Proving correctness of refactoring tuples to records

A correct-by-construction approach on a Haskell-like language

Jeroen Bastenhof[1]

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

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

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Jeroen Bastenhof
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Luka Miljak, Koen Langendoen

Abstract

Refactoring is a useful tool for increasing the overall quality of software without making changes to how it interacts with the environment. To verify that a refactoring operation correctly transforms an expression, one can provide a formal proof. Using Agda, a dependently-typed language, as a proof assistant, we investigate the feasibility of proving the correctness of refactoring tuples to records for a small-scale language that shares similarities with Haskell. We construct this language in Agda using intrinsically-typed terms and define an accompanying refactoring function for refactoring tuples to records. We prove that the refactoring is well-typed and that it replaces all tuple occurrences. Big-step semantics are used to show the relation between the intrinsically-typed language and its resulting output value. Additionally, we show that we can construct a relation between the values of an expression before and after refactoring. By presenting these proofs we gain more insights into the feasibility of proving the correctness of tuple to record refactoring. Furthermore, we argue that the proofs given for this small-scale language can serve as inspiration for proving comparable properties of the refactoring in the context of Haskell and beyond.

1 Introduction

Whenever one refactors software, they do that to improve the structure and design of a program without altering its external behaviour [1]. Extensive testing is an approach that is able to increase the confidence that the external behaviour remains unaltered after applying the refactoring. However, doing so provides no explicit guarantees as opposed to a formal verification of its correctness [2].

According to a study conducted in 2021 by AlOmar et al. [3], previous research primarily focused on formally proving the correctness of common refactoring operations on class-based, object-oriented programming languages like Java and C++. Amongst these refactoring operations, the pull-up/down and renaming transformations were observed to be the most popular. The formal verification of refactoring operations' correctness in functional programming languages has received relatively less attention and research focus compared to their counterparts in class-based, object-oriented languages. As such, the coverage of commonly performed or language-specific refactoring operations in this domain is comparatively sparse or absent.

The objective of this research is to explore *the feasibility of proving the correctness of the refactoring process that transforms tuples into records for a Haskell-like language*, to which we will refer as *record refactoring*. A similar refactoring operation that targets the functional programming language Erlang is described by Lövei et al. [4]; however, the correctness of this operation remains unproven.

To keep the proofs manageable in terms of size and complexity, we work with a simplified subset of Haskell that we refer to as the *Haskell-like* language. Only the most important aspects that relate to record refactoring are retained to allow us to reason about record refactoring in a larger context while preserving the validity of the work.

Using Agda [5], a *dependently typed* programming language, as a proof assistant, the aim is to model a small language on which the verification of the refactoring can be performed. By using a proof assistant, we are less likely to make mistakes while constructing formal proofs. This is because a proof assistant aids us in finding edge cases that might otherwise go unnoticed in pen and paper-based proofs. These edge cases are found by checking that all steps that a proof is made of follow the rules of the specified logic. Additional background information on Agda is provided in section 2.2.

The following contributions, which provide insights into the feasibility of proving record refactoring correctness, are presented in this paper.

- An intrinsically-well-typed Haskell-like language with representations for tuples and records is given that serves as the foundation for the refactoring operation as well as all proofs that follow this refactoring (section 3).
- Big-step semantics are provided that describe the behaviour of a language by relating terms to values (section 4).
- A refactoring transformation from tuples to records is given that creates new record declarations and replaces all tuple occurrences with an instance of a declared record (section 5).
- A proof is given that shows that the refactoring operation does not break well-typedness (included in section 5 on refactoring and follows from the use of an intrinsically-well-typed language).
- A proof is given that shows that the refactoring operation eliminates all tuples from an expression (section 6).
- A relation between the values of the big-step semantics of an expression before and after refactoring is provided (section 7).

We discuss the contents of this paper with regard to the language specification and its relation to Haskell in section 9.

2 Background information

This section aims to give more detailed background information on the problem that this paper is concerned with. We do this by elaborating on the refactoring operation and the tooling that is used.

2.1 Refactoring tuples to records

Tuple-to-record refactoring is a technique that can be applied to existing software. It works by constructing an object declaration, using record syntax, where the types of its attributes match that of the tuple elements. The record syntax for declaring objects requires an identifying name for the type as well as the accessors for its fields. After declaring the new object, all tuple instances with the same type signature can be replaced with the new construct.

An example of the type of refactoring we are concerned with is shown in listing 1 and largely inspired by an example listed in work by Miran [6]. It is apparent that seeing `(String, String, Int)` does not tell us anything about the structure of the data, whereas seeing `Person` immediately tells us we are working with data that resembles a person. The astute reader might object to this claim arguing that we can solve this particular problem by constructing a type alias for the tuple, `type Person = (String, String, Int)`, and their objection undeniably holds true. However, using a type alias is not enforced by the type system, which might lead to inconsistent use. To enforce this, `newtype` should be used instead, for which we would have to define accessors ourselves. Additionally, if we were to extend the `Person` type in the future, updating the record approach would be more manageable since the fields provide a level of abstraction.

