

Structural Refactorings for Exploring Dependently-Typed Programming

Adam D. Barwell¹, Christopher Brown¹, Mun See Chang¹, Constantine
Theocharis¹, and Simon Thompson²

¹ School of Computer Science, University of St Andrews, Scotland, UK
`{adb23}{cmb21}{msc2}{kt81}@st-andrews.ac.uk`

² University of Kent, UK and Eötvös Loránd University, Hungary.
`S.J.Thompson@kent.ac.uk`

Abstract. In this (draft) paper we propose a number of new refactorings for dependently-typed programs. We first introduce a small dependently-typed functional programming language, called *Fluid*, with inductive types. We then introduce a number of refactorings for Fluid that introduce and refine dependent types. The refactorings are then demonstrated over a small expression language example.

Keywords: Refactoring · Dependent-types · Program Transformation.

1 Introduction

Dependently-typed programming languages represent a practical approach to verification in which both specifications and proofs of correctness are expressed directly in programs. These languages, including Idris [Bra21] and Agda [Nor08], provide strong typing mechanisms that permit logical properties to be expressed as types, so that well-typed programs will be proofs of those properties, meaning that correctness guarantees become an intrinsic part of an implementation, making programs that are “correct by construction”.

Despite this advantage, developing and maintaining dependently-typed programs that leverage these features can represent a significant challenge. Alongside the existing tasks of designing, developing, and testing software, the programmer is now presented with the additional task of defining correctness predicates and *enriching* a program to demonstrate conformity with its correctness criteria. An exploratory approach naturally presents itself, where the programmer first develops a simply-typed prototype implementation (i.e. without dependent-types), and then *enriches* that implementation with correctness guarantees. This enrichment is a *refactoring* [Opd92] process.

Refactoring techniques aim to improve the design and maintainability of software without changing (external) functionality [Fow99]. Although much refactoring is done manually, there has been substantial interest in building tools that support the process. Refactoring tools that enable the *semi-automatic*, programmer-guided, application of refactorings have been developed for a range

of languages [Bro08, Met, LT12], and have been integrated into IDEs [Fou] and editors via the Language Server Protocol [lan]. The principal advantages of refactoring tools are efficiency, and the avoidance of error. Efficiency is achieved via the automatic effecting of transformations, potentially across multiple definitions and files, and errors are avoided by the consistent and correct application of those transformations, which have themselves been demonstrated to be correct.

Since programs that fully leverage dependent types often exhibit tight coupling between types and values, refactoring such programs often necessitates changes that propagate across much of the codebase. Despite the obvious advantages of mechanising this process, there is currently limited refactoring tool support for dependently-typed languages.

Existing tool support focuses on proof-repair systems [RYLG19, Wil20] that automatically migrate code to a new but structurally-enriched type, such as the transition from lists to vectors, by means of *ornaments* [McB10]. Although these techniques represent a key component of type enrichment refactoring, they do not encompass its entirety.

In this paper we propose a complementary approach to refactoring dependently-typed programs, building a set of refactorings by generalising the steps in an expression evaluator case study. The refactorings presented are designed to be used by the programmer in an exploratory way to transform their simply-typed program into a dependently-typed program in an iterative manner. In an example like this, where refactorings are applied one after another, it is often the case that a refactoring step will arise as the result of an earlier refactoring step, and we will see examples of that here.

In the final version of the paper, we aim to make the following contributions.

1. We will introduce a small dependently-typed functional language, named Fluid, similar to Idris and based on a dependently-typed lambda calculus with inductive types.
2. We will introduce a number of new refactorings over Fluid for dependently-typed programming.
3. We will provide implementations of these refactorings in a new proof of concept refactoring tool.
4. We will demonstrate these refactorings in a number of examples, showing improved safety properties over the original programs.

2 Fluid, a dependently-typed programming language

Fluid is a small dependently-typed programming language with inductive data types. The purpose of Fluid is to provide a test-bed for automatic transformations on dependently-typed programs. Its syntax closely resembles that of Idris or Agda, but without elaborate features such as modules, mutually recursive definitions, or implicit parameters.

We give the syntax of Fluid in Figure 1. A Fluid program is a sequence of declarations and type definitions. A declaration D is a value bound to a global

name v . Function declarations comprise a sequence of clauses that pattern match on the arguments of the function. Clauses that match on a set of incompatible patterns can be marked as `impossible`.

Program $P ::= (D \mid T)^*$	Term $t ::=$
Name v	$\mid (v : t_1) \rightarrow t_2$ (function type)
Declaration $D ::=$	$\mid (v : t_1) ** t_2$ (pair type)
$v : t$	$\mid \text{Type}$ (universe type)
$C[v]^*$	$\mid v$ (name)
Clause $C[v] ::=$	$\mid \backslash v \Rightarrow t$ (function)
$\mid v \ t_1 \dots t_n = t$	$\mid (t_1, t_2)$ (pair)
$\mid v \ t_1 \dots t_n \text{ impossible}$	$\mid t_1 \ t_2$ (application)
Type definition $T ::=$	$\mid \text{case } t \text{ of } t_1 \Rightarrow t'_1 \dots t_n \Rightarrow t'_n$ (case expression)
$\text{data } v : t \text{ where}$	$\mid _$ (wildcard pattern/infer term)
$v_1 : t_1 \mid \dots \mid v_n : t_n$	$\mid ?v$ (named hole)

Fig. 1: The syntax of Fluid as a recursive grammar.

