Towards a Refactoring Tool for Dependently-Typed Programs *Extended Abstract*

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

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

Simon Thompson
School of Computing
University of Kent
Canterbury, UK
s.j.thompson@kent.ac.uk

Susmit Sarkar   Edwin Brady
School of Computer Science
University of St Andrews
St Andrews, UK
Susmit.Sarkar@st-andrews.ac.uk   ecb10@st-andrews.ac.uk

While there is considerable work on refactoring functional programs, so far this had not extended to *dependently-typed* programs. In this paper, we begin to explore this space by looking at a range of transformations related to indexed data and functions.

1 Introduction

Dependently-typed programming languages are becoming increasingly popular in the functional programming community, allowing programmers to express logical properties and their proofs as part of a program. Languages such as Agda [13], Idris [3] and Coq [2] have gained particular prominence in the past decade. Dependent types permit the programmer to implement safe programs, by encoding safety properties as parts of the types themselves; these properties can be encoded by permitting the types to *depend* on values. However, programming dependently-typed systems is still very difficult for the average software developer, who may have had some prior training with functional languages, such as Haskell, but lacks the background and experience in logic and type systems to make appropriate use of the dependent types to build strong safe systems as a part of their day-to-day practice.

Refactoring [14, 9], the process of changing the structure of a program without changing its behaviour, has a long history, going back to the 1977 Fold/UnFold system [6], proposed by Burstall and Darlington. Refactoring aims to provide programmers with the ability to systemically restructure their code to better reflect its purpose, making it more amenable to change and maintainability, or simply to make it more understandable. Developers often make these structure changes manually, but it is also possible to use a refactoring tool that attempts to automate the transformation. Many refactoring tools have been built for a variety of different languages, such as Java [9], C++ [15], Haskell [11, 5, 4] and Erlang [10]. The advantage of refactoring tool support is that it not only presents the programmer with a structured set of well-defined transformations, but it also alleviates the error-prone burden of manually making modifications across an entire project that might contain thousands of source code modules.

To date, however, there are no refactoring tools for dependently-typed programming languages. Indeed, even though several refactorings have been proposed and even implemented for (non-dependently-typed) functional programs, there is no taxonomy of refactorings specifically for dependently-typed

Submitted to:
WPTE 2022

programs. In this paper, we address this problem by discussing a number of new refactorings to be implemented for the dependently-typed programming language, *Pi-Forall*[1] developed by Weirich. We particularly focus on refactorings for extending and maintaining data types in *Pi-Forall*.

A key characteristic of dependently-typed programming is to use value-dependent types to refine types, and an example of this to help intuition is replacing lists of arbitrary length by vectors of a known length: the additional parameter – here a natural number – is often known as an *index*. The examples we discuss include:

- introducing a new index to an existing data type, with the consequence of modifying functions over the existing data type to be able to take possible proof terms to satisfy the proof obligations generated by indexing;
- refinement of existing indices, including refactorings to add new parameters to constructors of data types, such as proof obligations affecting indices, unification of indices across constructor parameters, and refining indices for particular constructor cases; and
- refactoring of the functions and expressions over the original data types.

We demonstrate our refactorings over a small language implementation, presented in Section 3. In the longer term, our goal is to develop a new refactoring tool, *RePi*, that gives the programmer effective tool-support to assist in the development of dependently-typed programming.

2 Background

2.1 Refactoring

Refactoring is the process of changing the structure of a program whilst preserving its functional semantics. Refactoring can, *inter alia*, increase code quality, programming productivity, and code reuse. The term refactoring was first introduced by Opdyke [14], and the concept goes at least as far back as Burstall and Darlington's fold/unfold system [6]. In order to facilitate refactoring, a variety of refactoring tools have been developed [10, 11, 8, 16], enabling source-to-source transformations to be applied semi-automatically under the programmer's guidance. Refactorings provided by such tools often target value-level constructs, such as renaming, lambda lifting, and generalising a definition, but can also provide refactorings that transform data type definitions, e.g. introducing a new constructor or grouping similar constructors together[2].

