# Structural Refactorings for Exploring Dependently Typed Programming

Adam D. Barwell[1][0000−0003−1236−7160], Christopher Brown[1][0000−0001−6030−2885], Mun See Chang[1][0000−0003−2428−6130], Constantine Theocharis[1][0009−0001−0198−2750], and Simon Thompson[2][0000−0002−2350−301X]

[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.** Dependent types provide users with the tools to embody specifications in types, with implementations carrying proofs that the specifications are met. One approach to developing programs in a dependently typed language develops such programs by enriching simply-typed programs through a process of refactoring.

**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], Agda [Nor08], Coq [Coq] and Lean [dMKA+15], provide strong typing mechanisms that permit logical properties to be expressed as types, and 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 properties of the code as types (or predicates) and *enriching* a program to demonstrate conformance with these requirements. An exploratory approach naturally presents itself, in which 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 its (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 functional languages [Bro08,Met,LT12] among others, and have been integrated into IDEs [Fou] and editors, initially in an *ad hoc* way and more recently by means of 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 for which it would be impractical to perform the refactoring by hand, and errors are avoided by the consistent and correct application of those transformations, which can themselves be demonstrated to be correct [ST08,TH23].

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 [Rin21,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* [McB11]. Although these techniques represent a key component of type enrichment refactoring, they do not encompass its entirety.

In this paper, we introduce a complementary approach to refactoring dependently typed programs, building a set of refactorings that generalise 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 the application of another refactoring, and we will see examples of that here. In this paper, we make the following contributions:

1. We introduce a small dependently typed functional language, named Fluid, similar to a subset of Idris and based on a dependently typed lambda calculus with inductive types.
2. We introduce a number of refactorings over Fluid for dependently typed programming.
3. We implement these refactorings in a prototype refactoring tool.
4. We demonstrate these refactorings in a number of examples, and show improved safety properties over the original programs.

The implementation of Fluid and the prototype refactoring tool can be found at `github.com/kontheocharis/fluid`.

## 2 Fluid

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 Fig. 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, from top to bottom. Clauses that match on a sequence of incompatible patterns can be marked as `impossible`.

Program $P$ ::= $(D \mid T)*$

Name $v$

Declaration $D$ ::=
- $v$ `:` $t$
- $C[v]*$

Clause $C[v]$ ::=
- $\mid v\ t_1 \ldots t_n$ `=` $t$
- $\mid v\ t_1 \ldots t_n$ `impossible`

Type definition $T$ ::=
- `data` $v$ `:` $t$ `where`
- $\mid v_1$ `:` $t_1 \ldots \mid v_n$ `:` $t_n$

Term $t$ ::=

| | |
|---|---|
| `(`$v$ `:` $t_1$`)` `->` $t_2$ | (function type) |
| `(`$v$ `:` $t_1$`)` `**` $t_2$ | (pair type) |
| `Type` | (universe type) |
| $v$ | (name) |
| `\`$v$ `=>` $t$ | (function) |
| `(`$t_1$`,` $t_2$`)` | (pair) |
| $t_1\ t_2$ | (application) |
| `case` $t$ `of` $\mid t_1$ `=>` $t'_1 \ \ldots\ \mid t_n$ `=>` $t'_n$ | (case expression) |
| `_` | (wildcard pattern/infer term) |
| `?`$v$ | (named hole) |

Fig. 1: The syntax of Fluid.

Type definitions in Fluid support inductive families [Dyb94], similar to those in Agda or Idris, but without implicit parameters. 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 discuss in this paper.

Fluid 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 symbols `_::_` and `[]` are overloaded and used for both lists and vectors, disambiguated by context, where their type arguments are inferred from usage. For brevity, we omit the typing rules and semantics of Fluid; they closely mirror those of CIC, but with some small limitations, serving as a prototype language to demonstrate the principle of our refactoring approach. Fluid does not consider universe levels or a distinction

between Type and Prop[3]; Fluid supports multiple declarations and arbitrary recursion, which CIC does not; and, finally, the implementation of Fluid does not force the exhaustiveness of patterns, but does ensure that `impossible` branches are correctly marked so.

## 3 Refactorings for Dependently Typed Programs

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 § 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 can often arise as the result of earlier refactoring steps, and we envisage that the refactorings will typically be used as part of a sequence of transformation steps, rather than individually. The refactorings presented here do not constitute an exhaustive list, but rather a sample that is useful from a potentially wide and common class of transformations.

We illustrate each refactoring by giving examples representing code before and after their application, with high-level descriptions of the pre-conditions needed. In many cases the inverse transformations are possible with stronger pre-conditions, and are currently the target of future work. In the figures presented, we use yellow where a change has been made by the refactoring tool, red where code has been removed, green where code has been added, and blue where a hole has been introduced.

### 3.1 Adding an Index to a Data Type

*Adding an Index to a Data Type* adds an index to a specified data type. An index is a way of grouping the constructors of a data type into distinct sets—for example, grouping lists by their length yields the type of vectors, indexed by `Nat`. For a given data type `D : T1 -> ... -> Tn -> Type` and a requested index type `I` and position $i$, the refactoring refactors the definition of `D` to

```
D : T1 -> ... -> Ti -> I -> ... -> Tn -> Type.
```

Additional parameters of type `I` are added to each constructor of `D` and used for the indices of any inductive steps and the index of the constructor itself. All references to the type are updated for these newly added parameters.