Type definitions in Fluid are similar to those in Agda or Idris, but without the ability to define mutually inductive types or implicit parameters. Given these minor restrictions, any indexed inductive data type can be defined in Fluid. For clarity of presentation, Fluid does not support universe levels or universe polymorphism, unlike systems such as the Calculus of Inductive Constructions (CIC) [PM15]. These features are orthogonal to the transformations we are interested in.

Although it can be defined as an inductive type, for convenience, Fluid supports the dependent pair type as a first-class construct due to its extensive use in transformations. Fluid also supports named holes: in addition to being a useful tool when writing programs, holes are essential for some transformations that require additional information to be provided by the user. The examples in this paper sometimes use `if` expressions, which are not present in Figure 1, but are syntactic sugar for `case` expressions on a defined `Bool` type. Similarly, the symbols `_::_` and `[]` are used for the `List` and `Vect` constructors.

For brevity, we omit the typing rules of Fluid; they closely mirror the standard typing rules of CIC [PM15], but without consideration for universe levels or a distinction between `Type` and `Prop`.³ Full rules will be given in the final version of the paper.

³ A consequence of the lack of universe levels is that `Type : Type` is a valid typing judgement in Fluid. This is known to cause inconsistencies [Hur95], but they do not affect the approach that we present here.

3 Refactoring Specifications

In this section, we describe a new catalogue of refactorings for dependently-typed programs, which are motivated by the exploration of a case study in Section 4. This means that the refactorings have been designed to be exploratory in nature, allowing the programmer to refine their program and types by a series of steps. Each refactoring step often arises as the result of earlier refactoring steps, and we envisage that the refactorings will typically be used as a sequence of transformation steps, rather than as individual refactorings.

We illustrate each refactoring by giving examples representing code before and after their application, along with high level descriptions of the pre-conditions needed. In many cases the inverse transformations are possible with stronger preconditions, and are currently the target of future work. The refactorings presented here will be fully implemented in the final version of the paper.

3.1 Adding an index to a data type

Adding an index to a data type allows the data type to be parameterised by a type at its use. All references to the type need to be updated to take into account the newly added index.

<pre> 1 data Data1 : Type where 2 C1 : (a: Type) -> Nat -> 3 Data1 -> Data1 4 C2 : Nat -> Data1 5 C3 : Data1 </pre>	<pre> 1 data Data1 : List Nat -> 2 Type where 3 C1 : (ind1 : List Nat) -> 4 (ind2 : List Nat) -> 5 (a: Type) -> Nat -> 6 Data1 ind1 -> 7 Data1 ind2 8 C2 : (ind : List Nat) -> 9 Nat -> Data1 ind 10 C3 : (ind : List Nat) -> 11 Data1 ind </pre>
(a) Before	After

Fig. 2: Adding an index in a data type.

Example Figure 2 shows an example of an index being added to a data type, of type `List Nat`. A new parameter to each constructor is added to capture the index. In constructors with inductive parameters, a new parameter is added for each inductive step. For example, a new parameter, `ind2` of type `List Nat` has been added to constructor `C1` to capture the index in the type overall, while an additional parameter `ind1` of the same type has also been added to capture the index in the inductive parameter. All new parameters are given fresh names.

<pre> 1 f : Nat -> Data1 2 f n = C2 n </pre>	<pre> 1 f : (ind : List Nat) -> Nat -> 2 Data1 ind 3 f ind n = C2 ind n </pre>
(a) Before	After

Fig. 3: Adding an index in a function.

Updating Functions All functions defined over `Data1` must be updated to include the new index if being added. Figure 3 gives an example of such a function that is updated. A new parameter is introduced to capture the index argument for the data type. All clauses are updated to add a new argument. Any references to the constructors of `Data1` are updated to pass in the new index parameters. Any recursive calls are updated to pass the indexed parameters. The refactoring must also update all the call sites of `f` to pass in a proof obligation, this is achieved by transforming the call sites to pass a named hole. This hole can then be refined into a proof obligation through another series of refactoring steps and user intervention.

3.2 Specialising an index in a data type

Specialising an index in a data type allows a specified constructor to pass a specialised index to the data type.

<pre> 1 data Data1 : List Nat -> Type where 2 C1 : (ind1 : List Nat) -> 3 (ind2 : List Nat) -> 4 (a: Type) -> Nat -> 5 Data1 ind1 -> Data1 ind1 6 C2 : (ind : List Nat) -> 7 Nat -> Data1 ind 8 C3 : (ind : List Nat) -> 9 Data1 ind </pre>	<pre> 1 data Data1 : List Nat -> Type where 2 C1 : (ind1 : List Nat) -> 3 (ind2 : List Nat) -> 4 (a: Type) -> Nat -> 5 Data1 ind1 -> Data1 ind1 6 C2 : (ind : List Nat) -> 7 Nat -> Data1 [] 8 C3 : (ind : List Nat) -> 9 Data1 ind </pre>
(a) Before	After

Fig. 4: Specialising an index for a constructor.

Example Figure 4 specialises a constructor by replacing an indexed parameter with a value. In the type `Data1`, the constructor `C2` has been specialised to replace the `Data1` index, `ind` with an empty list, `[]`, therefore specialising the use of the constructor. The parameter for the unused index is retained as there is a separate refactoring to remove it.

