GADTless Programming in Haskell 98

Martin Sulzmann¹ and Meng Wang²

¹ School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

² Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
meng.wang@comlab.ox.ac.uk

Abstract. Generalized algebraic data types (GADTs) allow to write sophisticated, type-safe programs and transformations. But not many languages, respectively their underlying implementations, support GADTs. We show that Pottier’s and Gauthier’s polymorphic typed defunctionalization, which was supposed to rely on GADTs, can actually be represented in Haskell 98. Our results help to get a better understanding of GADTs and we identify sufficient conditions under which we can replace GADTs with features available in standardized languages such as Haskell 98.

1 Introduction

Generalized algebraic data types (GADTs) are an extension of (boxed) existential types [12]. In contrast to algebraic data types we may refine the type of a GADT depending on the particular constructor. The power behind GADTs is that we can make use of these type refinements when pattern matching over a GADT.

There is a real “industry” that exploits the power of GADTs in sophisticated ways to write more expressive programs and transformations. We refer to [14, 24, 19, 20] for a selection of examples. This trend is supported by an increasing number of functional and object-oriented language implementations [18, 9] which support GADTs and thus allow more users to make use of GADTs in their programs. However, GADTs still have to be adopted by standard language definitions [13, 17] (respectively, whether GADTs will be part of a new language standard is still under active discussion [8]). Hence, it is important to identify conditions under which powerful features such as GADTs can be replaced by simpler features while not substantially changing the programs and their meaning. We refer to this as GADTless programming.

In this paper, we consider GADTless programming in Haskell 98 [17]. Haskell 98 represents a minimal, portable version of the Haskell language including an accompanying standard library. We show that some non-trivial GADT-based transformations and programming examples can actually be expressed in “pure”
Haskell 98. As we will see later, we need one extension, rank-2 types, which is not part of Haskell 98 but supported by common implementations.

Specifically, we make the following contributions:

- We review the polymorphic defunctionalization transformation by Pottier and Gauthier to a target language with support for GADTs. We show that the GADTs arising in the defunctionalization of polymorphic programs can be represented in terms of a form of type coercion expressible via Haskell 98’s newtype construct (Section 5.2). Thus, polymorphic typed defunctionalization can be supported by type-preserving compilers which support Haskell 98 style newtype but not necessarily GADTs.

- Other uses of GADTs require an alternative encoding approach where we use explicit coercion functions. For the first time, we identify sufficient conditions under which an encoding of GADTs with explicit coercion functions is possible (Section 6).

In some cases, GADTs can also be encoded by type classes which are part of Haskell 98. To be precise, Haskell 98 supports single-parameter and constructor classes. In our opinion, GADTless programming using type classes is more complicated. We defer a detailed discussion to Section 7.

We continue in Section 2 where we introduce some basic notation used throughout the paper. Section 3 reviews the key concepts behind GADTs. In source programs, we use GADTs as supported by GHC. In Section 4, we discuss the key ideas behind the encodings of GADTs via newtype and explicit coercion functions. Section 7 concludes and also discusses related work. The Appendix contains additional material such as proofs of the results stated.

2 Preliminaries

We write $\bar{o}$ to denote a sequence of objects $o_1, \ldots, o_n$. We write $fv(o)$ to denote the set of free variables in some object $o$.

We assume that the reader is familiar with the concepts of substitution, unifiers, most general unifiers (m.g.u.) etc [11]. For example, $[t/a]$ denotes the substitution which has the effect of replacing each occurrence of variable $a$ by term $t$. Often, we abbreviate $[t_1/a_1, \ldots, t_n/a_n]$ by $[t/a]$.

We make use of constraints $C$ consisting of conjunction of primitive constraints such as $t_1 = t_2$ describing equality among types $t_1$ and $t_2$. The set of types is described below. We also assume basic familiarity with first-order logic. We write $\models$ to denote the model-theoretic entailment relation. We refer to [25] for details.