*Example* Fig. 2 shows an example of an index being added to a data type, of type `Nat`. For each constructor, an index is added for each inductive step, as well as for the overall index of the constructor. For example, a new parameter, `n2` of type `Nat` has been added to constructor `With` to capture the index in the type overall, while an additional parameter `n1` of the same type has also been added to capture the index in the inductive parameter. All new parameters are fresh.

[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] in the logical interpretation of the language, but that should not affect the approach that we present here.

All other parts of the program which reference `Bits` must be updated to include the new index that was added. Fig. 3 gives an example of such a function that is updated. Two new parameters are introduced to capture the index argument for the data type. All clauses are updated to match on the new parameters. Any references to the constructors of `Bits`, as well as recursive calls, are updated to pass in the new index parameters. Finally, all the call sites of `flip` are updated to provide proofs for the added parameters.

When there exists a unique solution for the required arguments that can be found through pattern unification, it is inserted automatically. When no such solution is found, holes are added and it is the user's responsibility to fill them. This refactoring can be viewed as producing the "freely indexed" version of the original data type. As such, it is semantically equivalent to the original and the transformation process should not lead to any type errors or difference in program behaviour (modulo holes).

### 3.2 Specialising an Index of a Constructor

*Specialising an Index of a Constructor* sets the index of a specified constructor to a custom value. For a constructor `C` with return type `D t1 ... ti ... tn`, specialising the $i$th index to value `v` results in the return type of `C` being refactored to `D t1 ... v ... tn`. The type of `v` must match the type of `ti`. This refactoring does not update any usage sites, as this is currently intractable automatically: the user must ensure that the newly specialised index is compatible with the rest of the program.

*Example* Fig. 4 specialises two constructors by setting their indices to `Z` and `(S n2)` respectively. The parameter for the unused index is retained as there is a separate refactoring to remove it (§ 3.4).

### 3.3 Relating Constructor Parameters

*Relating Constructor Parameters* allows a new parameter to be inserted in a constructor `C` to relate some parameters via a dependent type. The input consists of the positions of the constructor parameters `t1 ... tn` to relate, and the predicate term `P` that should be used to relate them. The constructor definition

```
1 data Bits : Type where
2 | Empty : Bits
3 | With : Bool
4     -> Bits
5     -> Bits
```

(a) Before

```
1 data Bits : Nat -> Type where
2 | Empty : (n : Nat) -> Bits n
3 | With : Bool
4     -> (n1 : Nat) -> Bits n1
5     -> (n2 : Nat) -> Bits n2
```

(b) After

Fig. 2: Adding an index in a data type. An index of type `Nat` is added to the data type `Bits` at position 0.

```
flip : Bits
    -> Bits
flip Empty = Empty
flip (With x xs) =
    With (not x)
         (flip xs)
```

(a) Before

```
flip : (n1 : Nat) -> Bits n1
    -> (n2 : Nat) -> Bits n2
flip _ ( Empty _) n2 = Empty n2
flip _ (With x n1' xs _) n2 =
    With (not x) ?n1
         (flip n1' xs ?n1) n2
```

(b) After

Fig. 3: Updating a usage site of the data type after adding an index.

```
data Bits : Nat -> Type where
| Empty : (n : Nat) -> Bits n
| With : Bool
    -> (n1 : Nat) -> Bits n1
    -> (n2 : Nat) -> Bits n2
```

(a) Before

```
data Bits : Nat -> Type where
| Empty : (n : Nat) -> Bits Z
| With : Bool
    -> (n1 : Nat) -> Bits n1
    -> (n2 : Nat) -> Bits (S n2)
```

(b) After

Fig. 4: Specialising the constructors `Empty` and `With` to `Z` and `(S n2)`.

is refactored to include an additional parameter `P t1 ... tn`, inserted in the final parameter position. Usage sites of `C` are updated with a hole at the position of the newly added parameter.

*Example* In Fig. 5, introducing a relation between the two parameters of constructor `Sat` is done by inserting a new parameter of type `LTE found searched`. All pattern matches over the constructor `Sat` are modified to introduce a variable in the position of the added relation. Any applications of `Sat` are also transformed by introducing a hole at the relation position, allowing the programmer to complete it later.

```
data Solution : Type where
| Sat : (searched : Nat)
    -> (found : Nat)

    -> Solution
| Unsat : (searched : Nat) ->
    Solution
```

(a) Before

```
data Solution : Type where
| Sat : (searched : Nat)
    -> (found : Nat)
    -> LTE found searched
    -> Solution
| Unsat : (searched : Nat) ->
    Solution
```

(b) After

Fig. 5: Relating the parameters `searched` and `found` in the `Sat` constructor of the `Solution` data type.

```
data Bits : Nat -> Type where
| Empty : (n : Nat) ->  Bits Z
| With : Bool
   -> (n1 : Nat) -> Bits n1
   -> (n2 : Nat) ->  Bits (S n2)
```

(a) Before

```
data Bits : Nat -> Type where
| Empty : Bits Z
| With : Bool
    -> (n1 : Nat) -> Bits n1
    -> Bits (S n1)
```

(b) After

Fig. 6: Collapsing the `Nat` indices of the data type `Bits` as defined previously.