3.3 Relating constructor parameters

Relating constructor parameters allows a new parameter to be inserted in a constructor to relate some parameters via a dependent type. This type is already supplied and known prior to the refactoring. When introducing the relation, the relevant parameters to the relation should already be present as parameters to the constructor.

<pre> 1 data Data1 : List Nat -> Type where 2 C1 : (ind : List Nat) -> 3 (n : Nat) -> Data1 ind 4 C2 : (ind : List Nat) -> 5 Data1 ind 6 C3 : (ind : List Nat) -> 7 Data1 ind </pre>	<pre> 1 data Data1 : List Nat -> Type where 2 C1 : (ind : List Nat) -> 3 (n : Nat) -> Elem n ind 4 -> Data1 ind 5 C2 : (ind : List Nat) -> 6 Data1 ind 7 C3 : (ind : List Nat) -> 8 Data1 ind </pre>
(a) Before	After

Fig. 5: Relating constructor parameters.

Example In Figure 5, introducing a relation between the two parameters of constructor **C1** is done by inserting a new parameter of type **Elem n ind**, where **n** is the second parameter to **C1** (which is of type **Nat**) and **ind** is the first parameter to **C1** (which is of type **List Nat**). The type **Elem** is provided to the refactoring by the programmer. All pattern matches over the constructor **C1** are modified to introduce a variable in the relation position. Any recursive calls using the constructor **C1** are also transformed to include the proof if one can be inferred from the assumptions in the position of the parameter, otherwise a hole is introduced, allowing the programmer to complete it later.

3.4 Unifying data type indices

Unifying data type indices replaces the indices of each occurrence of the data type being defined, with a single index, in a constructor.

Example Figure 6 demonstrates an example of unifying an index. In the case of constructor **C1** we have two situations where we populate the index for the type **Data1** with the variables **ind1** and **ind2**. In the *After* case, we remove one of the index variables, and populate all index positions with the same variable **ind1**, and remove the redundant index variables from the constructor parameters.

<pre> 1 data Data1 : List Nat -> Type where 2 C1 : (ind1 : List Nat) -> 3 (ind2 : List Nat) -> 4 (a: Type) -> Nat -> 5 Data1 ind1 -> Data1 ind2 6 C2 : (ind : List Nat) -> 7 Nat -> Data1 ind 8 C3 : (ind : List Nat) -> 9 Data1 ind </pre>	<pre> 1 data Data1 : List Nat -> Type where 2 C1 : (ind1 : List Nat) -> 3 (a: Type) -> Nat -> 4 Data1 ind1 -> Data1 ind1 5 C2 : (ind : List Nat) -> 6 Nat -> Data1 ind 7 C3 : (ind : List Nat) -> 8 Data1 ind </pre>
(a) Before	After

Fig. 6: Unifying data type indices.

3.5 Relating function parameters

Similarly to *Relating constructor parameters* (Section 3.3), a dependent type can be inserted as a new parameter to a function that relates a number of the other function parameters. The aim of this refactoring is to constrain the valid set of input values to the ones that can satisfy the added relation. If the arguments for the added relation are not already supplied by the function, new parameters satisfying the missing arguments will also be added. The function is updated to introduce pattern matching variables and possible holes in the recursive cases (if the type unification system cannot resolve the proof of the relation).

<pre> 1 f : (l : List Nat) -> (x : Nat) 2 -> Maybe Nat 3 f l x = ... </pre>	<pre> 1 f : (l : List Nat) -> (x : Nat) 2 -> Elem x l -> Maybe Nat 3 f l x p = ... </pre>
(a) Before	After

Fig. 7: Relating function parameters.

Example In Figure 7, we introduce a new parameter of type `Elem Nat (List Nat)` to the function `f`. Here `x` is used to provide the type of `Nat` and `l` is used for the type of `List Nat` (these are prompted by the user). All clauses over `f` are updated to include a pattern variable representing the inserted parameter.

3.6 Expanding a pattern variable

Expanding a pattern variable replaces clauses containing a specified pattern variable in a function with new clauses that enumerate the possible patterns for that variable, to one level of depth.

<pre> 1 f : (l : List Nat) -> (x : Nat) 2 -> Elem x l -> Maybe Nat 3 f l x p = Just x </pre>	<pre> 1 f : (l : List Nat) -> (x : Nat) 2 -> Elem x l -> Maybe Nat 3 f l x Here = Just x 4 f l x (There p) = Just x </pre>
(a) Before	After

Fig. 8: Expanding pattern matching for an outer variable.

<pre> 1 f : (l : List Nat) -> (x : Nat) 2 -> Elem x l -> Maybe Nat 3 f l x Here = Just x 4 f l x (There p) = Just x </pre>	<pre> 1 f : (l : List Nat) -> (x : Nat) 2 -> Elem x l -> Maybe Nat 3 f (x::rest) x Here = Just x 4 f (l::rest) x (There p) = Just x </pre>
(a) Before	After

Fig. 9: Expanding pattern matching for an inner variable.

Example Figure 8 expands the parameter argument `p` to pattern match over constructors of the `Elem` type. This results in two clauses for both the `Here` and `There` case. A further application of this refactoring on the same function is illustrated in Figure 9 where the parameter `l` is then expanded. Note here that in the first clause the head of the list must be `x` to match the `Elem` proof `Here`. All cases matching `[]` are impossible due to `Elem` relating both `l` and `x`, and are therefore omitted here.