A similar refactoring operation for the functional programming language Erlang, where groups of related data are converted to records, is described by Lövei et al. [4]. However, they do not provide a formal proof of the correctness of the refactoring operation. This appears to be a recurring problem in the existing literature where more effort is directed towards proving the correctness of refactoring operations for class-based, object-oriented languages as opposed to functional languages [3].

```
-- Before: less expressive and extensible
lastName :: (String, String, Int) -> String
lastName (_, s, _) = s

-- After: more expressive and extensible
data Person = Person { initials :: String
                     , lastName :: String
                     , age :: Int
                     }
```

Listing 1: Tuple to record refactoring example based on work by Miran [6].

2.2 Agda

Agda is a *dependently typed* functional programming language. Dependent typing allows types to depend on terms as opposed to having a clear separation between types and values [7]. To give a brief impression of what dependent typing is about, we refer to a commonly used example of constructing a `Vec A n` (see listing 2), which is a list annotated by its length [7; 8; 9].

```
data Vec (A : Set) : ℕ → Set where
    []  : Vec A zero
    _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Listing 2: Dependent type example (Vec).

The vector from listing 2 is a type that is *indexed* on its length as indicated by `ℕ → Set`. Thus, $\forall n \in \mathbb{N}$, `Vec A n` is a valid type. The `A` is referred to as a *parameter* of `Vec A n`. The type can be constructed by using one of its *constructors*. The `[]` constructor creates the empty vector where the size is forced to zero. The other constructor, `_::_`, takes a value of type `A` and inserts it into a vector of arbitrary length, therefore increasing its length by 1.

Due to the vector being indexed by its length, we can construct a function that allows us to retrieve the head of a *non-empty* list. Listing 3 uses pattern matching to extract an element from a vector that contains *at least* one element. Furthermore, we can easily extract the length of a vector by constructing an *implicit argument* that has a value identical to the index $\mathbb{N}$, as shown in listing 4.

```
head : ∀ {A n} → Vec A (suc n) → A
head (x :: xs) = x
```

Listing 3: Function for retrieving the head of a non-empty list.