Types are stratified into simple types which consist of variables $a$, functions $t_1 \rightarrow t_2$ and user-definable types $T t$ (for example lists $[t]$, pairs $(t_1, t_2)$ etc). Type schemes can either be simple types $t$ or of the form $\forall a. C \Rightarrow t$. We commonly write $\sigma$ to refer to type schemes. We write $\forall a. t$ as a short-hand for $\forall a. b = b \Rightarrow t$.

We assume a functional expression language consisting of variables $x$, function application $e_1 e_2$, function abstraction $\lambda x.e$ and let statements $\text{let } f =
e₁ in e₂ which can be optionally annotated let f :: ∀a.C ⇒ t; f = e₁ in e₂ where
a = fs(C, t). In source programs, we commonly omit the quantifier ∀a. We also
assume case statements case e of [p₁ → e₁]∈I where patterns p are of the form x or
K p. K is the constructor belonging to a user-definable type T a. We assume that
the types of constructors and primitive functions are recorded in some initial type
environment Γinit. Type environments Γ are of the form \{x₁ : σ₁, ..., xₙ : σₙ\}. We assume that fs(Γ) = \{σ₁, ..., σₙ\}.

3 Generalized Algebraic Data Types

The power of GADTs comes through (local) type refinement which effectively
corresponds to the presence of (local) type equalities t₁ = t₂. To illustrate the
workings of GADTs, we take look at a classic example. The following defines a
GADT Exp a to represent a simple expression language and a strongly-typed
evaluator eval for this language. We will make use of GHC Haskell style syntax
[6] in example programs throughout the paper.

data Exp :: * -> * where
    Zero :: Exp Int
    Succ :: Exp Int -> Exp Int
    Pair :: Exp b -> Exp c -> Exp (b, c)

eval :: Exp a -> a
eval = \ x -> case x of
    Zero -> 0
    (Succ e) -> (eval e) + 1
    (Pair x y) -> (eval x, eval y)

At first look it may be surprising that eval has type ∀a. Exp a → a. For
example, consider the first clause where we return the value 0 of type Int which
clearly does not match the expected result type a. However, the program is
well-typed because of the type refinement enabled by the pattern match over
Zero.

Here is a more detailed explanation. Variable x has type Exp a. For the
program to be well-typed, we must verify that the pattern clause Zero→0 has
type Exp a → a. When pattern matching over constructor Zero of type Exp Int
we refine type a to Int within the body of the pattern clause. We can represent
type refinement with either substitutions or type equalities. We choose the latter.
Hence, we must check that 0 has type a under the (local) type equality Int = a.
Intuitively, it should be clear that this check is successful.

Let us take a look at the formal underpinnings of GADTs. The GADT type
system is based on a variant of the familiar Hindley/Milner type system. Because
we choose to use type equalities to represent type refinement we use a variant
of the HM(X) type system framework [15] where the constraint domain X is
the Herbrand constraint domain. The core GADT system can be summarized in
the following four typing rules. All other typing rules are the familiar ones from
HM(X) and are omitted for brevity.

\[ C, \Gamma \vdash_{GADT} e : t_1 \]
\[ C, \Gamma \vdash_{GADT} e : t_2 \]

We assume that constraints \( C \) and \( D \) contain type equalities. Via rule (Eq) we can change the type of an expression to \( t_2 \) if the given constraint \( C \) implies that \( t_1 \) and \( t_2 \) are equal, written \( C \models t_1 = t_2 \). Applied to our above example, we can deduce that \( \text{Int} = a, \{0 : \text{Int}\} \vdash_{GADT} 0 : a \).

In rule (Pat), we type check pattern clauses. Out of the pattern \( p \) we extract the constraints arising and compute the pattern binding which assigns types to pattern variables. For simplicity, we assume that constructors \( K \) take a single argument only. See rule (PK). We ignore the universal quantifier \( \forall b \) for the moment. In our representation, we assume that the types of constructors are normalized such that their result type matches the type of the GADT. Thus, we find that \( \text{Zero} : \forall a.\text{Int} = a \Rightarrow \text{Exp} \ a \). It is straightforward to convert source GADT definitions such as \( \text{data Exp :: * -> * where Zero :: Exp Int} \) to the external representation. The main point to note is that the constraint \( D \) arising out of \( p \) is used in combination with the given constraint \( C \) to type check the body \( e \) of the pattern clause. Also note that constraint \( D \) is only used for type checking \( e \) and does not affect other program parts.