2.2 Dependent Types

Dependently-typed languages feature type systems that are enriched such that types can depend on values. A key feature of full-spectrum dependently-typed languages, e.g. Agda [13], Idris [3], and *Pi-Forall* (Section 2.3), is that data declarations can have indices. Unlike parameters, e.g. for polymorphism, indices allow the programmer to further constrain the set of values that a given type represents, e.g. restricting lists (or *vectors*) to a given length. Given these restrictions, the use of indices can significantly affect the structure of the overall program, e.g. fetching an element in a vector by its position does not require the function to countenance failure (as in Haskell) when the types ensure that the position being accessed is less than the length of the vector. Introducing, or otherwise transforming, such indices can

[1]`https://github.com/sweirich/pi-forall`

[2]`https://www.cs.kent.ac.uk/projects/refactor-fp/catalogue/Layered.html`

therefore be seen as directly affecting the structure of the program, and can require many subsequent, and potentially *non-local*, transformations.

2.3 *Pi-Forall*

Pi-Forall is a dependently typed programming language first developed for a series of lectures at the Oregon Programming Languages Summer School 2013 [1], and has subsequently featured in Weirich's keynote talk at Compose 2015 [18]. Built as a teaching aid, it intentionally implements a simple, but core, dependently-typed calculus. It is implemented in Haskell and comprises both syntax and a type checker. The relatively small size of *Pi-Forall* (e.g. when compared with Idris or Agda), whilst simultaneously incorporating key features of larger dependently-typed languages, makes it an ideal target for our prototype refactoring tool.

3 Refactorings for Indexing Data Types

This section proposes new refactorings to be implemented in *RePi* for *Pi-Forall*. Although the examples and concepts of the refactorings are themselves couched in *Pi-Forall*, the concepts and techniques described here are, in fact, just as applicable to other dependently-typed languages. We focus on a preliminary set of refactorings that transform data types, using the running example of a λ-calculus interpreter. The refactorings are characteristic of changes made in initial exploration of encoding such interpreters in dependently-typed languages. Throughout the presentation to avoid the overloading of data type and constructor names, we manually append the refactored data type and its constructors with a version in order to differentiate between refactored versions in the examples given. We intend the refactorings presented to be semi-automatic, in that they may require additional parameters from the user to proceed.

3.1 Adding an Index to a Data Type

The index to be added is introduced in the first indexing position for the data type (σ) in question. The type, τ, of the introduced index is provided. All constructors that have an argument of type σ are refactored to pass in a variable, ε, so that σ is applied to ε, in all positions of the constructor that originally referenced σ. All functions over the type σ must also be transformed, so that their type signatures represent the transformed σ; all pattern matching over constructors of σ must be transformed; all recursive calls must be transformed to take into account the introduced variable; all constructions of σ in the body of the function must also be transformed to take into account the new constructor parameters.

Example As an example, consider the *Pi-Forall* program in Figure 1. The example in the *Before* segment of the figure (Figure 1a) shows the syntax data type, `AST`, for a minimal λ-calculus with numbers (`NumA`), variables (`VarA`), applications (`AppA`) and lambdas (`LamA`). An evaluator for the language is given on Line 7, which reduces terms in the standard way. Adding an index to the data type `AST` is shown in the *After* segment (renamed to `AST2`), where an index parameter, `a` of type `List Nat` is added to the data type. This may arise in trying to index the `AST` with a context. All corresponding constructors of the type are adjusted to reflect this new data type parameter. The `eval` function is also refactored by changing its type to take an erasable argument `vars : List Nat`, and by indexing any `AST` arguments. Pattern matching clauses are updated, as are recursive calls.

```
data AST2  (a : List Nat) : Type where
 NumA2 of (c : Nat)
 VarA2 of (a : Nat)
 AppA2 of (vars1 : List Nat)
          (vars2 : List Nat)
          (e1 : AST2 vars1)
          (e2 : AST2 vars2)
 LamA2 of (vars1 : List Nat)
          (a : Nat)
          (body : AST2 vars1)

eval : [vars : List Nat] -> Env ->
       AST2 vars
    -> Maybe Nat
eval = \
      [vars] env term . case term of
 NumA2 c -> Just c
 VarA2 v -> lookup v env
 AppA2 vars1 vars2 e1 e2 ->
  case e1 of
   LamA2 vars3 a body ->
    case (eval [vars2] env e2) of
     Nothing -> Nothing
     Just e2V ->
        eval [vars3]
           (Cons (Tup a e2V ) env) body
   _ -> Nothing
 LamA2 vars1 a body -> Nothing
```

```
data AST  : Type where
 NumA of (c : Nat)
 VarA of (a : Nat)
 AppA of (e1 : AST) (e2 : AST)
 LamA of (a : Nat) (body : AST)

eval : Env -> AST -> Maybe Nat
eval = \env term . case term of
 NumA c -> Just c
 VarA v -> lookup v env
 AppA e1 e2 ->
  case e1 of
   LamA a body ->
    case (eval env e2) of
     Nothing -> Nothing
     Just e2V -> eval (Cons (Tup a e2V ) env) body
   _ -> Nothing
 LamA a body -> Nothing
```

(a) Before

(b) After

Figure 1: Adding an index to the AST data type, before and after the refactoring.