3.7 Introducing a substitution via equality

If the programmer knows that two parameters to a function are equal, then one can be substituted with the other, and the redundant parameter can be removed. This refactoring is analogous to unifying data type indices (Section 3.4) and is particularly useful when two specific variables need to be related in order to form relevant proof obligations.

Example In Figure 10, the refactoring, given the assumption `l2 = l1` from the programmer, replaces occurrences of `l2` with `l1`. This means that the `Elem` parameter indexes over `l1` instead of `l2` (which is removed). The pattern matches are also subsequently changed: the variable `l2` is removed from all clauses, and the pattern matching is updated to reflect the substitution. In the example in Figure 10, the refactoring performs further adjustments to take into account the equality and, in the first clause, the `Elem` proof that `x` must be the lead of `l1`.

3.8 Eliminating tautologies in `case` expressions

Pattern matching on parameters to functions can often result in a unification of variables in the parameters to the function. This can have the side effect of also

<pre> 1 f : (l1 : List Nat) -> 2 (l2 : List Nat) -> 3 Data1 l1 -> (x : Nat) -> 4 Elem x l2 -> Maybe Nat 5 f l1 (x::l2) (C l1) x Here = Just x 6 f l1 l2 (C l1) x (There p) = Just x </pre>	<pre> 1 f : (l1 : List Nat) -> 2 Data1 l1 -> 3 (x : Nat) -> Elem x l1 -> 4 Maybe Nat 5 f (x::l1) (C (x::l1)) x Here = Just x 6 f l1 (C l1) x (There p) = Just x </pre>
(a) Before	After

Fig. 10: Substituting l2 for l1, assuming that l2 = l1.

unifying variables in the right-hand-side of the function. One possible result is that `case` expressions can have tautologies introduced.

<pre> 1 f : (l1 : List Nat) -> Data1 l1 -> (x : Nat) -> Elem x l1 -> Maybe Nat 2 f (_::l1) (C (_::l1)) x Here = 3 if x == x then Just x 4 else Nothing 5 f l1 (C l1) x (There p) = Just x </pre>	<pre> 1 f : (l1 : List Nat) -> Data1 l1 -> (x : Nat) -> Elem x l1 -> Maybe Nat 2 f (_::l1) (C (_::l1)) x Here = Just x 3 f l1 (C l1) x (There p) = Just x </pre>
(a) Before	After

Fig. 11: Eliminating the tautology `x == x` in an `if` expression.

Example Figure 11 illustrates a tautology in the `if` expression of the right-hand-side of the first clause for `f`. Here, we have an `if` expression with the tautologous condition `x == x`. As this always evaluates to `True`, the refactoring transforms it into the `then` branch. This transformation can also work for more general `case` expressions, as shown in in Figure 12. Case expressions whose condition is structurally identical to a single branch can be substituted for the code inside that branch, with the pattern variables substituted with the corresponding terms from the condition.

Non-trivial tautologies in `case/if` expressions—such as in Figure 11—need to be identified by the user, but structurally identical constructors in the expression and in a branch can be identified automatically like in Figure 12.

3.9 Eliminating Maybe

Eliminating a `Maybe` in the return type of a function is possible if there are no `Nothing` cases. Eliminating the `Nothing` cases is possible by first transforming

<pre> 1 g : (n : Nat) -> Nat 2 g n = case Just (S n) of 3 Just x => S x 4 Nothing => n </pre>	<pre> 1 g : (n : Nat) -> Nat 2 g n = S (S n) </pre>
(a) Before	After

Fig. 12: Eliminating a needless `case` expression.

these cases in a way that makes them `impossible`. The `Maybe` can then be eliminated by removing the `Just` constructs.

<pre> 1 f : (l1 : List Nat) -> 2 Data1 l1 -> 3 (x : Nat) -> Elem x l1 -> 4 Maybe Nat 5 f [] (C []) x Here impossible 6 f (y::ys) (C (y::ys)) x impossible 7 f (x::l1) (C (x::l1)) x Here = Just x 8 f l1 (C l1) x (There p) = Just x </pre>	<pre> 1 f : (l1 : List Nat) -> 2 Data1 l1 -> 3 (x : Nat) -> Elem x l1 -> 4 Nat 5 f [] (C []) x Here impossible 6 f (y::ys) (C (y::ys)) x impossible 7 f (x::l1) (C (x::l1)) x Here = x 8 f l1 (C l1) x (There p) = x </pre>
(a) Before	After

Fig. 13: Eliminating a `Maybe` type.

Example In Figure 13, the return type of the function (originally `Maybe Nat`) is transformed into `Nat`. The clauses which return `Just x` are transformed into `x`.

4 A Language Evaluator

In this section we demonstrate the refactorings from Section 3 on an evaluator for a small expression language. The use case is implemented in Fluid which reads and behaves very similarly to Idris. For brevity, the examples make use of implicit parameters, even though Fluid only supports explicit parameters.