The universal quantifier \( \forall b \) comes into play in case of (boxed) existential types. For example, consider

\( \text{data Exp :: * -> * where Pair :: Exp a -> Exp b -> Exp (a,b)} \)

which is written \( \text{Pair} : \forall a, b, c.(b, c) = a \Rightarrow \text{Exp} \ b \rightarrow \text{Exp} \ c \rightarrow \text{Exp} \ a \) in the internal system. Notice that variables \( b \) and \( c \) do not appear in the result type \( \text{Exp} \ a \). We refer to them as “existential” or “abstract” variables. In programs we are not allowed to make any specific assumptions about them. Hence, logically these variables must be considered as universally quantified. We take care of this in the type system by checking that \( b \cap \text{fs}(C, \Gamma, t_2) = \emptyset \) in rule (Pat).

The challenge for the typed intermediate languages of todays modern compiler systems is to have a sufficiently rich target language which can host GADTs. A naive translation of GADTs to “plain” System F will result in a loss of type-preservation. The reason is that via rule (Eq) (type refinement) we can change the type of an expression without changing the expression itself. There exist sophisticated extensions of System F, for example consider [5, 26, 23], which support a “cast” operator to translate type refinement. For example, in [26] the expression \( e \) is translated to \( \gamma \mathbin{\Rightarrow} e \) where \( \mathbin{\Rightarrow} \cdot \cdot \cdot \) represents a new language.
construct which takes a witness $\gamma$ for $t_1 = t_2$ and an expression $e$ of type $t_1$ and yields an expression of type $t_2$. But not every language and its underlying implementation support such cast operations.

4 GADT Encodings via Newtypes and Explicit Coerctions

To encode GADTs in a conventional system such as Haskell 98 we need to model type equalities $t_1 = t_2$ and their effect. The idea is to replace each type refinement step via safe coercion functions. Each type refinement step $e : t_1 \rightsquigarrow e : t_2$ is then turned into the function application $\gamma e$ where $\gamma$ is the coercion representing the “directed” equality $t_1 = t_2$ (from left to right). In essence, the encodings apply the proofs-are-programs principle where the coercion $\gamma$ represents a proof (term) for $t_1 = t_2$. Of course, to ensure correctness of this encoding scheme, we need to guarantee that at run-time each coercion $\gamma$ evaluates to the identity by construction.

There are two approaches known in the literature to encode such coercion functions. One approach, employed in [1,3,16,28], uses “Leibniz” equality

```
newtype a :=: b = Proof { coerce :: forall f . f a -> f b }
refl :: a :=: a
refl = Proof id
newtype Flip f a b = Flip { unFlip :: f b a }
 symm :: a :=: b --> b :=: a
 symm p = unFlip (coerce p (Flip refl))
 trans :: a :=: b --> b :=: c --> a :=: c
 trans p q = Proof (coerce q . coerce p)
``` 

In the above, we employ newtype declarations as supported by Haskell 98 extended with rank-2 type (notice the forall f). The purpose of the rank-2 type is to ensure that the identity function is the only inhabitant of a :=: b. We ignore here the two “exceptional” cases $\perp$ and $\\backslash x --> \perp$.