### 3.4 Collapsing Constructor Parameters

*Collapsing Constructor Parameters* collapses one or more of the indices in a constructor. Alongside *Specialising an Index of a Constructor*, it can be used to refine the free indexing structure produced by *Adding an Index to a Data Type* and is a more general form of removing redundant parameters from constructors. The input is a target constructor `C`, a list of positions for a subset of its parameters `t1 ... tn`, and an optional distinguished parameter `ti` among those. The refactoring removes each parameter `t1 ... tn` from the constructor, other than `ti`. Occurrences of `t1 ... tn` in the rest of the constructor are replaced with `ti`. If no `ti` is provided, the parameters `t1 ... tn` are simply removed, in which case they must not be referenced elsewhere in the constructor. All usages of `C` are updated to remove the appropriate parameters, and all usages of the removed parameters are replaced with the retained parameter (if present). If the correct argument to the retained parameter cannot be solved through pattern unification, it is filled with a hole.

*Example* Fig. 6 demonstrates an example of collapsing constructor parameters in a data type. In the case of `Empty`, there is no target index, so the index `n` is removed. In the case of `With`, the indices `n1` and `n2` are collapsed to `n1`, with `(S n2)` being replaced with `(S n1)`. In Fig. 7, a usage site of `Bits` is updated to remove the collapsed parameters. In the `With` application, the argument `n1` in the output is retained only because pattern unification dictates so; in general it would be replaced with a fresh hole to ensure that the program still typechecks.

```
flip : (n : Nat) -> Bits n
    -> Bits n
flip Z ( Empty n)  = Empty n
flip (S n) (With x n1 xs n2 ) =
    With (not x) n1
        (flip n1 xs) n2
```

(a) Before

```
flip : (n : Nat) -> Bits n
    -> Bits n
flip Z Empty = Empty
flip (S n) (With x n1 xs) =
    With (not x) n1
        (flip n1 xs)
```

(b) After

Fig. 7: Updating a usage site of `Bits` after collapsing parameters. It is assumed that some further refactoring has occurred since Fig. 3.

```
idx : (x : Nat)
    -> (ys : List Nat)

    -> Maybe Nat
idx x [] = Nothing
idx x (y::ys) =
    case y == x of
    | True => Just x
    | False => idx x ys
```

(a) Before

```
idx : (x : Nat)
    -> (ys : List Nat)
    -> Elem x ys
    -> Maybe Nat
idx x [] p = Nothing
idx x (y::ys) p =
    case y == x of
    | True => Just x
    | False => idx x ys ?p
```

(b) After

Fig. 8: Relating the parameters `x` and `ys` of the indexing function `idx` with `Elem`.

### 3.5 Relating Function Parameters

Similarly to *Relating Constructor Parameters*, *Relating Function Parameters* inserts a dependent type as a new parameter to a function `f` to relate 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. The function `f` is updated to introduce pattern matching variables for the new parameters. Usage sites of `f`, including recursive calls, are augmented with holes for the new parameters (which is often solved by pattern unification).

*Example* In Fig. 8, we introduce a new parameter of type `Elem x ys` to the function `idx`. All equations over `f` are updated to include a pattern variable representing the inserted parameter. The recursive call to `idx` in the second clause is augmented with a fresh hole that must be provided by the user—pattern unification does not help here.

### 3.6 Expanding a Pattern Variable

*Expanding a Pattern Variable* replaces equations containing a specified pattern variable in a function with new equations that enumerate the possible patterns for that variable, to one level of depth. This refactoring follows similar, existing, approaches, such as the Interactive Editing mode for Idris (`https://idris2.readthedocs.io/en/latest/tutorial/interactive.html`). The expanded equations that become *impossible* as a result of the refactoring are removed. The refactoring has two variants: *i)* expand patterns for a single equation; and, *ii)* expand patterns for all equations.

```
count : (x : Nat)
    -> (ys : List Nat)
    -> Elem x ys -> Nat

count x ys p = ?result
```

(a) Before

```
count : (x : Nat)
    -> (ys : List Nat)
    -> Elem x ys -> Nat
count x ys Here = ?result
count x ys (There p) = ?result
```

(b) After

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

```
1 count : (x : Nat)
2     -> (ys : List Nat)
3     -> Elem x ys -> Nat
4 count x ys Here = ?result
5 count x ys (There p) = ?result
```

(a) Before

```
1 count : (x : Nat)
2     -> (ys : List Nat)
3     -> Elem x ys -> Nat
4 count x (y::ys') Here = ?result
5 count x ys (There p) = ?result
```

(b) After

Fig. 10: Expanding pattern matching for a variable over a single equation.

```
1 count : (x : Nat)
2     -> (ys : List Nat)
3     -> Elem x ys -> Nat
4 count x ys Here = ?result
5 count x ys (There p) =
6     ?result
```

(a) Before

```
1 count : (x : Nat)
2     -> (ys : List Nat)
3     -> Elem x ys -> Nat
4 count x (y::ys') Here = ?result
5 count x (y::ys') (There p) =
6     ?result
```

(b) After

Fig. 11: Expanding pattern matching for a variable over all equations.

*Example* Fig. 9 expands the parameter argument `p` to pattern match over constructors of the `Elem` type. This results in two equations for both the `Here` and `There` case. A further application of this refactoring on the same function is illustrated in Fig. 10 where the parameter `ys` is then expanded for the equation on Line 3. All cases matching `[]` are impossible due to `Elem` relating both `ys` and `x`, and are removed by the expansion process. Finally, Fig. 11 demonstrates the second variant of the refactoring which expands *all* equations for the pattern variable selected. This results in `ys` being expanded to `(y::ys')` in the equation on Line 4. The impossible case matching `[]` is removed in the figure for brevity.