The section proceeds as a programmer would explore and enrich a program to make it more expressive and dependently-typed, applying refactoring steps as they go. Figure 14 shows the basic setup of the language, with a data type, `Expr` representing numbers, variables and addition. An evaluator evaluates expressions by looking up variables in an environment via the `lookupVar` function, otherwise numbers are evaluated as literals, and addition is evaluated by adding the evaluation of both of its operands. The goal here is to make use of dependent-types

```

1  data Expr : Type where
2    Num : Nat -> Expr
3  | Var : Nat -> Expr
4  | Add : Expr -> Expr -> Expr
5
6  lookupVar : (x : Nat) -> (env : List (Nat, Nat)) -> Maybe Nat
7  lookupVar x [] = Nothing
8  lookupVar x ((y, val)::ys) = if x == y then Just val
9                                else lookupVar x ys
10
11 eval : (env : List (Nat, Nat)) -> Expr -> Maybe Nat
12 eval env (Num n) = Just n
13 eval env (Var x) = lookupVar x env
14 eval env (Add e1 e2) =
15   case eval env e1 of
16     Just e1' => case eval env e2 of
17       Just e2' => Just (plus e1' e2')
18       Nothing => Nothing
19     Nothing => Nothing

```

Fig. 14: The basic language and evaluator.

to produce a stronger version of `eval`: presently, if a variable is not found in the environment, `eval` returns `Nothing`. This means that if an empty environment or even the wrong environment is passed to `eval`, the evaluator will not be able to produce a result. A much stronger version of the function would be to remove the `Maybe` in `eval`'s return type, meaning that `eval` will always return a result if the correct environment is given as a parameter. It is removing this `Maybe` that we set as our goal for this use case, employing the refactorings that were outlined in Section 3 under user-guidance to achieve it.

4.1 Step 1: Introduce an index to `Expr`

With the basic language and evaluator set up, the first step is to start refining the type `Expr` so that it is indexed over a list representing variables that are defined (or in-scope). The idea here would be to add a further refinement to the type later on so that a proof that a variable is indeed defined in its scope can be passed to the `Var` constructor as a parameter.

In order to represent the variables that are in scope, we choose the *Add an index to a data type* refactoring from Section 3.1. The user gives `List Nat` as the type of the index to be added. The result of the refactoring is illustrated in Figure 15, where `Expr` has been refactored to take a `List Nat` type as an index. All the constructors to `Expr` have also been refactored to pass a variable of type `List Nat` as an index. In the `Add` constructor, the refactoring introduces three variables for the three different usages of the index variable. These will be unified together in a further refactoring step. The `eval` function has also

```

1 data Expr : (vars : List Nat) -> Type where
2   Num : (vars : List Nat) -> Nat -> Expr vars
3 | Var : (vars : List Nat) -> Nat -> Expr vars
4 | Add : (vars : List Nat) -> (vars1 : List Nat) -> (vars2 : List Nat)
5         -> (Expr vars1) -> (Expr vars2) -> Expr vars
6 ...
7 eval : (vars : List Nat) -> (env : List (Nat, Nat)) -> (Expr vars) ->
8       Maybe Nat
9 eval vars env (Num vars n) = Just n
10 eval vars env (Var vars x) = lookupVar x env
11 eval vars env (Add vars v1 v2 e1 e2) =
12   case eval v1 env e1 of
13     Just e1' => case eval v2 env e2 of
14       Just e2' => Just (plus e1' e2')
15       Nothing => Nothing
16   Nothing => Nothing

```

Fig. 15: Step 1: Introduce a `List Nat` index to `Expr`.

been transformed by the refactoring, as its type is modified to take a `List Nat` parameters (called `vars`) and update the type `Expr` to take `vars` as an index. The clauses of `eval` are transformed to introduce pattern variables for the new parameter added to `eval` and also in the pattern matching on the constructors to `Expr`. The recursive calls have also been updated, this is done by the refactoring understanding that the index variable for the `e1` case must be `v1` (and similarly for the `e2` case).

4.2 Step 2: Specialise and unify index variables in `Expr`

The next step requires us to start specialising and refining the type `Expr` to better reflect representing well-formed expressions. The result of this step is illustrated in Figure 16. Firstly, the index in the `Num` constructor is *specialised* (via the *Specialising an index in a data type* refactoring from Section 3.2). Secondly, the three parameters to `Add` representing the three instances of the index are unified into a single parameter. This represents the fact that in addition, the two operands must both be well formed over the same environment of identifiers (the identifiers in scope are the same for both `e1` and `e2`). The function `eval` is transformed automatically to remove the pattern variables for the old `vars1` and `vars2`. The recursive call is also automatically updated.

4.3 Step 3: Relate the parameters of `lookupVar`

As the goal is to transform the return type of `eval` into `Nat` instead of `Maybe Nat`, we must also do the same to `lookupVar`. To this end, we add a proof parameter to `lookupVar` that requires that the variable is indeed defined in its

```

1 data Expr : (vars : List Nat) -> Type where
2   Num : (vars : List Nat) -> Nat -> Expr []
3   | Var : (vars : List Nat) -> Nat -> Expr vars
4   | Add : (vars : List Nat) -> (Expr vars) -> (Expr vars) -> Expr vars
5   ...
6 eval : (vars : List Nat) -> (env : List (Nat, Nat)) -> (Expr vars) ->
7       Maybe Nat
8 eval [] env (Num vars n) = Just n
9 eval vars env (Var vars x) = lookupVar x env
10 eval vars env (Add vars e1 e2) =
11   case eval vars env e1 of
12     Just e1' => case eval vars env e2 of
13       Just e2' => Just (plus e1' e2')
14       Nothing => Nothing
15     Nothing => Nothing

```

Fig. 16: Step 2: Specialise `Num` and unify index variables in `Add`.

