *environment*, $\sigma = (\vec{A}_i, \vec{\eta})$, to be a pair comprising a vector of constructors, $\vec{A}_i$, and a vector of *scope levels*, $\vec{\eta}$. In our AST, environments are defined by the type `Env` (Listing 1). In addition to $\vec{A}_i$ and $\vec{\eta}$, the sole constructor, (`MkEnv` $\vec{A}_i$ $\vec{\eta}$ `ok`), takes a proof, `ok`, that all identifiers across $\vec{A}_i$ and $\vec{\eta}$ are indeed unique. Splitting constructors and variables simplifies both proofs of membership and extending environments with new scope levels in our implementation. Moreover, we use vectors since this allows the implicit representation of the ordering of scope levels. Intuitively, a scope level, $\eta = (l, x_{\vec{\langle l,i \rangle}})$, is a pair comprising a level identifier, l, and a vector of variables, $x_{\vec{\langle l,i \rangle}}$, that are declared in the same pattern, vector of patterns, or where-block. Scope levels are formally defined by the type `ScopeLevel` in Listing 1, such that η corresponds to (`MkScopeLevel` l $x_{\vec{\langle l,i \rangle}}$ `lin`), where `lin` is a proof that all variable names in $x_{\vec{\langle l,i \rangle}}$ are distinct. Environments are *extended*, denoted $\eta \cup x_{\vec{\langle l,i \rangle}}$, by a vector of variables, $x_{\vec{\langle l,i \rangle}}$, when all variables in $x_{\vec{\langle l,i \rangle}}$ have the same level identifier. Extending environments is defined by `EnvAddNewLevel` in Listing 1. `EnvAddNewLevel` is indexed by two environments, `env_init` and `env_new`, a vector of variables, `vars` (denoted $x_{\vec{\langle l,i \rangle}}$), and a proof, `sl`, that all variables in $x_{\vec{\langle l,i \rangle}}$ (i.e. `vars`) have the same level identifier. When $x_{\vec{\langle l,i \rangle}}$ is empty, no new scope level is added to the environment, and thus the environment does not change (i.e. `env_init = env_new`). Otherwise, `env_new` is formed by prepending a new scope level to $\vec{\eta}$ in `env_init`, i.e.

```
env_new = MkEnv A⃗ᵢ (MkScopeLevel lv envitems :: η⃗)
```

where `env_init = (MkEnv` $\vec{A}_i$ $\vec{\eta}$`)`. Here, `envitems` is the result of transforming each variable in $x_{\vec{\langle l,i \rangle}}$ to environment items. In addition to `sl`, i.e. the proof that all variables in $x_{\vec{\langle l,i \rangle}}$ have the same level identifier, `EnvAddNewLevel` has three proof obligations:

1. that the level identifier for all variables in $x_{\vec{\langle l,i \rangle}}$ is fresh with respect to the level identifiers used in `lvls` (i.e. `isNewLevel`);
2. that no two variable names in `envitems` are the same; and

```
1  ⟨mod [
2    ⟨def sum⟨1,4⟩ [
3      ⟨⟨[n⟨2,5⟩, Nil1 []]⟩^σ1 = ⟨n⟨2,5⟩⟩^σ2 where []⟩^σ1,
4      ⟨⟨[n⟨2,5⟩, Cons0 [h⟨2,6⟩, t⟨2,7⟩]]⟩^σ1 = ⟨(+)⟨0,2⟩ h⟨2,6⟩ rest⟨3,11⟩⟩^σ3
5        where [
6          ⟨def ⟨[rest⟨3,11⟩]⟩^σ3 =
7            ⟨sumRest⟨3,10⟩ n⟨2,5⟩ t⟨2,7⟩⟩^σ3 where []⟩^σ3,
8          ⟨def sumRest⟨3,10⟩ [
9            ⟨⟨[n⟨4,12⟩, t⟨4,13⟩]⟩^σ3 =
10             ⟨sum⟨1,4⟩ n⟨4,12⟩ t⟨4,13⟩⟩^σ4  where []⟩^σ3
11       ]⟩^σ3
12     ]
13   ]⟩^σ1,
14   ⟨def [main⟨1,14⟩] = sum⟨1,4⟩ 0 (enumFromTo⟨0,3⟩ 1 4) where []⟩^σ1
15 ]⟩^σ0
```

Listing 2: The module `Sum` from Listing 1a in Hs98 with indexed environments.