Types introduced via the newtype declaration are new, distinct types isomorphic to existing ones. We will refer to such types as newtypes from now on. They are similar to (labeled) data type declaration such as

```
data a :=: b = Proof { coerce :: forall f . f a --> f b }
``` 

The difference is that newtypes cause no run-time type overhead. Constructor Proof coerces a value from type a :=: b to type forall f . f a --> f b. The important point is that these coercions can be implemented without execution time overhead; newtype does not change the underlying representation of an object [17].
The up-shot of the above is that the new type \(a :=: b\) encodes Leibniz’ law which states that if \(a\) and \(b\) are equivalent then we may substitute one for the other in any context. In the above, we provide a few sample type coercion functions representing the expected equality laws: reflexivity, symmetry and transitivity. Functions \texttt{to} and \texttt{from} represent directed coercions. Thus, we can encode the GADT program from Section 3 in Haskell 98 as follows.

```haskell
data Exp' a where  
  Zero' :: a :=: Int -> Exp' a  
  Succ' :: a :=: Int -> Exp' a -> Exp' a  
  Pair' :: a :=: (b,c) -> Exp' b -> Exp' c -> Exp' a

eval' :: Exp' a -> a  
eval' = \x -> case x of  
  (Zero' p) = (from p) 0  
  (Succ' p e) = (from p) (((to p) (eval' e)) + 1)  
  (Pair' p x y) = (from p) ((eval' x, eval' y))
```

Each GADT equation \(t_1 = t_2\) appearing in a constructor is turned into an additional argument \(t_1 :=: t_2\). For example, \(\text{Zero} : \forall a.\text{Int} = a \Rightarrow \text{Exp} a\) is turned into \(\text{Zero}' : \forall a.\text{Int} :=: a \rightarrow \text{Exp}' a\). We find that \(\text{Exp}' a\) is a Haskell 98 data type (though we use here GHC syntax for writing data type declarations). In the body of each pattern clause, we replace each type refinement step (Eq) via an application of the directed coercion functions \texttt{to} or \texttt{from}.

The alternative to the newtypes method is to represent type equality via some explicit coercion functions [2].

```haskell
type EQ a b = (a->b,b->a)  
refl :: EQ a a  
refl = (id,id)  
sym :: EQ a b -> EQ b a  
sym (f,g) = (g,f)  
trans :: EQ a b -> EQ b c -> EQ a c  
trans (f1,g1) (f2,g2) = (f2.f1,g1.g2)  
list :: EQ a b -> EQ [a] [b]  
list (f,g) = (map f, map g)
```

The gist of the GADT encoding method remains the same. In contrast to the newtypes approach, the explicit coercion approach incurs a run-time penalty. For example, consider the compositional law for lists where we actually have to traverse the entire list to coerce each element into its proper type. We also need to guarantee that each coercion evaluates to the identity at run-time. This is enforced by the type system in case of the newtypes approach.

As we will shortly show in the upcoming Section 5.2, the GADTs which arise in Pottier’s and Gauthier’s polymorphic typed defunctionalization transformation can be encoded via newtypes. For many other GADT programs we seem to require explicit coercions in the encoding. The details are in Section 6.
5 Defunctionalization

Defunctionalization [21, 22] is a global program transformation to turn higher-order programs into first-order order ones. The basic idea is to replace every lambda abstraction with its own data constructor that will carry what environment is needed, and replace every application of the higher-order function with an apply function that will interpret the data structure.

For example, defunctionalization of the higher-order program

\[
\begin{align*}
\text{empty} & = \lambda x_1 \rightarrow \text{False} \\
\text{insert} & = \lambda x_2 \rightarrow \lambda x_3 \rightarrow \lambda x_4 \rightarrow (x_2 == x_4) \mid \mid (x_3 x_4) \\
\text{res} & = \text{insert 1 empty 2}
\end{align*}
\]

yields the following first order program

\[
\begin{align*}
\text{apply} & = \lambda f \rightarrow \lambda \text{arg} \rightarrow \\
& \quad \text{case f of} \\
& \quad \quad x_1 \rightarrow \text{False} \\
& \quad \quad x_2 \rightarrow \text{let } x_2 = \text{arg in } x_3 x_2 \\
& \quad \quad (x_3 x_2) \rightarrow \text{let } x_3 = \text{arg in } x_4 x_2 x_3 \\
& \quad \quad (x_4 x_2 x_3) \rightarrow \text{let } x_4 = \text{arg in } (x_2 == x_4) \mid \mid (\text{apply } x_3 x_4)
\end{align*}
\]

\[
\begin{align*}
\text{empty} & = x_1 \\
\text{insert} & = x_2 \\
\text{res} & = \text{apply (apply (apply (insert 1)) empty) 2}
\end{align*}
\]