3.2 Unifying Constructor Indices

This refactoring allows multiple data type parameter instances to be unified within a constructor. A constructor is selected, and the parameter positions to be unified are identified. Any functions over the data type must be refactored in order to correspond with the unified indices. This may not always be possible.

Example Consider the example in Figure 2. In this example, we have refactored two constructors of the original AST data type from Figure 1. Firstly AppA has been refactored to unify the data type indices: the result is that vars1 is now passed in both places as a parameter to AST. The LamA constructor has

```
data AST3 ... where
...
  AppA3 of (vars1 : List Nat)
          (e1 : AST3 vars1)
          (e2 : AST3 vars1)
  LamA3 of (vars3 : List Nat) (a : Nat)
          (body : AST3 (Cons a vars3))

...
  AppA3 vars1 e1 e2 ->
   case (eval3 [vars1] env e2) of
    Nothing -> Nothing
     Just e2V ->
      case e1 of
       LamA3 vars3 a body ->
        eval [Cons a vars3]
          (Cons (Tup a e2V ) env) body
       _ -> Nothing
...
```

Figure 2: After unifying indices in the AppA constructor, and constraining the index of the LamA

```
 1  data ASTEn  (v : List Nat) : Type where
 2   ...
 3   VarAE of (vars : List Nat) (a : Nat)
 4            (prf : Elem Nat a vars)
 5   ...
 6
 7  convert : (vars : List Nat) -> (a : AST) -> Maybe (ASTEn vars)
 8  convert = \vars a . case a of
 9   NumA c -> Just (NumAE c)
10   VarA a ->
11    case (isElem a vars) of
12     Yes prf -> Just (VarAE vars a prf)
13     No t con -> Nothing
14   AppA a b ->
15    case (convert vars a) of
16     Nothing -> Nothing
17      Just a1 ->
18       case (convert vars b) of
19        Nothing -> Nothing
20        Just b1 -> Just (AppAE vars a1 b1)
21   LamA a body ->
22    case (convert (Cons a vars) body) of
23     Nothing -> Nothing
24     Just body2 -> Just (LamAE vars a body2)
25
26  eval : [vars : List Nat] -> Env -> ASTEn vars -> Maybe Nat
27  eval = \[vars] env term . case term of
28   ...
29     VarAE vars v prf -> lookup v env
30   ...
```

Figure 3: After adding an `Elem` proof constraint to the variable constructor

also been refactored so that the index parameter on Line 6 is changed to a construction clause: `Cons a vars3`.

3.3 Add a Proof Obligation to a Data Constructor

The refactoring adds a new parameter to a data constructor that represents a proof obligation that needs to be satisfied. The proof obligation will be represented as a dependent type, and the proof itself will be supplied as a member of that type in the expressions where the constructor is used. Suppose a type τ has a constructor α, and we wish to add a proof obligation that satisfies the type $\Delta\gamma$ (i.e. the type Δ with γ as an index), the refactoring would add a new parameter to the constructor to bind γ if it is not already declared, and then add the proof in the final argument position.