### 3.7 Eliminating Tautologous Cases

Expanding pattern variables can result in redundant `case` expressions, which can be eliminated. Potentially arising from the unification of pattern variables, a `case` expression that can match only a single clause can be safely replaced with the RHS of that clause.

A common example can be found in `case` expressions equivalent to if statements (Fig. 12), where arguments passed to an equality test are the same. In such cases, since the matching pattern `True` declares no arguments, the corresponding RHS can replace the `case` expression verbatim.

The *Eliminating Tautologous Cases* refactoring takes a location within a program and the name of a binary operator, say (==), representing an equality test. The location must be contained within a `case` expression such that its subject is of the form $x == x$, where $x$ is a variable. The `case` expression must

```
idx : (x : Nat)
    -> (xs : List Nat)
    -> Elem x xs -> Maybe Nat
idx x (x :: ys) Here =
    case x == x of
    | True =>  Just x
    | False => idx x ys ?h1
idx x (y :: ys) (There p) =
    case y == x of
    | True => Just x
    | False => idx x ys ?h2
```

(a) Before

```
idx : (x : Nat)
    -> (xs : List Nat)
    -> Elem x xs -> Maybe Nat
idx x (x :: ys) Here =
    Just x


idx x (y :: ys) (There p) =
    case y == x of
    | True => Just x
    | False => idx x ys ?h2
```

(b) After

Fig. 12: Eliminating the tautology `x == x` in a `case` expression.

have at least one clause, with pattern `True`. The result, illustrated in Fig. 12, is the replacement of the `case` expression with the RHS of the `True` clause.

In principle, *Eliminating Tautologous Cases* can be generalised beyond this specific form of `case` expression. In particular, where prior refactoring stages produce a subject of the form `C e1 ... en`, where `C` is a constructor. Here, the RHS of the matching clause replaces the `case` expression, substituting the pattern variables for `e1 ... en`. Further generalisation, e.g. where the subject is a function call that returns a given constructor for all inputs, would require additional inspection of the called function or use of *Unfolding* [BD77]. Examples may arise as a precursor to *Eliminating* `Maybe` (§ 3.8).

### 3.8 Eliminating `Maybe`

*Eliminating* `Maybe` in the return type of a function `f` is a refactoring which can be applied if `f` never returns `Nothing`. Eliminating the `Nothing` cases is possible by first transforming these cases in a way that makes them `impossible` (for example through *Expanding a Pattern Variable*). The `Maybe` can then be eliminated by removing the `Just` constructs from each clause. If a clause is of the form `Just x`, it is transformed to `x`. If it is of the form `f x1 ... xn`, it is already valid since it is a recursive call. If a recursive call is being matched with a `case` expression, only the `Just` branch is retained and the `Nothing` branch is removed. In all other cases, `Maybe` cannot be eliminated. In terms of usage sites, all (non-recursive) occurrences of `f x1 ... xn` are replaced with `Just (f x1 ... xn)` since the usage sites still expect `f` to produce a `Maybe`.

*Example* In Fig. 13, the return type of the function (originally `Maybe Nat`) is transformed into `Nat`. The clauses which return `Just x` are transformed into `x`. The recursive call `idx x ys p` is left untouched since it also returns `Nat` by virtue of the refactoring.

```
idx : (x : Nat)
    -> (xs : List Nat)
    -> Elem x xs
    -> Maybe  Nat
idx x (x :: ys) Here = Just  x
idx x (y :: ys) (There p) =
    case y == x of
    | True => Just  x
    | False => idx x ys p
```

(a) Before

```
idx : (x : Nat)
    -> (xs : List Nat)
    -> Elem x xs
    -> Nat
idx x (x :: ys) Here = x
idx x (y :: ys) (There p) =
    case y == x of
    | True => x
    | False => idx x ys p
```

(b) After

Fig. 13: Eliminating the `Maybe` return type from the function `idx`.

## 4 A Language Evaluator

In this section, we demonstrate the refactorings in § 3 on the evaluator for a small expression language in Fig. 14a. The evaluator operates in a standard way: natural numbers are treated as literals; variables are substituted for literals stored in an environment; and addition expressions are evaluated by adding the evaluation of its operands. The section proceeds as a series of steps that may be typically taken by a programmer to explore and enrich a program with dependent types. The goal is to use dependent types to produce a stronger version of `eval`. Presently, `eval` returns `Nothing` if a variable is not found in the environment, and will not produce a positive result for an environment that is not *covering*, i.e. containing all variables in a given expression. A stronger version of `eval` would remove the `Maybe` in its return type, guaranteeing a positive result for every expression and corresponding covering environment. We therefore illustrate use of the refactorings defined in § 3 via the elimination of this `Maybe`.

**Step 1: Introducing an index to `Expr`.** We begin by refining `Expr` so that it is indexed over a list, representing variables that are defined (or are in scope). This is achieved by applying *Adding an Index to a Data Type* (§ 3.1), providing the name of the type to be refactored, i.e. `Elem`, and both the type and name of the index, i.e. `List V` and `vars`, respectively. The refactored code is given in Fig. 14b. All three constructors now take an additional parameter, serving as the index `vars`. Usage sites within the type declaration, seen here in parameters to `Add`, are refactored to take an index, for which fresh parameters are introduced. Since the type signature of `eval` also represents a usage site, it now takes an additional argument. Constructor patterns are updated to include the new parameters. In the recursive calls, holes have been introduced and automatically resolved to `vars`, forced by the types of `e1` and `e2`.