3. that all identifiers for variables and constructors in the extended vector of scope levels are unique.

Since environments control when and where variables and constructors can occur in the AST, we index each declaration, expression, and pattern by an environment. Each pattern, p, expression, e, declaration, d, and function clause, c, is indexed by an environment, σ; denoted $\langle p \rangle^\sigma$, $\langle e \rangle^\sigma$, $\langle d \rangle^\sigma$, and $\langle c \rangle^\sigma$, respectively. Modules also have an environment, σ_0, representing the Prelude. Each declaration in $\vec{d}$ is indexed by the environment, σ, where $\sigma = \sigma_0 \cup x_{\vec{\langle l,i \rangle}}$, given the vector of variables, $x_{\vec{\langle l,i \rangle}}$, that are exposed in $\vec{d}$, i.e. function names and variables declared in pattern declarations. Similarly, in pattern declarations, σ is extended by the variables exposed by $\vec{d}$, and both e and declarations in $\vec{d}$ are indexed by the resulting σ_1. In function clauses, since where-blocks can shadow variables defined in patterns, σ is first extended with the variables declared in $\vec{p}$, and then extended with the variables exposed by $\vec{d}$. In λ-expressions, e is indexed by the environment, σ_1, that is formed by extending σ with the variables in $\vec{p}$. In all other cases, an environment is propagated, without change, to all subexpressions or subterms. When $x_{\langle l,i \rangle}$ occurs in an expression, $\langle e \rangle^\sigma$, $x_{\langle l,i \rangle}$ must be in σ, which we denote $x_{\langle l,i \rangle} \in \sigma$ and is defined by the type `(ElemEnv` σ $x_{\langle l,i \rangle}$`)`.

Example 2.1 (Sum with Environments)**.** *Listing 2 shows how we can represent the syntax of the module in Figure 1 with the following environments.*

$$
\begin{array}{lcl}
\sigma_0 & := & ([Cons_0, Nil_1], [(0, [(+)_{\langle 1,2 \rangle}, enumFromTo_{\langle 0,3 \rangle}])]) \\
\sigma_1 & := & \sigma_0 \cup [sum_{\langle 1,4 \rangle}, main_{\langle 1,14 \rangle}] \\
\sigma_2 & := & \sigma_1 \cup [n_{\langle 2,5 \rangle}] \\
\sigma_3 & := & \sigma_2 \cup [n_{\langle 2,5 \rangle}, h_{\langle 2,6 \rangle}, t_{\langle 2,7 \rangle}, rest_{\langle 3,11 \rangle}, sumRest_{\langle 3,10 \rangle}] \\
\sigma_4 & := & \sigma_3 \cup [n_{\langle 4,12 \rangle}, t_{\langle 4,13 \rangle}]
\end{array}
$$

Here, σ_0 is the Prelude environment, σ_1 is the environment for each declaration in the module, σ_2 is the environment for the RHS of the first clause of $sum_{\langle 1,4 \rangle}$, σ_3 is the environment for the RHS and where-block definitions of the second clause of $sum_{\langle 1,4 \rangle}$, and σ_4 is the environment for the RHS of $sumRest_{\langle 3,10 \rangle}$. For clarity, we do not annotate subterms when no change occurs to the environment nor do we change an environment when a where-block is empty, e.g. in $main_{\langle 1,4 \rangle}$. Additionally, we say $\langle \vec{\bullet} \rangle^\sigma$ to denote that all elements in $\vec{\bullet}$ are indexed by σ, where $\bullet$ stands for any term in Hs98.

```
1  renameModule : (lv : Nat) -> (id : Nat)
2              -> (oldName : HsNameTy Variable)
3              -> (newName : HsNameTy Variable)
4              -> (mod : HsModuleTy)
5              -> Maybe HsModuleTy
```