```
length : ∀ {A n} → Vec A n → ℕ
length {n = n} _ = n
```

Listing 4: Function that returns the length of a vector.

Due to Agda's dependently typed nature, it can be used as a proof assistant for constructing formal proofs. Whenever we write a proof in Agda, we rely on Agda providing a correct judgement. Therefore, it can be stated that this paper relies upon Agda's trustworthiness.

A proof in Agda can be represented as an algorithm. Under the Curry-Howard correspondence, we can construct a *complete* program[1] that matches the signature of a proof. When Agda's type checker accepts the complete program, it essentially means that it accepts the proof as a well-typed construct. If the construct in Agda that represents your proof is incorrect/incomplete, Agda might still accept your proof. In such cases, the 'proof' does not tell us anything meaningful.

3 Intrinsically-typed Haskell-like language

To effectively reason about the correctness of the refactoring operation and demonstrate its implementation, it is important to possess a language construct that serves as the foundation for performing said refactoring.

For the sake of this paper, the syntax that builds the Haskell-like language we seek to refactor is constructed in Agda using intrinsically-typed terms. An intrinsically-typed term can be defined as an expression annotated with its deduced type. Intrinsic typing ensures that the constructed language is well-typed by default.

The constructed language is based upon the simply typed language calculus first introduced by Church [10]. Additionally, we make use of de Bruijn indices to eliminate names from lambda terms and therefore refer to variables as an index in a typed context [11]. This simplifies the process of working with the language as, e.g., renaming operations do not need to be considered. By using de Bruijn indices, we

[1] A complete program must not contain any statements that are yet to be proven.

implicitly state that all identifiers must be unique, an important precondition.

Additionally, a design that closely resembles the notation for typing relations[2] is used. This design makes it easier to reason about the language from a pure theoretical perspective, when needed. The most relevant rules related to the refactoring that this paper is concerned with can be found in figure 1. From the information given above the horizontal line, we can deduce/infer properties of the inferred statement below the line. An explanation of the symbols used in the typing relations is provided below for further clarity.

- Γ: a type-context that stores the types of variables within scope.
- Γ^d: a global declaration context that stores all declared record types.
- τ: a type.

Whenever we write $\Gamma, \Gamma^d \vdash e : \tau$, it can be interpreted as having a term e where its typing judgement, given the environments Γ and Γ^d, evaluates to a type τ.

The typing rules listed in figure 1 can be interpreted as follows;

- (TR1) The typing judgement of a tuple term is a composite of the typing judgements of the expressions that it is composed of.
- (TR2) The typing judgement of a record instance term is a composite of the typing judgements of the expressions that it is composed of. Additionally, the typing judgements should satisfy a known record declaration.
- (TR3) A lookup operation on an expression that reduces to a tuple has a typing judgement that corresponds with the type of the term at the given position.
- (TR4) A lookup operation on an expression that reduces to a record instance has a typing judgement that corresponds with the type of the term at the given position.

The astute reader might notice that the typing relations for the record-related rules (TR2 and TR4) in figure 1 lack the accompanying labels for the record name and its field accessors. For the declaration context, we also make use of de Bruijn indices [11], indicating the lack of a record name. For the field accessors, we argue that using an implicitly generated field accessor can be elaborated[3] into a generic lookup over its contents, making the accessors superfluous at the underlying level. From a parser's point of view, this information is still vital and must be provided.

Listing 5 consists of the translated typing relations from figure 1 to Agda. Observe that there is a strong resemblance between the typing relations and the code. The `TypeResolver` is a construct used to group the types of an arbitrarily sized sequence of intrinsically-typed terms, therefore mimicking the sequences of terms from rules TR1 and TR2 depicted in figure 1.

[2]A typing relation is a collection of inference rules where each rule assigns a type to a term [12].

[3]Expressing/simplifying a language construct in an abstracted manner to reduce overhead at the underlying level.

$$\frac{\Gamma, \Gamma^d \vdash e_1 : \tau_1 \quad \Gamma, \Gamma^d \vdash e_2 : \tau_2 \quad ... \quad \Gamma, \Gamma^d \vdash e_n : \tau_n}{\Gamma, \Gamma^d \vdash \texttt{tuple } (e_1, e_2, ..., e_n) : \texttt{tupleT } (\tau_1, \tau_2, ..., \tau_n)} \text{ TR1}$$

$$\frac{\begin{array}{l} x : \texttt{recordDecl } (\tau_1, \tau_2, ..., \tau_n) \in \Gamma^d \\ \Gamma, \Gamma^d \vdash e_1 : \tau_1 \quad \Gamma, \Gamma^d \vdash e_2 : \tau_2 \quad ... \quad \Gamma, \Gamma^d \vdash e_n : \tau_n \end{array}}{\Gamma, \Gamma^d \vdash \texttt{recInst } x \ (e_1, e_2, ..., e_n) : \texttt{recT } (\tau_1, \tau_2, ..., \tau_n)} \text{ TR2}$$

$$\frac{\begin{array}{l} \Gamma, \Gamma^d \vdash e : \texttt{tuple } (\tau_1, \tau_2, ..., \tau_n) \\ x : \tau_x \in (\tau_1, \tau_2, ..., \tau_n) \end{array}}{\Gamma, \Gamma^d \vdash \texttt{tLookup } e \ x : \tau_x} \text{ TR3}$$

$$\frac{\begin{array}{l} \Gamma, \Gamma^d \vdash e : \texttt{rec } (\tau_1, \tau_2, ..., \tau_n) \\ x : \tau_x \in (\tau_1, \tau_2, ..., \tau_n) \end{array}}{\Gamma, \Gamma^d \vdash \texttt{rLookup } e \ x : \tau_x} \text{ TR4}$$

Figure 1: Typing rules for tuples, record instances, and lookup actions in the Haskell-like language.

4 Big-step semantics

Big-step semantics, introduced under the name *natural semantics* by Kahn [13], is used to describe the relation between input terms and the resulting value [9]. We can use a sequence of inference rules to show this relation, similar to what we did for denoting typing relations in section 3. We use the notation $e \Downarrow v$ to indicate that some term e relates to a value v.

The inference rules used to denote the big-step semantics of the constructs directly related to the refactoring operation are shown in figure 2. The representation in Agda can be found in listing 6 and uses a call-by-value strategy as opposed to Haskell's call-by-need strategy. In addition to the symbols listed in section 3, we introduce supplementary symbols below.

- γ: a value-context where every value $v \in \gamma$ has a corresponding type $t \in \Gamma$.
- v: a value.
- e: an intrinsically-typed term.

An explanation of the big-step semantic rules shown in figure 2 can be found below.

- (BR1) A compound expression of terms related to a particular value that a tuple consists of can be related to a tuple value that groups the values of these terms.
- (BR2) A compound expression of terms related to a particular value that a record instance consists of can be related to a record value that groups the values of these terms. A corresponding record declaration must be referred to when constructing the record instance.
- (BR3) A lookup in an expression that relates to a tuple output should relate to the value at the position of the lookup in the related value.
- (BR4) A lookup in an expression that relates to a record instance output should relate to the value at the position of the lookup in the related value.

```
data TypeResolver ... : List Type → Set where
    []ᵀ : TypeResolver Γ Γᵈ []
    _::_ : (Γ , Γᵈ ⊢ t)
        → TypeResolver Γ Γᵈ ts
        → TypeResolver Γ Γᵈ (t :: ts)