We assume that primitives == and || are not affected by defunctionalization. Function closure construction is encoded as injection (construction of a sum of type Arrow a b) whereas function application is encoded as case analysis (sum elimination). We represent Arrow a b via a data type where for each lambda abstraction we introduce the following constructors. Free variables are passed in as arguments to constructors.

\[
\begin{align*}
x_1 : \forall a. \text{Arrow a Bool} \\
x_2 : \forall a. \text{Arrow (Arrow a Bool) (Arrow a Bool)} \\
x_3 : \forall a. a \rightarrow \text{Arrow (Arrow a Bool) (Arrow a Bool)} \\
x_4 : \forall a. a \rightarrow \text{Arrow a Bool} \rightarrow \text{Arrow a Bool}
\end{align*}
\]

We have achieved a defunctionalization of programs, but the problem is that the transformation is not type-preserving (assuming we use Hindley/Milner or System F as our target language). The first problem is that the above constructors do not share the same (result) type. Hence, we cannot represent them via an algebraic data type. The second problem is that function apply is not typable in Hindley/Milner or System F. For example, the first clause of apply has return type Bool whereas the second clause returns a value of different type Arrow (Arrow a Bool) (Arrow a Bool).

A possible work-around is to specialize function apply with each monomorphic type; in the case of (parametric) polymorphism, monomorphization is needed prior to defunctionalization. However, this leads to code duplication and is even
impossible in case the source language supports polymorphic recursion (which
is the case for Haskell 98).

Fortunately, Pottier and Gauthier [19] came up with an ingenious idea to
solve this problem. They show how to define a type-preserving defunctionaliza-
tion of polymorphic programs if the target language is enriched with GADTs. We
rewiew the gist of their method in Section 5.1. A closer inspection of their method
reveals that the GADTs arising in the defunctionalization of polymorphic pro-
grams can be replaced via newtypes. We formalize this insight in Section 5.2.

5.1 Polymorphic Typed Defunctionalization via GADTs

We apply Pottier’s and Gauthier’s type-preserving defunctionalization transfor-
mation to our running example. Their key idea is to use a GADT (instead of an
algebraic data type) to represent the sum type Arrow.

data Arrow :: * -> * -> * where
  X1 :: Arrow a1 Bool
  X2 :: Arrow a1 (Arrow (Arrow a1 Bool) (Arrow a1 Bool))
  X3 :: a -> Arrow (Arrow a Bool) (Arrow a Bool)
  X4 :: a1 -> Arrow a1 Bool -> Arrow a1 Bool

In the GADT type system, function apply is well-typed.

apply :: Arrow a1 a2 -> a1 -> a2
apply = \ f -> \ arg ->
  case f of
    X1 -> False
    X2 -> let x = arg in X3 x2
    (X3 x2) -> let x3 = arg in X4 x2 x3
    (X4 x2 x3) -> let x4=arg in (x2==x4) || (apply x3 x4)

For example, consider the last case. Pattern matching over (X4 x2 x3) yields a
binding \{x2 : a1; x3 : Arrow a1 Bool\} and (local) constraint Bool = a2. The let
statement yields the binding \{x4 : a1\}. We straightforwardly find that (x2==x4)
|| (apply x3 x4) has type Bool. From Bool = a2 and via application of rule
(Eq) we can conclude that this program text has also type a2. Hence, the pattern
clause agrees with the type annotation. Similarly, we can verify type correctness
of the first three cases.