Listing 3: Type signature for implementation of renaming refactoring over modules in HS98.

```
1  renameExpr :  (lv, id : Nat)
2             -> (oldN, newN : Name Variable)
3             -> (env : Env)
4             -> (e : HsExpTy initEnv)
5             -> Maybe (HsExpTy env)
6  ...
7  renameExpr lv id oldN newN env (HsLambda ps e)
8    with (renamePatTy lv id oldN newN env ps)
9    | Nothing = Nothing
10   | Just ps'
11     with (decHsPatSecTy env ps')
12     | No np = Nothing
13     | Yes (Element env_e' psec') with (renameExpr lv id oldN newN env_e' e)
14       | Nothing = Nothing
15       | Just e' =
16         Just (HsLambda {env_e=env_e'} {psec=psec'} ps' e')
```

Listing 4: Partial definition for implementation of renaming refactoring; including λ and variable expressions in HS98.

3 Renaming and its Implementation

In this section, we give an overview of our implementation for renaming. Listing 3 gives the type signature for `renameModule`, which transforms a given module, $\langle m \rangle^{\sigma_0}$, into a new module, $\langle m' \rangle^{\sigma_0}$, such that all occurrences of a given variable, $x_{\langle l,i \rangle}$, are renamed to $y_{\langle l,i \rangle}$. Here, `lv` corresponds to l, `id` to i, `oldName` to x, `newName` to y, and `mod` to $\langle m \rangle^{\sigma_0}$. Intuitively, `renameModule` folds over $\langle m \rangle^{\sigma_0}$ such that all `HsNameTy` nodes that represent $x_{\langle l,i \rangle}$ in the AST of $\langle m \rangle^{\sigma_0}$ are replaced by an equivalent node representing $y_{\langle l,i \rangle}$ in the AST of $\langle m' \rangle^{\sigma_0}$, and that all environments, σ, in $\langle m \rangle^{\sigma_0}$ where $x_{\langle l,i \rangle} \in \sigma$ are transformed such that $y_{\langle l,i \rangle} \in \sigma$ in $\langle m' \rangle^{\sigma_0}$. Transforming $x_{\langle l,i \rangle}$ in some environment, σ, is a standard substitution operation and is denoted $\sigma[x_{\langle l,i \rangle} \to y_{\langle l,i \rangle}]$. In cases where renaming *fails*, e.g. when $y_{\langle l,i \rangle}$ will violate linearity, `renameModule` returns `Nothing`. In order to illustrate our implementation, we consider renaming over expressions. Listing 4 gives a partial definition of `renameExpr`, where we focus on λ-expressions. In addition to the expression being refactored, `e`, the variable to be renamed (comprising `lv`, `id`, and `oldN`), and the new name of the variable (`newN`), `renameExpr` takes as argument an environment, `env`. Any expression that is produced by `renameExpr` is indexed by `env`, where `env` represents $\sigma[\texttt{oldN}_{\langle \texttt{lv,id} \rangle} \to \texttt{newN}_{\langle \texttt{lv,id} \rangle}]$, and σ corresponds to `initEnv`, the (arbitrary) environment by which the original expression is indexed. λ-expressions are reconstructed in the `HsLambda` case. First, the patterns of the λ, `ps`, are transformed via `renamePatTy`. As with `renameHsNameTy`, `renamePatTy` only transforms a pattern when it represents $\texttt{oldN}_{\langle \texttt{lv,id} \rangle}$ and the resulting vector of patterns, `ps'`, is indexed by `env`. In order to recurse into `e`, `env` is extended by the variables in `ps'` via `decHsPatSecTy`. The result of `decHsPatSecTy` returns a dependent pair comprising a new environment, `env_e'`, and the

```
1  data DecldVectHsDeclTy : (lv,id : Nat) -> (x : Name Variable)
2                         -> (ds : Vect nds (HsDeclTy env dk))
3                         -> Type where
4    HereF : DecldVectHsDeclTy lv id x (HsFunBind (HsIdentVar lv id x) cs :: ds)
5    HereFInClauses : (there : DecldVectHsDeclTy lv id x cs)
6                  -> DecldVectHsDeclTy lv id x (HsFunBind (HsIdentVar k j z) cs :: ds)
7    ...
8
9  data DecldHsModuleTy : (lv,id : Nat) -> (x : Name Variable) -> (m : HsModuleTy)
10                      -> Type where
11   There : (p : DecldVectHsDeclTy lv id x ds) -> DecldHsModuleTy lv id x (HsModule env vs ds)
```