**Step 2: Collapse index variables in `Expr`.** We refine the introduced index to `Expr`, enforcing the assumption that a given expression, $e$, is indexed by `vars` containing all variables occurring in $e$. This is achieved in two phases. Firstly,

```
data Expr : Type where
    | Num : (n : Nat) -> Expr
    | Var : (x : V) -> Expr
    | Add : Expr -> Expr -> Expr

lookupVar : (x : V) -> (env : List (V ** Nat)) -> Maybe Nat
lookupVar x [] = Nothing
lookupVar (MkV x) ((MkV y, val)::ys) = case isEqual x y of
    | True => Just val
    | False => lookupVar (MkV x) ys

eval : (env : List (V ** Nat)) -> Expr -> Maybe Nat
eval env (Num n) = Just n
eval env (Var x) = lookupVar x env
eval env (Add e1 e2) = case eval env e1 of
    | Nothing => Nothing
    | (Just e1') => case eval env e2 of
        | (Just e2') => Just (plus e1' e2')
        | Nothing => Nothing
```

(a) The basic language and evaluator.

```
data Expr : (vars : List V) -> Type where
    | Num : (n : Nat) -> (vars : List V) -> Expr vars
    | Var : (x : V) -> (vars : List V) -> Expr vars
    | Add : (vars1 : List V) -> Expr vars1
            -> (vars2 : List V) -> Expr vars2
            -> (vars : List V) -> Expr vars
...
eval : (env : List (V ** Nat)) -> (vars : List V) -> Expr vars
    -> Maybe Nat
eval env vars (Num n vars) = Just n
eval env vars (Var x vars) = lookupVar x env
eval env vars (Add vars v1 e1 v2 e2) = case eval env v1 e1 of
    | Nothing => Nothing
    | Just e1' => case eval env v2 e2 of
        | Just e2' => Just (plus e1' e2')
        | Nothing  => Nothing
```

(b) Step 1: Introduce a `List Nat` index to `Expr`.

Fig. 14: The basic expression language, evaluator, and the first refactoring step.

by ensuring that expressions of the form `Add v v1 e1 v2 e2` share their index, `v`, with their immediate subexpressions, `e1` and `e2`. Secondly, by relating the variable `x` and the index `v` in expressions of the form `Var x v`. In the first phase, we apply *Collapsing Constructor Parameters* (§ 3.4), passing as arguments: the name of the type, `Expr`; the name of the constructor, `Add`; and the indices to collapse, `vars`, `vars1`, and `vars2`. In the second phase, we apply *Relating Constructor Parameters* (§ 3.3), passing as arguments: the name of the type and constructor, `Expr` and `Var`; the parameters being related, `x` and `vars`; and the name of the type relating them, `Elem`. The result is given in Fig. 15a. The second and third equations of `eval` change as a consequence of both phases: in the second equation, `Var` has an additional parameter, `p`; and `Add` in the third equation has fewer parameters, reflecting the collapsing of its indices.

**Step 3: Relate the parameters of `lookupVar`.** In order to successfully strengthen `eval`, we must similarly apply *Eliminating `Maybe`* (§ 3.8) to its helper function, `lookupVar`. To this end, we first introduce two parameters and pattern match on both in each equation. This is achieved in three phases: *i)* introducing `vars`; *ii)* introducing `p`, a membership relation; and expanding `vars` and `p`. In the first phase, we call a standard refactoring *Introduce Function Parameter* [Li06] with arguments: function name, `lookupVar`; position where it should be added, `0`; and both the name, `vars`, and type, `List V`, of the parameter. The second phase follows similarly, introducing `(p : Elem x vars)` as the third parameter, where `x` is the variable to be found in `env`. The third phase comprises a call to *Expanding a Pattern Variable* (§ 3.6), specifying the name of the function and both `env` and `p` as arguments. As a consequence of the first two stages, the call sites in both `lookupVar` and `eval` are updated, introducing holes for the new arguments. We give the result of all three phases in Fig. 15b. Henceforth, we elide impossible equations for clarity. The holes `?h1`, `?h3`, and `?h4` in `lookupVar` can be filled manually by the programmer with `ys`, `ys`, and `p`, respectively. Although `?h2` is currently unsatisfied, future steps may alter this.

**Step 4: Relate environment and `Expr` index.** We next relate `vars` with the given environment, `env`, in both `env` and `lookupVar`. This step comprises three phases: *i)* adding a new list parameter, `vals`; relating `vars`, `vals`, and `env` by `p`; and *iii)* expanding the new relation, `p`. For this step, we assume the type declaration `Unzip` in Fig. 16, relating `vars` and `env`. In the first phase, we apply *Introduce Function Parameter* to introduce `(vars : List Nat)`, representing the list of values in `env`. In the second phase, we apply *Relating Function Parameters* (§ 3.5) to introduce `(p : Unzip env vars vals)` as a function parameter. Finally, we apply *Expanding a Pattern Variable* (§ 3.6) on `p`, which precipitates applications on `vals`, `env`, and `vars`. Note that this step may remove equations that were previously possible, e.g. two equations from `lookupVar`. As in Step 4, we manually fill in the holes. The resulting code is given in Fig. 16.