data _,_⊢_ : Ctx → DataCtx → Type → Set where
    ...
    tuple : TypeResolver Γ Γᵈ ts
        ------------------------
        → Γ , Γᵈ ⊢ tupleT ts
    recInst : (recDecl ts) ∈ Γᵈ
        → TypeResolver Γ Γᵈ ts
        ----------------------
        → Γ , Γᵈ ⊢ recT ts
    tLookup : Γ , Γᵈ ⊢ tupleT ts
        → t ∈ ts
        ------------------------
        → Γ , Γᵈ ⊢ t
    rLookup : Γ , Γᵈ ⊢ recT ts
        → t ∈ ts
        ----------------------
        → Γ , Γᵈ ⊢ t
```

Listing 5: Intrinsically-typed terms in Agda for the rules listed in figure 1.

The `ReductionResolver` is a construct similar to the `TypeResolver` (see section 3) that supports arbitrarily sized sequences of term-value relations. It tries to approach the notation of rules BR1 and BR2 from figure 2 as much as possible.

5 Record refactoring

A refactoring can be defined as a transformation of one program into another. As such, we can model a refactoring in Agda as a function that transforms some language construct $\Gamma, \Gamma^d \vdash t$ to a new, refactored, construct $\Gamma', \Gamma^{d\prime} \vdash t'$. We can reduce the operation of refactoring said construct by breaking the transformation into smaller sub-problems[4]. An example of such a sub-problem is to refactor the types only, i.e., construct a function `ref-type : Type → Type` and use that to indicate that `t' = ref-type t`.

An important aspect worth mentioning is that, since the Haskell-like language is *intrinsically-well-typed* (see section 3), the result of the refactoring will be well-typed as well. Thus, performing the refactoring is at the same time a proof that the new language construct is well-typed. A side-effect of this is that the implementation of the transformation becomes more complex (see section 9.1).

Within this paper, we consider record refactoring as a replacement of *all* tuple occurrences by record instances of globally-declared records. Thereby, we treat all tuples as if they were different from each other and argue that if two or more tuples share the same type, they can make use of the same record declaration *after* refactoring. The reason for this is that we cannot 'observe' in what context a tuple is being used. An example of such a case is a program that works with complex numbers and points represented as tuples (see listing 7) where we cannot distinguish what record declaration should be used where without the alias being present.

$$\frac{\gamma, \Gamma^d \vdash e_1 \Downarrow v_1 \quad \gamma, \Gamma^d \vdash e_2 \Downarrow v_2 \quad \ldots \quad \gamma, \Gamma^d \vdash e_n \Downarrow v_n}{\gamma, \Gamma^d \vdash \texttt{tuple}\ (e_1, e_2, \ldots, e_n) \Downarrow \texttt{tuple}\ (v_1, v_2, \ldots, v_n)}\ \text{BR1}$$

$$\frac{x : \texttt{recordDecl}\ (\tau_1, \tau_2, \ldots, \tau_n) \in \Gamma^d \quad \gamma, \Gamma^d \vdash e_1 \Downarrow v_1 \quad \gamma, \Gamma^d \vdash e_2 \Downarrow v_2 \quad \ldots \quad \gamma, \Gamma^d \vdash e_n \Downarrow v_n}{\gamma, \Gamma^d \vdash \texttt{rec}\ x\ (e_1, e_2, \ldots, e_n) \Downarrow \texttt{rec}\ (v_1, v_2, \ldots, v_n)}\ \text{BR2}$$

$$\frac{\gamma, \Gamma^d \vdash e \Downarrow \texttt{tuple}\ (v_1, v_2, \ldots, v_n) \quad x : \tau_x \in (\tau_1, \tau_2, \ldots, \tau_n)}{\Gamma, \Gamma^d \vdash \texttt{tLookup}\ e\ x \Downarrow v_x}\ \text{BR3}$$

$$\frac{\gamma, \Gamma^d \vdash e \Downarrow \texttt{rec}\ (v_1, v_2, \ldots, v_n) \quad x : \tau_x \in (\tau_1, \tau_2, \ldots, \tau_n)}{\Gamma, \Gamma^d \vdash \texttt{rLookup}\ e\ x \Downarrow v_x}\ \text{BR4}$$

Figure 2: Big-step semantics for tuples, record instances, and lookup actions in the Haskell-like language.

5.1 Structure of refactoring operation

A top-level overview of the structure of the proof can be found in listing 8. Observe that the resulting type is composed of all intermediate sub-problems that aid in the refactoring. Completing the proof, which appears to be a regular function, shows that we can map any construct to a new, refactored, construct where *all*[5] tuples have been replaced by record instances.

5.2 Refactoring types and type contexts

Since the refactoring is only concerned with replacing tuples by records, the types of all other constructs remain the same. Since the types of record fields and tuple elements can both be represented by a list of types, mapping a tuple type to a record type is trivial. Therefore, refactoring the types and type contexts can easily be realized by recursively mapping all tuple-type occurrences to record types.

5.3 Extending the declaration context

To exchange a tuple occurrence for a record instance, we must provide a record declaration for said tuple and swap the tuple construct with a record instance referring to this declaration. In case of a tuple lookup, we change the operation to a record lookup that extracts a record element from a refactored expression evaluating to a record instance.

To obtain the new record declarations, we treat an arbitrary expression in the Haskell-like language as a tree and use a postorder traversal where we extract the type signature of a tuple occurrence, convert it to a record declaration, and

[4] This is not always possible when the refactoring operation behaves differently depending on the surrounding language constructs.

[5] See section 6 for a proof that there are no remaining tuples post-refactor.