The upshot of Pottier’s and Gauthier’s method is that defunctionalization
can be added to the toolbox of type-preserving compiler writers if the target
language supports GADTs. Our insight is that GADTs are not necessary in
Pottier’s and Gauthier’s defunctionalization method. We can always replace the
GADTs arising in the defunctionalization via newtypes while retaining type
preservation. That is,

First, we replace the Arrow GADT by
data Arrow': : * -> * -> * where
X1' :: a2 := : Arrow' a1 a2
X2' :: a2 := : (Arrow' (Arrow' a1 Bool) (Arrow' a1 Bool)) -> Arrow' a1 a2
X3' :: a1 := : Arrow' a1 Bool -> a2 := : Arrow' a1 a2
X4' :: a2 := : Bool -> a1 -> Arrow' a1 Bool -> Arrow' a1 a2

Each GADT constructor such as X1 : ∀a1, a2. a2 = Bool ⇒ Arrow' a1 a2 is turned into X1' : ∀a1, a2. a2 := : Bool → Arrow' a1 a2. We find that Arrow' is a Haskell 98 data type (though we use here GHC syntax for writing data type declarations).

The challenge is to replace apply by an equivalent definition which is typable in Haskell 98. The key observation is that uses of the GADT rule (Eq) for typing apply can be omitted by inserting newtypes coercions to or from.

apply' :: Arrow' a1 a2 -> a1 -> a2
apply' = \ f -> \ arg ->
case f of
  (X1' ep2) -> (to ep2) False
  (X2' ep2) -> let x2 = arg in (to ep2) (X3' x2 refl refl)
  (X3' ep1 ep2 x2) -> let x3 = arg in (to ep2) (X4' x2 ((from ep1) x3) refl1)
  (X4' ep2 x2 x3) -> let x4 = arg in (to ep2) ((x2=x4) || (apply' x3 x4))

empty = X1' refl
insert = X2' refl
res = apply' (apply' (apply' insert 1)) empty 2

For example, in the last clause application of the coercion (to ep1) has the same effect as application of the typing rule (Eq) under constraint Bool = a2. Thus, we achieve a type-preserving defunctionalization via newtypes for our running example. The above observation can be generalized for all GADTs which arise in Pottier’s and Gauthier’s defunctionalization method. The details are in the next section.

5.2 Polymorphic Typed Defunctionalization via Newtypes

Our scheme follows the approach in [19]. Defunctionalization is defined in terms of judgments \( \Gamma \vdash^F e : t \leadsto e' \) where we assume that source expression e has type t under environment \( \Gamma \), and the target expression e' is the defunctionalized version of e. We write \([t]\) to denote the defunctionalized version of type t (which naturally extends to type environment \([\Gamma]\)):

\[
[a] = a \quad \quad \quad [t_1 \rightarrow t_2] = \text{Arrow'} \ [t_1] \ [t_2] \quad \quad \quad [T \ [t]] = T \ [t]
\]

The crucial difference to [19] is that we only rely on Haskell 98 features in the target language. For each lambda abstraction \( \lambda x . e : t_1 \rightarrow t_2 \), where \( \bar{y} = fn(\lambda x . e) \) and each \( y_i \) has type \( t_i \) in the given type environment, we introduce a Haskell 98 data constructor \( \chi' : \forall a_1, a_2, f(t_1, t_2, \bar{f}) \), \( a_1 := : [t_1] \rightarrow a_2 := : [t_2] \rightarrow [\bar{f}] \rightarrow \text{Arrow'} a_1 a_2 \) to the initial type environment. We assume that \( a_1 \) and \( a_2 \) are fresh variables. For example, for \( \lambda x_1 \rightarrow \text{False} \) we introduce \( \chi' : \forall a_1, a_2, b.a_1 := : 

\]