environment. In order to do this, we introduce a proof via the `Elem` type that `x` is defined in its environment. As `env` is defined a list of tuples, we cannot use it directly in a proof of `Elem`, therefore we will introduce a new parameter representing the list of variables in scope (similarly to Figure 16). Figure 17 illustrates the result of this refactoring. Here, two new parameters have been added to `lookupVar`: `vars` representing the list of variables in scope, and `Elem x vars` (via the *Relating function parameters* refactoring from Section 3.5), representing a proof that `x` is defined in `vars`. The function is transformed to introduce the new parameters. Secondly, we *expand* the pattern variables of type `Elem x vars` via the *Expanding a pattern variable* from Section 3.6.

```

1 lookupVar : (vars : List Nat) -> (x : Nat) -> (env : List (Nat, Nat))
2           -> (Elem x vars) -> Maybe Nat
3 lookupVar (x::vars) x [] Here = Nothing
4 lookupVar (v::vars) x [] (There p) = Nothing
5 lookupVar (x::vars) x ((y,val)::ys) Here =
6   if x == y then Just val
7   else lookupVar (x::vars) x ys Here
8 lookupVar (v::vars) x ((y,val)::ys) (There p) =
9   if x == y then Just val
10  else lookupVar (v::vars) x ys (There p)

```

Fig. 17: Step 3: Introduce parameter `(vars : List Nat)` and relate function parameters using `(Elem x vars)` in `lookupVar`; expand pattern variables of type `(Elem x vars)`.

4.4 Step 4: Relate the parameters of the `Var` constructor

```

1  -- assume we have
2  data Unzip : List (Nat, Nat) -> List Nat -> List Nat -> Type where
3      NilUZ : Unzip [] [] []
4      ConsUZ : Unzip xs vs ws -> Unzip ((x,y)::xs) (x :: vs) (y :: ws)
5  ...
6  Var : (vars : List Nat) -> (n : Nat) -> Elem n vars -> Expr vars
7  ...
8  eval : (ws : List Nat) -> (vars : List Nat)
9         -> (env : List (Nat, Nat)) -> (Expr vars)
10        -> (Unzip env vars ws) -> Maybe Nat
11 eval [] [] [] (Num n) NilUZ = Just n
12 eval [] [] [] (Var [] x Here) NilUZ impossible
13 eval (y::ws) (x::vars) ((x,y)::env) (Var (x::vars) x Here) (ConsUZ u)
14     = lookupVar (x::vars) x ((x,y)::env) Here
15 eval [] [] [] (Var (x'::[]) x (There p)) NilUZ impossible
16 eval (y::ws) (x'::vars) ((x',y)::env) (Var (x'::vars) x (There p))
17     (ConsUZ u) = lookupVar (x'::vars) x ((x',y)::env) (There p)
18 eval ws vars env (Add vars e1 e2) u =
19   case eval ws vars env e1 u of
20     Just e1' => case eval ws vars env e2 u of
21       Just e2' => Just (plus e1' e2')
22       Nothing => Nothing
23   Nothing => Nothing

```

Fig. 18: Step 4: Relate the parameters of the `Var` constructor with `Elem n vars`, introduce a parameter `(ws : List Nat)` and relation `Unzip env vars ws` to the function `eval`; expand pattern variables.

In this step we first introduce a relation in the constructor `Var` that requires a proof that `n` is defined in `vars`. We then turn our attention to the `eval` function where we need to relate `env` and `vars` (the index to `Expr`). We do this in a similar way to Step 3, where we introduce an `Unzip` predicate to `eval` over an introduced parameter `ws`. The `Unzip` predicate is assumed to exist as either part of the program, or of a standard library; we assume access to both its type definition and covering function. The final step is to expand the pattern matching over the `Unzip env vars ws` type causing variables to unify and impossible cases to be introduced by the refactoring tool. The resulting code is given in Figure 18.

4.5 Step 5 Introduce predicate and expand pattern variables

In this step, we perform a similar refactoring sequence to Step 4, turning our attention instead to `lookupVar`. Here we refactor `lookupVar` to introduce a predicate `Unzip env vars ws`, again relating `env` and `vars`. The pattern matches

```

1 lookupVar : (vars : List Nat) -> (x : Nat) -> (env : List (Nat, Nat))
2           -> (Elem x vars) -> (Unzip env vars ws) -> Maybe Nat
3 lookupVar [] x [] Here NilUZ impossible
4 lookupVar [] x [] (There p) NilUZ impossible
5 lookupVar (y::vars) y ((y,val)::ys) Here (ConsUZ u) =
6   if y == y then Just val
7   else lookupVar (y::vars) y ys ?proof1 ?proof2
8 lookupVar (v::vars) x ((v,val)::ys) (There p) (ConsUZ u) =
9   if x == v then Just val
10  else lookupVar (vars) x ys p u

```

Fig. 19: Step 5: Introduce predicate `Unzip env vars ws`; expand pattern variables.

for `lookupVar` are also expanded, introducing impossible cases. The result is illustrated in Figure 19. Note that the expanding pattern matching has introduced unification on the variables, introducing a tautology in the `if` expression on Line 6. We can eliminate this in the next step and remove the `Maybe` in the return type.

4.6 Step 6 Remove tautology and eliminate `Maybe` from `lookupVar`

```

1 lookupVar : (vars : List Nat) -> (x : Nat) -> (env : List (Nat, Nat))
2           -> (Elem x vars) -> Unzip env vars ws -> Nat
3 lookupVar [] x [] Here NilUZ impossible
4 lookupVar [] x [] (There p) NilUZ impossible
5 lookupVar (y::vars) y ((y,val)::ys) Here (ConsUZ u) = val
6 lookupVar (v::vars) x ((v,val)::ys) (There p) (ConsUZ u) =
7   if x == v then val
8   else lookupVar (vars) x ys p u

```

Fig. 20: Step 6: Removing tautology `y == y`; eliminate `Maybe`.