```
data ReductionResolver ... :
      TypeResolver Γ Γᵈ ts
    → PolyList ts → Set where

    []ᴿ : ReductionResolver γ Γᵈ []ᵀ []
    _::_ : (γ , Γᵈ ⊢ e ⇓ v)
        → ReductionResolver γ Γᵈ tr vs
        → ReductionResolver γ Γᵈ (e :: tr)
                                 (v :: vs)

data _,_⊢_⇓_ : Env Γ → (Γᵈ : DataCtx)
    → (Γ , Γᵈ ⊢ t) → Value → Set where
    ...
    ⇓tuple : ReductionResolver γ Γᵈ tr vs
        ---------------------------------
        → γ , Γᵈ ⊢ tuple tr ⇓ tuple vs
    ⇓recInst : ReductionResolver γ Γᵈ tr vs
        -----------------------------------
        → γ , Γᵈ ⊢ recInst x tr ⇓ rec vs
    ⇓tLookup : γ , Γᵈ ⊢ e ⇓ tuple vs
        ----------------------------
        → γ , Γᵈ ⊢ (tLookup e x)
            ⇓ (poly-list-lookup vs x)
    ⇓rLookup : γ , Γᵈ ⊢ e ⇓ rec vs
        ----------------------------
        → γ , Γᵈ ⊢ (rLookup e x)
            ⇓ (poly-list-lookup vs x)
```

Listing 6: Big-step semantics in Agda for the rules listed in figure 2.

```
type Complex = (Double, Double)
type Point = (Double, Double)

cpAbs :: Complex -> Double
cpAbs (re, im) = sqrt (re ** 2 + im ** 2)

manhDist :: Point -> Point -> Double
manhDist (x1, y1) (x2, y2) = abs (x1 - x2)
                           + abs (y1 - y2)
```

Listing 7: Example of using points and complex numbers represented by tuples.

prepend it to the original declaration context, Γ^d. As such, we end up with a context $\Gamma^{d\prime}$ = ds ++ Γ^d where ds is used to represent the list of new declarations prepended to our initial declaration context.

The most complex part of record refactoring is to provide a lookup proof for the new record declaration ($d \in \Gamma^{d\prime}$) when constructing a record instance. This is trivial for the lookup of a refactored tuple that is the root of an expression since the postorder traversal ensures that the declaration for this tuple is the first element in the new declaration context. Now, consider the example depicted in figure 3. The lookup proof for the record declaration of the refactored tuple on the right branch should point to the second declaration in the refactored declaration context. Thus, we must be aware of the outer context of an expression when generating such a lookup. As the refactoring progresses recursively in a downward movement, it is vital to provide a *precise* trace of the

```
ref : (e : Γ , Γᵈ ⊢ t)
    → ref-ctx Γ
    , ref-d-ctx ((ref-tuples-to-decls e) ++ Γᵈ)
    ⊢ ref-type t
ref e = ref-h e (e-root e)
```

Listing 8: Top-level refactoring method for refactoring tuples into records.

position we are currently in (from a top-level point of view). This trace can then be used to generate the appropriate lookup for the new declaration.

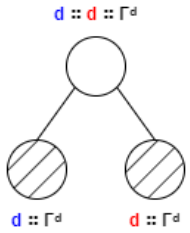

Figure 3: Example of the refactored declaration context at different levels of a language construct. Dashed circles indicate refactored tuples.

5.4 Providing lookback evidence

To provide a trace that allows us to pinpoint the location of the current construct that is recursively being refactored, we define a dependent type, `EmbedInto e e'`, that provides said evidence. As the name hints, `EmbedInto` tells us that an expression `e` is part of a larger construct `e'`. A simplified version of this construct can be found in listing 9. We use the `e-root` constructor to define the top-level construct. Upon moving toward the next step, we provide the appropriate evidence to ensure that the trace is well-defined.

```
data EmbedInto : ... → Set where
    -- Types of all expressions are excluded
    -- from this example
    e-root : (e : Γ , Γᵈ ⊢ t)
        --------------------
        → EmbedInto e e
    e-app-l : EmbedInto (e₂ · e₁) e
        --------------------------
        → EmbedInto e₂ e
    e-app-r : EmbedInto (e₂ · e₁) e
        --------------------------
        → EmbedInto e₁ e
    ...
```

Listing 9: Fragment of the dependent type used to provide evidence that an expression is part of a larger construct.

6 No remaining tuples (post-refactor)

Using Agda, we can prove that an arbitrary refactored expression contains no tuples. We accomplish this by construct-

ing a proof that shows that any refactored expression can be mapped to a construct that does not support tuples as input. We refer to this construct as `HasNoTuples` and provide an example in listing 10. Now, all that remains is to show that an arbitrary refactored expression can be mapped to this construct. This is done by creating a function that maps any expression `e : Γ , Γᵈ ⊢ t` to a `HasNoTuples Γ' Γᵈ' (ref e)`. This appears to be a rather trivial proof as shown in listing 11. Notice that all constructs that were formerly expressed as tuples can use the constructors for record-related constructs (post-refactoring), as they have become records. Therefore, we can conclude that the refactored expression is free of tuples.