**Step 5: Remove `case` expressions in `lookupVar`.** We next remove the remaining hole in Fig. 16, via the removal of the tautologous `case` expression that contains it. This is achieved by applying *Eliminating Tautologous Cases* (§ 3.7) to the first equation of `lookupVar`. Concurrently, we may manually remove the

```
data Expr : (vars : List V) -> Type where
    | Num : Nat -> (vars : List V) -> Expr vars
    | Var : (x : V) -> (vars : List V) -> (p : Elem x vars)
            -> Expr vars
    | Add : (vars : List V) -> Expr vars -> Expr vars
            -> Expr vars
...
eval : (env : List (V ** Nat)) -> (vars : List V) -> Expr vars
    -> Maybe Nat
eval env vars (Num n vars) = Just n
eval env vars (Var x vars p) = lookupVar x env
eval env vars (Add vars e1 e2) = case eval env vars e1 of
    | Nothing => Nothing
    | Just e1' => case eval env vars e2 of
        | Just e2' => Just (plus e1' e2')
        | Nothing  => Nothing
```

(a) Step 2: Specialise `Var` and unify index variables in `Add`.

```
lookupVar : (vars : List V) -> (x : V) -> (p : Elem x vars)
            -> (env : List (V ** Nat)) -> Maybe Nat
lookupVar (y::ys) x (Here y ys) [] = Nothing
lookupVar (y::ys) x (There x ys p y) [] = Nothing
lookupVar (y::ys) (MkV x) (Here y ys) ((MkV z, val)::zs) =
    case isEqual x z of
        | True => Just val
        | False => lookupVar ?h1 (MkV x) ?h2 zs
lookupVar (y::ys) (MkV x) (There x ys p y) ((MkV z, val)::zs) =
    case isEqual x z of
        | True => Just val
        | False => lookupVar ?h3 (MkV x) ?h4 zs
```

(b) Step 3: Introduce parameter `(vars : List V)`; relate parameters using `(Elem x vars)` in `lookupVar`; expand `(Elem x vars)` pattern variables.

Fig. 15: Refactoring steps 2 and 3.

```
lookupVar : (vals : List Nat) -> (vars : List V) -> (x : V)
         -> (q : Elem x vars) -> (env : List (V ** Nat))
         -> (p : Unzip env vars vals) -> Maybe Nat
lookupVar (v::vs) (x::ys) (MkV x) (Here x ys) ((MkV x, v)::zs)
    (ConsUZ x v zs ys vs p) = case isEqual x x of
        | True => Just val
        | False => lookupVar vs ys (MkV x) ?h0 zs
lookupVar (v::vs) (y::ys) (MkV x) (There x ys q y) ((MkV y, v)::zs)
    (ConsUZ y v zs ys vs p) =
        lookupVar vs ys (MkV x) q zs p

eval : (vals : List Nat)
    -> (env : List (V ** Nat))
    -> (vars : List V)
    -> (p : Unzip env vars vals)
    -> Expr vars
    -> Maybe Nat
eval [] [] [] NilUZ (Num n []) = Just n
eval (v::vs) ((x, v)::env) (y::ys)
    (ConsUZ x v env ys vs p) (Var x (x::ys) (Here x ys)) =
        lookupVar (x::ys) x (Here x ys) ((k, n)::env)
eval (v::vs) ((y, v)::env) (y::ys) (ConsUZ y v env ys vs p)
    (Var x (y::ys) (There x ys q y)) =
        lookupVar (y::ys) x (There x ys q y) ((y, v)::env)
eval [] [] [] NilUZ (Var x [] q) =
    lookupVar [] x q []
eval (v::vs) ((y,v)::env) (y::ys)
    (ConsUZ y v env ys vs p) (Add (y::ys) e1 e2) =
        case eval (v::vs) (e::env) (y::ys)
            (ConsUZ y v env ys vs p) e1 of
        | Nothing => Nothing
        | (Just e1') =>
            case eval (v::vs) (e::env) (y::ys)
                (ConsUZ y v env ys vs p) e2 of
            | Nothing => Nothing
            | (Just e2') => Just (plus e1' e2')
eval [] [] [] NilUZ (Add [] e1 e2) =
    case eval [] [] [] NilUZ e1 of
    | Nothing => Nothing
    | (Just e1') => case eval [] [] [] NilUZ e2 of
        | (Just e2') => Just (plus e1' e2')
        | Nothing => Nothing
```

Fig. 16: Step 4: Introduce predicate `Unzip env vars ws`; expand pattern variables; manually fill remaining holes.

```
lookupVar : (vals : List Nat) -> (vars : List V) -> (x : V)
    -> (q : Elem x vars) -> (env : List (V ** Nat))
    -> (p : Unzip env vars vals) -> Maybe Nat
lookupVar (v::vs) (x::ys) (MkV x) (Here x ys) ((MkV x, v)::zs)
    (ConsUZ x v zs ys vs p) = Just val
lookupVar (v::vs) (y::ys) (MkV x) (There x ys q y) ((MkV y, v)::zs)
    (ConsUZ y v zs ys vs p) = lookupVar vs ps (MkV x) q ys p
```

Fig. 17: Step 5: Remove tautologous cases.