We can now remove the `Maybe` from the return type of `lookupVar`. First we remove the tautology from the `if` statement using the *Eliminating tautologies* refactoring from Section 3.8. Secondly, we eliminate the `Maybe` type, as there are no longer any cases that return a `Nothing` value (these were transformed into impossible cases in the previous step). This uses the *Eliminating Maybe* refactoring from Section 3.9. The result is shown in Figure 20.

4.7 Step 7 Eliminate `Maybe` from `eval`

```

1  eval : (ws : List Nat) -> (vars : List Nat) ->
2      (env : List (Nat, Nat)) -> (Expr vars) ->
3      (Unzip env vars ws) -> Nat
4  eval [] [] [] (Num n) NilUZ = n
5  eval [] [] [] (Var [] x Here) NilUZ impossible
6  eval (y::ws) (x::vars) ((x,y)::env) (Var (x::vars) x Here) (ConsUZ u)
    = lookupVar (x::vars) x ((x,y)::env) Here (ConsUZ u)
7  eval [] [] [] (Var (x'::[]) x (There p)) NilUZ impossible
8  eval (y::ws) (x'::vars) ((x',y)::env) (Var (x'::vars) x (There p)) (
    ConsUZ u) = lookupVar (x'::vars) x ((x',y)::env) (There p) (
    ConsUZ u)
9  eval ws vars env (Add vars e1 e2) u =
10     case eval ws vars env e1 u of
11     e1' => case eval ws vars env e2 u of
12     e2' => plus e1' e2'

```

Fig. 21: Step 7: Eliminate *Maybe*.

Finally we can eliminate the *Maybe* from the return type of `eval` using the *Eliminating Maybe* refactoring. As `lookupVar` no longer returns a *Maybe* type and `eval` does not return *Nothing*, the *Maybe* can be safely removed. The final result is shown in Figure 21.

5 Related work

Refactorings The Haskell Refactorer, HaRe [Li06,Bro08,LTR05,BLT11], is a refactoring tool for Haskell 98 that is itself also implemented in Haskell. The tool supports a wide number of refactorings, including *renaming*, *generalisation*, *lifting*, *folding*, and *clone detection* [BT10]. The refactorings are described in terms of pre-conditions, with a set of unit tests given as part of the implementation. More recently, Williams [Wil20] presents a refactoring for ML based on ornaments [McB10] that transforms functions between similar simply-typed structures. Following this, Robert [Rob18] explores *function-repair* (a precursor to proof-repair) after a change in their underlying data types. Both Robert's work and this paper deal with propagating changes in data types across functions. However, the main difference is that Robert derives the difference in data types after amendments whereas we aim to *guide* the programmer in making these amendments. Wibergh presents a preliminary catalogue of refactorings for Agda in their MSc thesis [Wib19], but to the best of our knowledge, lacks an implementation. Although the catalogue mostly comprises standard structural refactorings applied to Agda, it includes two that are closely related to our *Adding an index to a data type* refactoring in Section 3.1. Presently, the principle difference between Wibergh's and our refactoring lies in their respective target languages.

Proof-repair systems are concerned with the refactoring of proofs in the context of theorem provers [RYLG19, Ada15, Whi13]. While related, such systems differ from our approach in that they are primarily focused on tactics-based theorem provers, and rely on ornaments to guide their transformations between equivalent representations of enriched programs. Our approach differs in two main ways: 1. our approach transforms *programs*, rather than proof scripts (i.e. sequences of tactics); and 2. our approach is *exploratory*, i.e. transformations to facilitate enrichment of initially simply-typed programs.

Transformations over similar data types. There is much work on transformations across equivalent or isomorphic data types [BP01, RPY⁺21, ZH15], and across different representations of the same abstract data type [CDM13, DMS12, Lam13]. Ornaments [McB10] represent a means to relate data types that are structurally similar but not necessarily equivalent. Notably, ornaments have been used in transforming theorems by both Ringer [RYLG19] and Williams [WDR14], as discussed above. Our work goes beyond this use of ornaments, such that our approach enables the introduction and manipulation of arbitrary proof terms to functions and data types independently. Work on propagating changes precipitated by transformations to data types is studied by Robert [Rob18], where the repair functions are derived from the computed differences using elimination motives [McB00]. Meanwhile, Boite focuses on the specific case of propagating changes after adding constructors to data types [Boi04]. Johnsen accommodates proof reuse by generalising theorems by abstracting their proofs [JL04].

6 Conclusions & Future work

In this paper, we proposed a number of new refactorings for dependently-typed programs, defined for the *Fluid* programming language, a small dependently-typed functional language with inductive types. We presented our refactorings in the form of a small catalogue of transformations, with examples and descriptions of their general conditions. We also demonstrated our refactorings on a use case comprising a small expression language, where, through a series of applying our refactoring steps, we enriched the program to make use of dependent types via the introduction of predicates and proof terms. Using our refactorings, we were able to transform an evaluator that returned a *Maybe Nat* type to one that simply returned a *Nat*, and therefore demonstrated transformations that enabled a fallible function to be transformed into an infallible function: in our expression language, we transformed our evaluator from a function that could fail (i.e. return a *Nothing*) into one that will always give an evaluated expression. This example demonstrated that using our refactoring approach, developers are able to explore the use of dependent-types to develop stronger and safer programs.