Example As an example, consider the *Pi-Forall* program in Figure 3. Here, we visit the variable (`VarAE`) constructor case for `AST` where we wish to add a membership proof obligation that the variable `a` is a member of `vars`. This is useful to build a variant of `AST` that we can classify as enriched: forming a syntax that also keeps the scoping of variables as part of its representation. We can provide such a proof via the `Elem` type, which, on Line 4, we use to provide a proof obligation that `a` is a member of `vars`; here, `Nat` is simply a type parameter passed to `Elem`. The refactoring also adds the binding for `vars`. All expressions and functions over this data type also need to be transformed. In our example, we also must refactor our `eval` function to pattern match on the proof; this is demonstrated on Line 29. In this example we also employ the use of a conversion function that provides an interface between the original `AST` definition from Figure 1, called `convert`. We will make more use of this function in Section 3.4, but for now it provides a useful function for converting between the original syntax and its enriched version. We so far envisage this function to be supplied by the programmer. However, it might be possible for such a function to be semi-automatically generated, as it contains a large amount of boilerplate code that

```
 1  data ASTS  (v : SnocList Nat) : Type where
 2   NumA of (c : Nat)
 3   VarA of (vars : SnocList Nat) (a : Nat)
 4           (prf : SnocElem Nat a vars)
 5   AppA of (vars1 : SnocList Nat)
 6           (e1 : ASTS vars1)
 7           (e2 : ASTS vars1)
 8   LamA of (vars3 : SnocList Nat) (a : Nat)
 9           (body : ASTS (SCons vars3 a))
10
11  convert : (vars : SnocList Nat)
12         -> (a : AST) -> Maybe (ASTS vars)
13  convert = \vars a . case a of
14   NumA c -> Just (NumAS c)
15   VarA a ->
16    case (isElemSnoc a vars) of
17     Yes prf -> Just (VarAS vars a prf)
18     No t con -> Nothing
19   AppA a b ->
20    case (convert vars a) of
21     Nothing -> Nothing
22     Just a1 ->
23      case (convert vars b) of
24       Nothing -> Nothing
25       Just b1 -> Just (AppAS vars a1 b1)
26   LamA a body ->
27    case (convert (SCons vars a) body) of
28     Nothing -> Nothing
29     Just body2 -> Just (LamAS vars a body2)
30
31  eval : [vars : SnocList Nat] -> Env
32       -> ASTS vars -> Maybe Nat
33  eval = \[vars] env term . case term of
34   NumAS c -> Just c
35   VarAS vars v prf -> lookup v env
36   AppAS vars1 e1 e2 ->
37    case (eval [vars1] env e2) of
38     Nothing -> Nothing
39     Just e2V ->
40      case e1 of
41       LamAS vars3 a body ->
42        eval [SCons vars3 a]
43           (Cons (Tup a e2V ) env) body
44       _ -> Nothing
45  LamAS vars1 a body -> Nothing
```

Figure 4: Refactoring `List` to `SnocList`

converts between structures, apart from Lines 12–14 which have to scrutinised by a decision procedure to provide a proof obligation. This `convert` function is not unlike *ornaments* [12], providing mechanisms to convert between different types. Generation of these functions is an avenue of future work.

3.4 Transforming Index Types

In our final transformation, we propose a refactoring that allows the transformation of an index type. Suppose we have a data type `foo` with index `bar : T1`, the refactoring allows us to transform `T1` into a supplied type, `T2`. This requires all constructors to be refactored to transform the index into an index over the new type, and all functions and expressions must correspond with the new index type. This requires types signatures of functions over `Foo T1` to be transformed into `Foo T2`. Cases where indices are constructed in the original program must be refactored into constructed cases for the new type.

Example As an example, consider the *Pi-Forall* program in Figure 4. Here the refactoring has changed the original `List Nat` index of `AST` into a `SnocList`, to match usual presentations where the newest bound variable is at the end. Each constructor referencing the original index in the `AST` data type has been refactored to reference `SnocList` instead. The original construction in the `LamAS` case has been replaced to use `SCons` from the `SnocList` type. The conversion function, on Line 11, is refactored to use `SnocList` and most of the refactoring proceeds in an expected way. The original `eval` function is also refactored to reference the new index type.

4 Related Work

Though there is a considerable body of work on refactoring typed functional programs, this has not been carried over to dependent types. An overview of the scope of this kind of refactoring, as well as the technologies that support it, is given by Li and Thompson [17].

The discussion of indexed types given here is related to the theory of *ornaments*, as developed by McBride [12], and applied to more practical programming problems by Williams, Dagand and Rémy [20], who characterise ornaments informally thus: "a data type ornaments another if they both share the same recursive skeleton" which allows programs that only operate on that underlying structure to be transferred from one to the other.

On the other hand, there is little work on refactoring of proofs. An exception is the work of Aspinall, Dietrich, and Whiteside [19, 7]; although this can be seen through the lens of "propositions as types; proofs as programs", the thrust of this work is to see the refactorings from the logical point of view, using that terminology and mindset.