Listing 5: Definitions for the property that a variable is declared in Hs98 modules and function declarations.

proof, `psec'`, that `env_e'` correctly extends `env` by the variables in `ps`. Finally, we recurse into `e` to refactor the body of the λ. The result is a reconstruction of the original λ where it is now indexed by `env` and any occurrences of $\texttt{oldN}_{\langle \texttt{lv,id} \rangle}$ in `ps` and `e` have been transformed. Modules, declarations, function clauses, and patterns are all refactored similarly.

4 Soundness of Renaming

In this section, we define what it means for an implementation of renaming to be sound in terms of the transformations effected and give a proof that the implementation described in Section 3 is sound. We show structural equivalence, analogous to proving equivalence to a de Bruijn index representation [3]. Intuitively, given two modules, $\langle m \rangle^{\sigma_0}$ and $\langle m' \rangle^{\sigma_0}$, where $\langle m' \rangle^{\sigma_0}$ is the result of renaming some variable $x_{\langle l,i \rangle}$ in $\langle m \rangle^{\sigma_0}$ to $y_{\langle l,i \rangle}$, a renaming refactoring is sound when we can prove that $x_{\langle l,i \rangle}$ is defined in $\langle m \rangle^{\sigma_0}$, that $y_{\langle l,i \rangle}$ is defined in $\langle m' \rangle^{\sigma_0}$, and that the structures of $\langle m \rangle^{\sigma_0}$ and $\langle m' \rangle^{\sigma_0}$ are equivalent.

4.1 Variable Declaration

Our soundness definition requires proofs that $x_{\langle l,i \rangle}$ is declared in $\langle m \rangle^{\sigma_0}$ and that $y_{\langle l,i \rangle}$ is declared in $\langle m' \rangle^{\sigma_0}$. This ensures that the eponymous renaming does indeed occur as a result of the refactoring. Whilst renaming could be applied to a module that does not contain the declaration of $x_{\langle l,i \rangle}$, and would return the module unchanged, we elide this case here because we are specifically interested in positive transformations. To aid our presentation, we use set inclusion to denote this property; e.g. $w_{\langle k,j \rangle} \in \langle m \rangle^{\sigma_0}$ denotes that $w_{\langle k,j \rangle}$ is declared in $\langle m \rangle^{\sigma_0}$. We say that $x_{\langle l,i \rangle} \in \langle m \rangle^{\sigma_0}$ if there exists some pattern variable or function name $w_{\langle k,j \rangle} \in \langle m \rangle^{\sigma_0}$ such that $x = w \wedge l = k \wedge i = j$. In our implementation, we define `DecldHsModuleTy` (Listing 5) as the family of types corresponding to $x_{\langle l,i \rangle} \in \langle m \rangle^{\sigma_0}$ for a given $x_{\langle l,i \rangle}$ and $\langle m \rangle^{\sigma_0}$. In Listing 5, `lv` corresponds to l, `id` to i, `x` to x, and `m` to $\langle m \rangle^{\sigma_0}$. `DecldHsModule` has a single constructor, `There`, which takes a proof, `later`, that $x_{\langle l,i \rangle}$ is declared in one of the declarations in `ds` at the top level of the module. `DecldVectHsDeclTy` is defined similarly: `HereF` states that $x_{\langle l,i \rangle}$ is declared as the name of the function declaration at the head of `ds`; and `HereFInClauses` states that $x_{\langle l,i \rangle}$ is declared in a pattern or where-block in one of the clauses that comprise the function at the head of `ds`. Definitions and decision procedures for expressions, patterns, and the remaining declarations follow similarly.

```
1  data RenamePrf : (lv,id : Nat) -> (oldN,newN : Name Variable)
2                -> (m1 : HsModuleTy)
3                -> (rn : Maybe
4                          (DecldHsModuleTy lvl id oldN m1,
5                           (m2 : HsModuleTy **
6                             (StructEq m1 m2, DecldHsModuleTy lvl id newN m2))))
7                -> Type where
8    RenameFail : RenamePrf lvl id oldN newN m1 sPrf Nothing
9    RenameSucc : (oldD : DecldHsModuleTy lvl id oldN m1)
10             -> (newD : DecldHsModuleTy lvl id newN m2)
11             -> (eq : StructEq m1 m2)
12             -> RenamePrf lvl id oldN newN m1 sPrf (Just (oldD, (m2 ** (eq,newD))))
```