`case` expression in the second equation since its subject will only evaluate to `False`. The resulting definition of `lookupVar` can be found in Fig. 17.

```
eval : (vals : List Nat) -> (env : List (V ** Nat))
    -> (vars : List V) -> (p : Unzip env vars vals)
    -> (Expr vars) -> Nat
eval [] [] [] NilUZ (Num n []) = n
eval (v::vs) ((x, v)::zs) (x::ys) (ConsUZ x v zs ys vs p)
    (Var x (x::ys) (Here x ys)) = lookupVar (v::vs) (x::ys) x
        (Here x ys) ((x, v)::zs) (ConsUZ x v zs ys vs p)
eval (v::vs) ((y, v)::zs) (y::ys) (ConsUZ y v zs ys vs p)
    (Var x (y::ys) (There x xs q y)) = lookupVar (v::vs) (y::ys) x
        (There x xs q y) ((y, v)::zs) (ConsUZ y v zs ys vs p)
eval (v::vs) ((y,v)::zs) (y::ys) (ConsUZ y v zs ys vs p)
    (Add (y::ys) e1 e2) = case eval (v::vs) ((y,v)::zs) (y::ys)
            (ConsUZ y v zs ys vs p) e1 of
        | e1' => case eval (v::vs) ((y,v)::zs) (y::ys)
                (ConsUZ y v zs ys vs p) e2 of
            | e2' => plus e1' e2'
eval [] [] [] NilUZ (Add [] e1 e2) = case eval [] [] [] NilUZ e1 of
    | e1' => case eval [] [] [] NilUZ e2 of
        | e2' => plus e1' e2'
```

Fig. 18: Step 6: Eliminate `Maybe`.

**Step 6: Eliminate `Maybe` from `eval`** Given that `lookupVar` is incapable of returning `Nothing`, we are able to apply *Eliminating `Maybe`* (§ 3.8). As a result, this removes the remaining sources of `Nothing` from the base cases of `eval`. Accordingly, we are able to apply *Eliminating `Maybe`* to `eval`. The resulting definition, and final result, of `eval` is shown in Fig. 18

## 5 Related Work

The Haskell Refactorer, HaRe [Li06,Bro08,LTR05,BLT10], is a refactoring tool for Haskell 98. 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 [McB11] that transforms functions between similar simply-typed structures. 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 § 3.1. Presently, the principal difference between Wibergh's and our work lies in their respective target languages. Refactoring has also been explored in the context of security, using dependent types as a mechanism to drive the rewriting system [BBM+22].

There is a large body of work on proof-repair systems. This work typically focuses on the transformation of proofs across types, and can be used for both theorem proving and aiding developers in writing dependently typed programs [Ada15,Whi13]. For example, Ringer et al. [RYLG19] propose a refactoring plugin for Coq, allowing the automatic identification of, and transformations across, ornaments. Our work complements this technique by taking an *exploratory* approach to program enrichment, e.g. by facilitating development of the types and operations that are used for proof-repair. Moreover, our approach increases flexibility by enabling introduction of predicate terms (§ 3.5), decoupled from types, and via broadening applicability of refactorings narrowing (or widening) function scopes (§ 3.8). 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 [McB11] relate data types that are structurally similar but not necessarily equivalent. Notably, ornaments have been used in transforming theorems by both Ringer et al. [RYLG19] and Williams et al. [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]. As before, our approach is complementary: we aim to *guide* the programmer in effecting the transformations for which Robert derives subsequent repairs. Meanwhile, Boite focuses on the specific case of propagating changes after adding constructors to data types [Boi04]. Johnsen and Lüth accommodates proof reuse by generalising theorems by abstracting their proofs [JL04].

There has been substantial work on *structural editing*, a related topic to refactoring. However, structural editing focuses on helping developers write new code, as opposite to structurally modifying existing, legacy code bases. A common problem in structural editing is the relation between linear editing and structural

hierarchical representations of the program source-code. This does not apply to refactoring tooling, which must work over an abstract syntax representation of the program source. Recent work includes Hüttel et al. [HELNGS+23] who describe a new technique for structural editing, allowing type-safe copy and paste functionality. In [MBO22], Moon et al. introduce `tylr` for tile-based structural editing, with the idea to eliminate the need for linear editing and instead use a tile-based representation approach, and Gopinathan [Gop22] introduces a structural editing plugin for OCaml. A recent development is to provide users with a *semantic difference* between code versions. The `https://semanticdiff.com` system gives one implementation of this approach, which, in particular, is able to recognise when some refactorings have taken place.

## 6 Conclusions and Future Work

In this paper, we introduced new refactorings for dependently typed programs, defined for the *Fluid* programming language. 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 applying a series of 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`, thereby demonstrating transformations facilitating conversion of a partial function (i.e. that can return `Nothing`) into a total equivalent that always produces a literal. This example demonstrated that, using our refactoring approach, developers can explore the use of dependent types to develop stronger and safer programs.

Although proving correctness of the presented refactorings is outwith the scope of this paper, the question of correctness represents an interesting avenue of future work. Refactorings for dependently typed programs, as explored in this paper, suggest a greater focus on the narrowing or widening of a function's scope. Consequently, some refactorings, such as eliminating a `Maybe` type (Section § 3.8), represent a departure from the standard refactoring correctness property of the original and refactored programs reducing to the same normal form, or set thereof. This suggests that any correctness property must account for *acceptable changes* to functionality, e.g. subsets of behaviours possibly identified by a simulation relation [San09]. Other future directions include exploring the use of ornaments for refactoring and the use of generic traversal libraries to aid implementation and formalisation [AAC+21].

**Acknowledgements** This work was supported by UK EPSRC, EP/V006290/.

## References

AAC+21. Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. a Type- and Scope-safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *J. Funct. Program.*, 31:e22, 2021.

Ada15. Mark Adams. Refactoring Proofs with Tactician. In *SEFM Workshops*, vol. 9509 of *LNCS*, pp. 53–67. Springer, 2015.