```
data HasNoTuples (Γ : Ctx) (Γᵈ : DataCtx) :
    Γ , Γᵈ ⊢ t → Set where
    num : ∀ {n}
        → HasNoTuples Γ Γᵈ (num n)
    fun : {b : t :: Γ , Γᵈ ⊢ u}
        → HasNoTuples Γ Γᵈ (fun b)
    rec : {tr : TypeResolver Γ Γᵈ ts}
          {x : recDecl ts ∈ Γᵈ}
        → HasNoTuples Γ Γᵈ (recInst x tr)
    rlu : {e : Γ , Γᵈ ⊢ recT ts} {x : t ∈ ts}
        → HasNoTuples Γ Γᵈ (rLookup e x)
    ...
```

Listing 10: Fragment of language construct that does not support tuples.

```
proof : (e : Γ , Γᵈ ⊢ t) → HasNoTuples
    (ref-ctx Γ)
    (ref-d-ctx (ref-tuples-to-decls e ++ Γᵈ))
    (ref e)
proof (num n)        = num
proof (fun b)        = fun
proof (tuple tr)     = rec
proof (tLookup e x)  = rlu
proof (recInst x tr) = rec
proof (rLookup e x)  = rlu
...
```

Listing 11: Fragment of proof that refactored expressions do not contain tuples.

7 Refactor value relation

Given an arbitrary expression, we show that we can derive a relation between the value of the expression before and after refactoring. By indexing a value on its type, we observe that this relation coincides with refactoring types.

The general outline of the proof is presented in listing 12. Here, we take the big-step semantics of an arbitrary language construct and the big-step semantics of the refactored counterpart and use that to show that we can construct a relation $v_1 \longrightarrow^V v_2$. The relation $v_1 \longrightarrow^V v_2$ describes that value v_1 of a non-refactored construct relates to v_2 post-refactoring.

Similarly, we define the arrows $\longrightarrow^{PV}$ and $\longrightarrow^{E}$ for sequences of typed values. These sequences expand to a series of value relations $\longrightarrow^V$. Even though both share the same definition, we use $\longrightarrow^{PV}$ for relations between internal sequences of the same size (e.g., tuple and record values) and $\longrightarrow^{E}$ for environments. This makes the expression more expressive.

```
proof : {e : [] , Γᵈ ⊢ t}
    → [] , Γᵈ  ⊢ e     ⇓ v₁
    → [] , Γᵈ' ⊢ ref e ⇓ v₂
    → v₁ ⟶ⱽ v₂
proof {e = e} e₁ e₂ =
    proof-h (e-root e) e₁ e₂ tt
```

Listing 12: Top-level method for constructing the refactor value relation (simplified refactored declaration context).

Listing 13 presents a simplified overview of the value relations. Observe that the relation for constructs like numbers and characters is rather trivial since it enforces us to only prove that their internal representation is identical. For sequences, we concatenate the evidence of value relations. A special case is the relation for closures as they contain an environment of their own. We need to provide additional evidence for these kinds of constructs by supplying the big-step semantics for the bodies.

```
-- Value to value relation
_⟶ⱽ_ : Value t → Value (ref-type t) → Set
num n₁      ⟶ⱽ num n₂      = n₁ ≡ n₂
char c₁     ⟶ⱽ char c₂     = c₁ ≡ c₂
clos b₁ γ₁ ⟶ⱽ clos b₂ γ₂ =
      {vᵀ₁⟶vᵀ₂ : vᵀ₁ ⟶ⱽ vᵀ₂}
    → (vᵀ₁ :: γ₁) , Γᵈ  ⊢ b₁ ⇓ vᵁ₁
    → (vᵀ₂ :: γ₂) , Γᵈ₁ ⊢ b₂ ⇓ vᵁ₂
    → vᵁ₁ ⟶ⱽ vᵁ₂
tuple vs₁  ⟶ⱽ rec vs₂    = vs₁ ⟶ᴾⱽ vs₂
rec vs₁    ⟶ⱽ rec vs₂    = vs₁ ⟶ᴾⱽ vs₂

-- Sequence-to-sequence relation
_⟶ᴾⱽ_ : PolyList ts
    → PolyList (ref-type-list ts) → Set
[]          ⟶ᴾⱽ []          = ⊤
(v₁ :: vs₁) ⟶ᴾⱽ (v₂ :: vs₂) =
    (v₁ ⟶ⱽ v₂) × (vs₁ ⟶ᴾⱽ vs₂)

-- Environment relation
_⟶ᴱ_ : Env Γ → Env (ref-ctx Γ) → Set
_⟶ᴱ_ = _⟶ᴾⱽ_
```

Listing 13: Value relations between non-refactored and refactored constructs that originate from the same expression (simplified closure).

Since the relation is constructed in a step-wise manner and internally relies on the refactoring operation, we again need to provide additional evidence to construct the post-refactor declaration context. To reiterate, the evidence we speak of

tells us that some expression `e` is embedded into a larger expression `e'` (see section 5.4). We take this into account in the helper function for constructing the relation that is shown in listing 14. Observe that whenever we encounter a function, we insert new evidence into the environment relation that can later be used to gather evidence for the resulting value of a lookup expression.