Listing 6: Definition of soundness for the renaming refactoring implementation, `rename`, over Hs98.

4.2 Structural Simplification

In order to prove that two modules, $\langle m \rangle^{\sigma_0}$ and $\langle m' \rangle^{\sigma_0}$ are structurally equivalent, we first define the process of *structural simplification*. Intuitively, structural simplification is defined as a surjection from a module, $\langle m \rangle^{\sigma_0}$, to an equivalent but simplified module, $\mu \in M'$. In our presentation, structural simplification is denoted $\langle m \rangle^{\sigma_0} \Downarrow \mu$. Our simplified syntax can be seen as being comprised of the syntax in Figure 2 sans environments and both variable and constructor names. Consequently, variables now comprise an identifier indexed by its level, i_l, where i is the variable identifier and l is the level identifier; we use this indexed notation to avoid ambiguity with standard variables from the enriched syntax. Similarly, constructors are represented by an identifier, i, only. This affects variable patterns in P', variable expressions in E', and function declarations in D'. All other syntactic constructs are unchanged. In our implementation, the simplified AST is equivalent to our standard Hs98 AST but stripped of all static semantic terms (i.e. proofs) and variable names. Intuitively, two modules are *structurally equivalent* when their structural simplifications are the same. We denote structural equivalence by $\langle m \rangle^{\sigma_0} \equiv \langle m' \rangle^{\sigma_0}$, where $\langle m \rangle^{\sigma_0}$ and $\langle m' \rangle^{\sigma_0}$ are modules s.t. $\mu = \mu'$, given that $\langle m \rangle^{\sigma_0} \Downarrow \mu$ and $\langle m' \rangle^{\sigma_0} \Downarrow \mu'$ hold and where $\mu, \mu' \in M'$.