BBM+22. Christopher Brown, Adam D. Barwell, Yoann Marquer, Olivier Zendra, Tania Richmond, and Chen Gu. Semi-automatic Ladderisation: Improving Code Security through Rewriting and Dependent Types. In *PEPM*, pp. 14–27. ACM, 2022.

BD77. Rod M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *J. ACM*, 24(1):44–67, 1977.

BLT10. Christopher Brown, Huiqing Li, and Simon J. Thompson. An Expression Processor: a Case Study in Refactoring Haskell Programs. In *TFP*, vol. 6546 of *LNCS*, pp. 31–49. Springer, 2010.

Boi04. Olivier Boite. Proof Reuse with Extended Inductive Types. In *Int. Con. on Theorem Proving in HOL*, pp. 50–65. Springer, 2004.

BP01. Gilles Barthe and Olivier Pons. Type Isomorphisms and Proof Reuse in Dependent Type Theory. In *FoSSaCS*, vol. 2030 of *LNCS*, pp. 57–71. Springer, 2001.

Bra21. Edwin C. Brady. Idris 2: Quantitative Type Theory in Practice. In *ECOOP*, vol. 194 of *LIPIcs*, pp. 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

Bro08. Christopher Mark Brown. *Tool Support for Refactoring Haskell programs*. PhD thesis, University of Kent, UK, 2008.

BT10. Christopher Brown and Simon J. Thompson. Clone Detection and Elimination for Haskell. In *PEPM*, pp. 111–120. ACM, 2010.

CDM13. Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *CCP*, pp. 147–162. Springer, 2013.

dMKA+15. Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *CADE*, vol. 9195 of *LNCS*, pp. 378–388. Springer, 2015.

DMS12. Maxime Dénès, Anders Mörtberg, and Vincent Siles. A Refinement-Based Approach to Computational Algebra in Coq. In *ITP*, pp. 83–98. Springer, 2012.

Dyb94. Peter Dybjer. Inductive Families. *For. Asp. Comp.*, 6(4):440–465, 1994.

Fou. Eclipse Foundation. Eclipse. `http://www.eclipse.org/`. 2023.

Fow99. Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999.

Gop22. Kiran Gopinathan. GopCaml: A Structural Editor for OCaml. `https://arxiv.org/abs/2207.07423`, 2022.

HELNGS+23. Hans Hüttel, Anja Elisasen Lumholtz Nielsen, Nana Gjerulf Sandberg, Christoffer Lind Andersen, and Peter Mikkelsen. A Structure Editor with Type-safe Copy/Paste. In *IFL*, ACM, 2023.

Hur95. Antonius J. C. Hurkens. A Simplification of Girard's Paradox. In *TLCA*, vol. 902 of *LNCS*, pp. 266–278. Springer, 1995.

JL04. Einar Broch Johnsen and Christoph Lüth. Theorem Reuse by Proof Term Transformation. In *Int. Con. on Theorem Proving in HOL*, pp. 152–167. Springer, 2004.

Lam13. Peter Lammich. Automatic Data Refinement. In *ITP*, pp. 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, UK, 2006.

LT12. Huiqing Li and Simon J. Thompson. A Domain-Specific Language for Scripting Refactorings in Erlang. In *FASE*, vol. 7212 of *LNCS*, pp. 501–515. Springer, 2012.

LTR05. Huiqing Li, Simon J. Thompson, and Claus Reinke. The Haskell refactorer, HaRe, and its API. In *LDTA*, vol. 141 of *ENTCS*, pp. 29–34. Elsevier, 2005.

MBO22. David Moon, Andrew Blinn, and Cyrus Omar. tylr: a Tiny Tile-based Structure Editor. In *TyDe*, pp. 28–37. ACM, 2022.

McB00. Conor McBride. Elimination with a Motive. In *TYPES*, pp. 197–216. Springer, 2000.

McB11. Conor McBride. Ornamental Algebras, Algebraic Ornaments. `https://tinyurl.com/yc7wmb2s`, 2011.

Met. Meta. Retrie: 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 *Adv. Funct. Program.*, vol. 5832 of *LNCS*, pp. 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, 2015.

Rin21. Talia Ringer. *Proof Repair*. PhD thesis, University of Washington, 2021.

Rob18. Valentin Robert. *Front-end Tooling for Building and Maintaining Dependently-Typed Functional Programs*. PhD thesis, 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 *PLDI*, pp. 112–127. ACM, 2021.

RYLG19. Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for Proof Reuse in Coq. In *ITP*, vol. 141 of *LIPIcs*, pp. 26:1–26:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

San09. Davide Sangiorgi. On the Origins of Bisimulation and Coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009.

ST08. Nik Sultana and Simon Thompson. Mechanical Verification of Refactorings. In *PEPM*. ACM, 2008.

TH23. Simon J. Thompson and Dániel Horpácsi. Refactoring = Substitution + Rewriting: Towards Generic, Language-Independent Refactorings. In *Eelco Visser Commemorative Symposium*, vol. 109 of *OASIcs*, pp. 26:1–26:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

Coq. The Coq Development Team. Coq. `https://coq.inria.fr` 2017

WDR14. Thomas Williams, Pierre-Évariste Dagand, and Didier Rémy. Ornaments in practice. In *WGP*, pp. 15–24. ACM, 2014.

Whi13. Iain Whiteside. *Refactoring Proofs*. PhD thesis, University of Edinburgh, UK, 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 *Con. on Intell. Comp. Math.*, 2015.