Our approach is still preliminary, and the catalogue of refactorings that we propose in this paper is still under development. For the final paper, we promise to present a full implementation of each of the refactorings presented here, together with a number of examples demonstrating the refactorings refine and

make use of dependent-types to build strong, safe, programs. We will also explore the use of transformation rules, to provide more precise descriptions of the refactorings. In the future, we intend to explore the idea of correctness and how it relates to refactorings that transform the domains and codomains of functions. We will also explore the use of ornaments in refactoring, characterise our existing refactorings in terms of ornaments, and provide refactorings that allow the transformation between different types, particularly simple types and their dependent versions.

Acknowledgements

This work was generously supported by UK EPSRC *Energise*, grant number EP/V006290/1.

References

- [Ada15] Mark Adams. Refactoring Proofs with Tactician. In *Software Engineering and Formal Methods: SEFM 2015 Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY* SCART, York, UK, September 7-8, 2015. Revised Selected Papers*, pages 53–67. Springer, 2015.
- [BLT11] Christopher Brown, Huiqing Li, and Simon Thompson. An Expression Processor: A Case Study in Refactoring Haskell Programs. In Rex Page, Zoltán Horváth, and Viktória Zsók, editors, *Trends in Functional Programming*, pages 31–49, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [Boi04] Olivier Boite. Proof Reuse with Extended Inductive Types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65. Springer, 2004.
- [BP01] Gilles Barthe and Olivier Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In *International Conference on Foundations of Software Science and Computation Structures*, pages 57–71. Springer, 2001.
- [Bra21] Edwin C. Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPICs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [Bro08] Christopher Brown. *Tool Support for Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, September 2008.
- [BT10] Christopher Brown and Simon Thompson. Clone Detection and Elimination for Haskell. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '10*, page 111–120, New York, NY, USA, 2010. Association for Computing Machinery.
- [CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.
- [DMS12] Maxime Dénès, Anders Mörtberg, and Vincent Siles. A Refinement-Based Approach to Computational Algebra in Coq. In *International Conference on Interactive Theorem Proving*, pages 83–98. Springer, 2012.

- [Fou] Eclipse Foundation. Eclipse. <http://www.eclipse.org/>. Accessed: 2023-12-7.
- [Fow99] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999.
- [Hur95] Antonius J C Hurkens. A Simplification of Girard’s Paradox. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications*, TLCA ’95, pages 266–278, Berlin, Heidelberg, April 1995. Springer-Verlag.
- [JL04] Einar Broch Johnsen and Christoph Lüth. Theorem Reuse by Proof Term Transformation. In *International Conference on Theorem Proving in Higher Order Logics*, pages 152–167. Springer, 2004.
- [Lam13] Peter Lammich. Automatic Data Refinement. In *International Conference on Interactive Theorem Proving*, pages 84–99. Springer, 2013.
- [lan] Official Page for Language Server Protocol. <https://microsoft.github.io/language-server-protocol/>. Accessed: 2023-12-6.
- [Li06] Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, University of Kent, Canterbury, Kent, UK, September 2006.
- [LT12] Huiqing Li and Simon J. Thompson. A Domain-Specific Language for Scripting Refactorings in Erlang. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7212 of *Lecture Notes in Computer Science*, pages 501–515. Springer, 2012.
- [LTR05] Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer: HaRe, and its API. In John Boyland and Görel Hedin, editors, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, pages 182–196, April 2005. Published as Volume 141, Number 4 of Electronic Notes in Theoretical Computer Science, <http://www.sciencedirect.com/science/journal/15710661>.
- [McB00] Conor McBride. Elimination with a Motive. In *International Workshop on Types for Proofs and Programs*, pages 197–216. Springer, 2000.
- [McB10] Conor McBride. Ornamental Algebras, Algebraic Ornaments. *Journal of functional programming*, 47, 2010.
- [Met] Meta. Retire: Haskell Refactoring Made Easy. <https://engineering.fb.com/2020/07/06/open-source/retrie/>. Accessed: 2023-12-7.
- [Nor08] Ulf Norell. Dependently Typed Programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.
- [Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois Urbana-Champaign, USA, 1992.
- [PM15] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. *All about Proofs, Proofs for All*, 55, January 2015.
- [Rob18] Valentin Robert. *Front-end Tooling for Building and Maintaining Dependently-Typed Functional Programs*. University of California, San Diego, 2018.

- [RPY⁺21] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. Proof Repair Across Type Equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 112–127, 2021.
- [RYLG19] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for Proof Reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- [WDR14] Thomas Williams, Pierre-Évariste Dagand, and Didier Rémy. Ornaments in Practice. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 15–24, 2014.
- [Whi13] Iain Johnston Whiteside. *Refactoring Proofs*. PhD thesis, The University of Edinburgh, 2013.
- [Wib19] Karin Wibergh. Automatic Refactoring for Agda, 2019. MSc thesis, University of Gothenburg.
- [Wil20] Ambre Williams. *Refactoring Functional Programs with Ornaments*. PhD thesis, Université de Paris, 2020.
- [ZH15] Théo Zimmermann and Hugo Herbelin. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. In *Conference on Intelligent Computer Mathematics*, 2015.