4.3 Soundness of Renaming Implementations

We give the definition for soundness of renaming by the type `RenamePrf` in Listing 6. `RenamePrf` is indexed by: the variable to be renamed, $x_{\langle l,i \rangle}$; its new name, y; a module, $\langle m \rangle^{\sigma_0}$; and the result of the renaming implementation, `rename` (Listing 8). In Listing 6, `lv` corresponds to l, `id` to i, `oldN` to x, `newN` to y, `m1` to $\langle m \rangle^{\sigma_0}$, and `rn` to the return type of `rename`. Since the renaming may fail, `RenamePrf` has two constructors: `RenameFail` and `RenameSucc`. The failure case, `RenameFail`, is trivial since we are only concerned that a successful application of `rename` transforms $\langle m \rangle^{\sigma_0}$ correctly. Intuitively, a successful application of `rename` to $\langle m \rangle^{\sigma_0}$ will produce the module $\langle m' \rangle^{\sigma_0}$ such that $x_{\langle l,i \rangle} \in \langle m \rangle^{\sigma_0}$, $y_{\langle l,i \rangle} \in \langle m' \rangle^{\sigma_0}$, and $\langle m \rangle^{\sigma_0} \equiv \langle m' \rangle^{\sigma_0}$ all hold. In `RenamePrf`, these correspond to `oldD`, `newD`, and `eq`, respectively. In order to prove that our implementation of renaming conforms to `RenamePrf`, we define `prfRename` in Listing 7. `prfRename` states that `rename` is sound w.r.t. `RenamePrf` for all variables, $x_{\langle l,i \rangle}$, names, y, and modules, $\langle m \rangle^{\sigma_0}$. `rename` is defined as an interface to `renameModule` in Listing 3 from Section 3. Listing 8 gives the definition for both `rename` and `rename'`. `rename` takes $x_{\langle l,i \rangle}$ and $\langle m \rangle^{\sigma_0}$, and returns either `Nothing`, indicating failure, or the refactored module $\langle m' \rangle^{\sigma_0}$ with proofs of

```
1  prfRename : (lv, id : Nat)
2            -> (oldN, newN : Name Variable)
3            -> (m1   : HsModuleTy)
4            -> RenamePrf lv id oldN newN m1 m1sP (rename lvl id oldN newN m1)
5  prfRename lv id oldN newN m1 with (rename lvl id oldN newN m1)
6    | Nothing = RenameFail
7    | Just (oldD, (m2 ** (eq, newD))) = RenameSucc oldD newD eq
```

Listing 7: Function representing the proof that for all variables, names, and modules, `rename` conforms to our soundness definition in Listing 6.

```
1  rename' : (lv, id : Nat)
2          -> (oldN, newN : HsNameTy Variable)
3          -> (m1 : HsModuleTy)
4          -> Maybe (m2 : HsModuleTy ** StructEq m1 m2)
5
6  rename : (lv,id : Nat) -> (oldN, newN : Name Variable)
7        -> (m1 : HsModuleTy)
8        -> Maybe (DecldHsModuleTy lv id oldN m1,
9                  (m2 : HsModuleTy ** (StructEq m1 m2, DecldHsModuleTy lv id newN m2)))
```

Listing 8: The interface function, `rename`, and its helper function, `rename'`, for the implementation of renaming over Hs98.

$x_{\langle l,i\rangle} \in \langle m\rangle^{\sigma_0}$, $y_{\langle l,i\rangle} \in \langle m'\rangle^{\sigma_0}$, and $\langle m\rangle^{\sigma_0} \equiv \langle m'\rangle^{\sigma_0}$. The proofs of both $x_{\langle l,i\rangle} \in \langle m\rangle^{\sigma_0}$ and $y_{\langle l,i\rangle} \in \langle m'\rangle^{\sigma_0}$ are obtained by calling `isDecldHsModuleTy` and assigned to `oldD` and `newD`, respectively. `rename'` calls `renameModule`, producing $\langle m'\rangle^{\sigma_0}$, and constructs the proof of $\langle m\rangle^{\sigma_0} \equiv \langle m'\rangle^{\sigma_0}$ by structurally simplifying both modules and determining whether they are propositionally equal via the decision procedure, `decEq`. Since both `prfRename` and `rename` are total functions that pass the type-checker, we can conclude that `rename` will always produce a value of type `RenamePrf` and is thus sound w.r.t. our definition.