```
proof-h : (ev : EmbedInto e e')
    → γ₁ , Γᵈ  ⊢ e ⇓ v₁
    → γ₂ , Γᵈ' ⊢ ref-h e ev ⇓ v₂
    → γ₁ ⟶ᴱ γ₂
    → v₁ ⟶ⱽ v₂
proof-h ev ⇓num ⇓num γ = refl
proof-h ev ⇓var ⇓var γ = ref-lookup-v γ
proof-h ev ⇓fun ⇓fun γ e₁ e₂ =
    proof-h (e-func ev) e₁ e₂ (vᵀ₁⟶vᵀ₂ , γ)
proof-h ev (⇓tuple rr₁) (⇓recInst rr₂) γ =
    proof-h-rr-tup ev rr₁ rr₂ γ
proof-h ev (⇓tLookup e₁) (⇓rLookup e₂) γ =
    ref-lookup-pl (proof-h (e-tup-l ev) e₁ e₂ γ)
...
```

Listing 14: Helper function used to construct the refactor value relation.

By constructing this relation, we show how the refactoring operation affects the resulting value.

8 Responsible Research

The proofs written in Agda serve as the most important foundation of the work presented in this report. To make this work more transparent to the general public, all proofs are available on a public GitHub repository.[6] Not only does this make the work more transparent, but it also makes it easier to reproduce the research.

The theory that is presented in this work should be sufficient for understanding the code written in the repository and give insight into the bidirectional translation between theory and Agda. For additional information on language design in Agda, one can refer to the Programming Language Foundations in Agda book [9]. In case of Agda itself, we recommend looking at their online wiki [5].

9 Discussion

In this section, we reflect on the contents of this paper by describing in section 9.1 why an intrinsically-typed language is being used. This is followed by section 9.2 where we place our research in a bigger context by comparing it to Haskell (and beyond).

9.1 On using intrinsically-typed terms

Throughout this paper, we worked with an intrinsically-typed language as the foundation for the refactoring operation and proofs. An alternative to using an intrinsically-typed language would be to define both separately (i.e., have a language consisting of terms only and define the typing relations on the side).

[6] https://github.com/JerBast/brp-agda-refactoring-jbastenhof

Both approaches have their own trade-offs. An intrinsically-typed language is well-typed by default and occupies less space than a separate language with an external proof for it being well-typed [9]. Additionally, due to the restriction that we put on ourselves by making it intrinsically-typed, it is easier for Agda to point out mistakes earlier on (e.g., accidentally swapping record declarations with different type signatures). A downside of using an intrinsically-typed language is that, e.g., writing the refactoring becomes harder as we are proving that the refactoring is well-typed at the same time. The main benefit of using a separate language with an external well-typedness proof, in the context of record refactoring, is that we can separate the work. However, this approach comes with the downside of having a higher chance that mistakes surface at a later stage.

We chose to use an intrinsically-typed language, despite it making the refactoring more complex. The main reason for this is that the additional constraints make it easier for Agda to point out potential mistakes sooner.

9.2 Bridge from Haskell-like to Haskell (and beyond)

The refactoring operation and all proofs described in this paper are centered around an intrinsically-typed Haskell-like language. As previously mentioned, the Haskell-like language is merely a simplified subset of Haskell that shares similarities.

To make the Haskell-like language closer to Haskell, we tried to stick to Haskell's top-level definition of constructing records, not its internal representation. All other constructs for defining data objects in Haskell were discarded but left a trace in the global declaration context that, as of now, only consists of record declarations. Record declaration names were omitted because of the use of de Bruijn indices [11]. Additionally, we argued that a parser could elaborate the implicitly generated field accessors to generic lookups. Despite making these simplifications and concessions, we tried to keep the essence of records, namely; a series of global declarations required to construct elements of the declared type. As such, it makes sense to state that this research does not directly translate to proving the correctness of an identical refactoring in Haskell, but instead, can be seen as a source of inspiration or as an indication of its feasibility.

Additionally, there are more functional programming languages out there that support record-like types and tuples (e.g., Erlang, OCaml, Standard ML). As the record representation of the given examples is alike to that of Haskell, we could potentially use this work as a source of inspiration/feasibility indication for these languages as well.

10 Related work

Most related work in the field of proving refactoring correctness is directed towards class-based, object-oriented languages, pull-up/down and renaming transformations being most popular, as opposed to languages that use a functional paradigm [3]. By performing further research in the area of refactoring operations for functional programming languages, we can decrease the knowledge gap between formal

refactoring proofs for class-based, object-oriented languages and functional ones.

As mentioned in section 2.1, a similar refactoring operation for refactoring tuples to records is described by Lövei et al. [4], but a correctness proof remains absent. By adding support for grouping arguments of a function and replacing them with the record-like objects presented in their paper, we can potentially prove similar properties for a language that is more Erlang-*like*.