5 Conclusions and Future Work

This work represents a first step towards our longer-term goal: to build a refactoring tool for dependently-typed programs in the *Pi-Forall* system. We chose *Pi-Forall* because it contains the essential features of dependent types, despite being small in comparison with other languages in the area, including Agda, and indeed Idris. Building a prototype tool for *Pi-Forall* should expose us to the main challenges of the area, and should also allow any insights to be transferred to building tools for these better-known systems. We will focus on developing refactorings that facilitate the introduction and transformation of programming constructs, which leverage the unique features offered by dependently-typed languages. The ultimate aim of such a refactoring tool is to assist the programmer in taking advantage of the potential safety guarantees that dependent types represent, and to enable programmers to more easily and safely effect potentially wide-ranging changes over diverse and complex programs.

Acknowledgements

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

References

[1] Amal Ahmed, Robert Harper, Dan Licata, Greg Morrisett, Simon Peyton-Jones, Frank Pfenning, Andrew Tolmach, Stephanie Weirich & Steve Zdancewic (2013): *Oregon Programming Languages Summer School*. Available at `https://www.cs.uoregon.edu/research/summerschool/summer13/curriculum.html`.

[2] Yves Bertot & Pierre Castéran (2013): *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[3] Edwin Brady (2016): *Type-driven Development With Idris*. Manning. Available at `http://www.worldcat.org/isbn/9781617293023`.

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

[5] Christopher Brown, Huiqing Li & Simon Thompson (2011): *An Expression Processor: A Case Study in Refactoring Haskell Programs*. In Rex Page, Zoltán Horváth & Viktória Zsók, editors: *Trends in Functional Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 31–49.

[6] R. M. Burstall & John Darlington (1977): *A Transformation System for Developing Recursive Programs*. *J. ACM* 24(1), p. 44–67, doi:10.1145/321992.321996. Available at `https://doi.org/10.1145/321992.321996`.

[7] Dominik Dietrich, Iain Whiteside & David Aspinall (2013): *Polar: A Framework for Proof Refactoring*. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, *Lecture Notes in Computer Science* 8312.

[8] Facebook (2020): *Retrie*. Available at `https://github.com/facebookincubator/retrie/`.

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

[10] Huiqing Li & Simon Thompson (2011): *A User-extensible Refactoring Tool for Erlang Programs*. Technical Report 4-11, University of Kent. Available at `http://www.cs.kent.ac.uk/pubs/2011/3171`.

[11] Huiqing Li, Simon Thompson & Claus Reinke (2005): *The Haskell Refactorer: HaRe, and its API*. In John Boyland & G´orel Hedin, editors: *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, pp. 182–196. Available at `http://www.cs.kent.ac.uk/pubs/2005/2158`. Published as Volume 141, Number 4 of Electronic Notes in Theoretical Computer Science, http://www.sciencedirect.com/science/journal/15710661.

[12] Conor McBride (2010): *Ornamental algebras, algebraic ornaments*. *Journal of Functional Programming* 47.

[13] Ulf Norell (2008): *Dependently Typed Programming in Agda*. In: *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, Springer-Verlag, Berlin, Heidelberg, p. 230–266.

[14] William F. Opdyke (1992): *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. Ph.D. thesis, University of Illinois at Urbana-Champaign.

[15] William F. Opdyke (1999): *Refactoring C++ Programs*. Technical Report, Lucent Technologies/ Bell Labs. Available at `st-www.cs.uiuc.edu/users/opdyke/wfo.990201.c++.refac.html`.

[16] Reuben NS Rowe & Simon J Thompson (2017): *Rotor: First steps towards a refactoring tool for ocaml*. In: *OCaml Users and Developers Workshop 2017*.

[17] Simon Thompson & Huiqing Li (2013): *Refactoring tools for functional languages*. *Journal of Functional Programming* 23(3).

[18] Stephanie Weirich (2015): *Pi-Forall: How to use and implement a dependently-typed language*. Available at `https://www.composeconference.org/2015/summary/#weirich`. Technical keynote at Compose 2015.

[19] Iain Whiteside, David Aspinall, Lucas Dixon & Gudmund Grov (2011): *Towards Formal Proof Script Refactoring*. In: *Proceedings of the 18th Calculemus and 10th International Conference on Intelligent Computer Mathematics*, MKM'11, Springer-Verlag, Berlin, Heidelberg, pp. 260–275.

[20] Thomas Williams, Pierre-Évariste Dagand & Didier Rémy (2014): *Ornaments in Practice*. In: *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, Association for Computing Machinery, New York, NY, USA, p. 15–24, doi:10.1145/2633628.2633631. Available at `https://doi.org/10.1145/2633628.2633631`.