5 Conclusions and Future Work

In this paper we introduced a refactoring framework for a subset of Haskell 98, where refactorings are implemented in Idris using dependent types. Within this framework, refactoring implementations include proofs of their correctness. This ensures that if a refactoring is able to successfully transform a given module, the resulting module conforms to the refactoring's formal definition. We demonstrated our framework on the canonical refactoring of renaming, with a renaming implementation and a verified theorem of its soundness. In the future, we will extend this work in a number of directions. We plan to write a denotational semantics of Hs98 and prove functional equivalence over the denotations. This will be needed when we consider refactorings where structural equivalence alone may not be sufficient to show soundness. We also plan to extend our framework to allow compositions of refactorings. By composing refactorings together, and composing their proofs, we can show the refactoring composition is correct by virtue of the composition of its proofs. We will also extend our framework with more Haskell refactorings from the traditional set (lifting definitions, adding arguments, etc.) and to refactorings that affect parts of a program's types. This will require extending our static semantics to model aspects of Haskell's type system. Doing so would allow both refactorings that transform Haskell programs in a type-

safe way and those that transform aspects of a program's type structure. Examples include: λ-lifting, where changing the scope of a definition (i.e. promoting a let declaration to a where; see [2] for details) could introduce monomorphism restrictions on definitions that are inferred to be polymorphic in some of their arguments; data type refactorings; and refactorings that aim to generalise a function making it more polymorphic, say, and capturing this in its type signature. We will also extend our subset of Haskell to include, e.g., multiple modules, pattern matching, type declarations and data types, let-expressions and monadic computations, thus demonstrating applicability of our techniques to more real-world Haskell use-cases.

References

[1] Edwin Brady (2017): *Type-driven development with Idris*. Manning Publications Co.

[2] Christopher Brown (2008): *Tool Support for Refactoring Haskell Programs*. Ph.D. thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK.

[3] N.G de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. Indagationes Mathematicae (Proceedings) 75(5), pp. 381 – 392, doi:https://doi.org/10.1016/1385-7258(72)90034-0. Available at `http://www.sciencedirect.com/science/article/pii/1385725872900340`.

[4] Martin Fowler (1999): *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., USA.

[5] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz & Darko Marinov (2013): *Systematic Testing of Refactoring Engines on Real Software Projects*. In Giuseppe Castagna, editor: *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, *Lecture Notes in Computer Science* 7920, Springer, pp. 629–653, doi:10.1007/978-3-642-39038-8_26. Available at `https://doi.org/10.1007/978-3-642-39038-8_26`.

[6] Simon L. Peyton Jones (2003): *Haskell 98: Lexical Structure*. *J. Funct. Program.* 13(1), pp. 7–16, doi:10.1017/S0956796803000418. Available at `https://doi.org/10.1017/S0956796803000418`.

[7] Huiqing Li & Simon Thompson (2005): *Formalisation of Haskell Refactorings*. In Marko van Eekelen & Kevin Hammond, editors: *Trends in Functional Programming*, pp. 182–196. Available at `http://www.cs.kent.ac.uk/pubs/2005/2250`.

[8] Tom Mens & Tom Tourwé (2004): *A Survey of Software Refactoring*. *IEEE Trans. Softw. Eng.* 30(2), p. 126–139, doi:10.1109/TSE.2004.1265817. Available at `https://doi.org/10.1109/TSE.2004.1265817`.

[9] Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson & Scott Owens (2019): *Characterising Renaming within OCaml's Module System: Theory and Implementation*. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, Association for Computing Machinery, New York, NY, USA, p. 950–965, doi:10.1145/3314221.3314600. Available at `https://doi.org/10.1145/3314221.3314600`.

[10] Nik Sultana & Simon Thompson (2008): *Mechanical Verification of Refactorings*. In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '08, Association for Computing Machinery, New York, NY, USA, p. 51–60, doi:10.1145/1328408.1328417. Available at `https://doi.org/10.1145/1328408.1328417`.

[11] Simon J. Thompson (2004): *Refactoring Functional Programs*. In Varmo Vene & Tarmo Uustalu, editors: *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures*, *Lecture Notes in Computer Science* 3622, Springer, pp. 331–357, doi:10.1007/11546382_9. Available at `https://doi.org/10.1007/11546382_9`.