Horpácsi et al. [14] utilize an approach where they decompose a refactoring operation in smaller refactoring steps for which they can verify correctness more easily. They make use of a nearly complete sub-language of Erlang for constructing the proofs. However, they do not provide additional insights into this sub-language as they consider it to be out-of-scope. Additionally, there is the work of Barwell et al. [15] which uses the dependently typed language Idris [16] to construct a formal definition of the Haskell 98 subset on which they perform a renaming operation and prove structural equivalence.

The language specification of the HLL, on which we base our proofs, matches the general outline of the language specifications presented in work by Wadler et al. [9] and Rouvoet [8]. Both make use of de Bruijn indices [11] to refer to an element of a context by its position. Wadler et al. generally use a more formalized approach by defining special constructs for, e.g., the type contexts. Rouvoet, on the other hand, uses a list to keep the representation simple and allow functions from the standard library[7] to be reused. In this paper, we tend to follow the latter approach to avoid making the language specification unnecessarily more complex.

11 Conclusions and Future Work

This section provides an overview of what we can conclude from the research presented in this paper. Additionally, we suggest what possible improvements/additions can be done in the future.

11.1 Conclusions

Within this paper, we have shown that an intrinsically-typed language can be constructed in Agda for which we can define a refactoring operation from tuples to records. Due to the language being intrinsically-typed, both the language itself as well as the refactoring are well-typed as well. Additionally, we proved that the refactoring successfully converts all tuple occurrences to record instances. Furthermore, we have shown that we can relate the terms in our language to values by constructing big-step semantics and how the refactoring alters the values. By proving these properties we have implicitly shown that proving the correctness of record refactoring is feasible.

Furthermore, we compared our Haskell-like language to Haskell and argued that our work can be used as a foundation or source of inspiration for proving the same properties on a similar refactoring performed on Haskell. The same applies to other functional programming languages that share the notion of records and tuples.

[7] https://github.com/agda/agda-stdlib

11.2 Future Work

The HLL is the foundation of the work presented in this paper. As the HLL is merely a simplified subset of Haskell, it would be interesting to see how we can extend this in the future. By gradually expanding the HLL to something more complex, still related to Haskell, we could potentially find new cases for which we can use similar techniques to those presented in this paper. A starting point would be to allow more complex data types to be constructed. Additionally, it would be interesting to see if we can use a call-by-name, or even call-by-need, approach instead of using a call-by-value evaluation strategy.

We discussed that by adapting the HLL to be more Erlang-*like*, we could potentially prove correctness for the approach described by Lövei et al. [4]. Furthermore, we described that languages with a similar notion of records and tuples (like Standard ML and OCaml) could potentially benefit from the approaches described in this research as well. As such, it would be interesting to see if we could adapt our HLL to these languages and prove similar properties.

Acknowledgements

I would like to thank Luka Miljak and Jesper Cockx for their guidance throughout this project. I thank Luka for providing me with new insights that were of great use while writing this paper. Additionally, I thank both Luka and Jesper for their excellent feedback and for sparking my interest in this area of research.

I would also like to thank the other members of my research project for their support and enthusiasm. In particular, I would like to thank José Carlos Padilla Cancio for inspiring me with his value relation approach. Without this approach, I would not have been able to finish my last proof.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] D. Horpácsi, J. Kőszegi, and S. Thompson, "Towards Trustworthy Refactoring in Erlang," *Electronic Proceedings in Theoretical Computer Science*, vol. 216, pp. 83–103, 7 2016.

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

[4] L. Lövei, Z. Horváth, T. Kozsik, and R. Király, "Introducing records by refactoring," *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, 2007.

[5] Agda Development Team, "Agda 2.6.3 documentation." https://agda.readthedocs.io/en/v2.6.3/, 2023.

[6] L. Miran, *Learn you a Haskell for great good!: A beginner's guide*, ch. Making Our Own Types and Type Classes. No Starch Press, 2012.

[7] U. Norell, *Dependently Typed Programming in Agda*, pp. 230–266. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[8] A. J. Rouvoet, "Correct by Construction Language Implementations," 2021.

[9] P. Wadler, W. Kokke, and J. G. Siek, "Programming language foundations in Agda." https://plfa.inf.ed.ac.uk/22.08/, Aug. 2022.

[10] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, no. 2, p. 56–68, 1940.

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

[12] B. C. Pierce, *Types and programming languages*. MIT Press, 2002.

[13] G. Kahn, "Natural semantics," in *STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science Passau, Federal Republic of Germany, February 19–21, 1987 Proceedings 4*, pp. 22–39, Springer, 1987.

[14] D. Horpácsi, J. Kőszegi, and Z. Horváth, "Trustworthy refactoring via decomposition and schemes: A complex case study," *Electronic Proceedings in Theoretical Computer Science*, vol. 253, p. 92–108, 2017.

[15] A. D. Barwell, C. M. Brown, and S. Sarkar, "Proving renaming for Haskell via dependent types : a case-study in refactoring soundness," in *8th International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2021)*, 2021.

[16] E. Brady, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, pp. 552–593, 9 2013.