9
\[
\begin{align*}
\text{(Var)} & \quad (x : \forall a.t) \in \Gamma \\
\Gamma & \vdash_{DF} x : [\bar{t} / \bar{a}]t \Rightarrow x
\end{align*}
\]
\[
\begin{align*}
\text{(App)} & \quad \Gamma \vdash_{DF} e_1 : t_2 \rightarrow t \Rightarrow e'_1 \quad \Gamma \vdash_{DF} e_2 : t_2 \Rightarrow e'_2 \\
\Gamma & \vdash_{DF} e_1 \ e_2 : t \Rightarrow apply' \ e'_1 \ e'_2
\end{align*}
\]
\[
\begin{align*}
\tilde{y} & = \text{fe}((\lambda x.e) \ \bar{y} = y_1, \ldots, y_n \quad \Gamma \vdash_{HM} y_i : t_i \quad \text{for } i = 1, \ldots, n \\
\Gamma & \vdash_{HM} X' : (a_1 := \{t_1\}) \rightarrow (a_2 := \{t_2\}) \rightarrow \text{Arrow} \ a_1 \ a_2 \\
\Gamma \cup \{x : a_1, ep_1 : a_1 := t_1, ep_2 : a_2 := t_2\} & \vdash_{DF} e : t_2 \Rightarrow e'
\end{align*}
\]
\[
\begin{align*}
\Gamma & \vdash_{DF} \lambda x.e : t_1 \rightarrow t_2 \Rightarrow \text{refl} \ \text{refl} \ \bar{y}
\end{align*}
\]
\[
\begin{align*}
\Gamma & \vdash_{DF} e : t_1 \Rightarrow e' \quad \Gamma \vdash_{DF} e : t_2 \Rightarrow e' \\
\Gamma & \vdash_{DF} e : t_2 \Rightarrow \text{from} \ \bar{p} \ e' \\
\Gamma & \vdash_{DF} e : t_1 \Rightarrow \text{to} \ \bar{p} \ e'
\end{align*}
\]

**Fig. 1. GADTless Defunctionalization**

\[b \rightarrow a_2 := \text{Bool} \rightarrow \text{Arrow}' \ b \ a_2\] which could be simplified to \(X_1' : \forall a_1, a_2, a_2 := \text{Bool} \rightarrow \text{Arrow}' \ a_1 \ a_2\) as we have done in the previous section.

Let us take a closer look at the defunctionalization rules in Figure 1. Each lambda-abstraction is injected into the datatype \(\text{Arrow}'\) with the corresponding data constructor. See rule (Abs). Proof witnesses encapsulating the identity function \((\text{refl})\) and free term variables \((\bar{y})\) are passed in as arguments. We write \(\Gamma \vdash_{HM} y_i : t_i\) to denote well-typing in Hindley/Milner extended with newtypes. The defunctionalization of function body \(e\) is done under an environment extended with two proof witnesses named as \(ep_1\) and \(ep_2\): one for the input type and one for the output type. Note that we bind the lambda variable \(x\) with type \(a_1\) instead of \(t_1\) in the environment under which \(e\) is typed. Rule (Eq1) is used to coerce the type of \(x\) from \(a_1\) to \(t_1\). As a result, every occurrence of \(x\) in \(e\) is expected to be substituted with \((\text{from} \ \bar{p}) \ x\) in \(e'\). The defunctionalization result \(e'\) does not appear in rule (Abs)’s conclusion; instead, it is used in constructing the special function \(\text{apply}'\) during a post-translation step (see below). This reflects the fact that in a defunctionalized program, actual function bodies are all collected by the definition of \(\text{apply}'\), and each closure resides at the invocation site only encapsulates an unique identifier and a value environment. For each function application, the closure is passed to function \(\text{apply}'\) for proper dispatching. See rule (App).

The purpose of rules (Eq1) and (Eq2) is to mimic the GADT typing rule (Eq) by inserting appropriate directed type coercions. Recall that the difficulty of defunctionalization lies in defining a well-typed function \(\text{apply}\). In the GADT-based defunctionalization setting, given \(\text{apply}'\)'s type \(\text{Arrow} \ a_1 \ a_2 \rightarrow a_1 \rightarrow a_2\), each defunctionalized function \(\lambda x.e'\) of type \([t_1] \rightarrow [t_2]\) must be made to match \(\text{apply}'\)'s signature. In other words, a function of type \([t_1] \rightarrow [t_2]\) must accept an input of type \(a_1\) and returns an output of type \(a_2\). Consider the third case of our running example, the function \(\lambda x_3 \rightarrow x_4 \ x_2 \ x_3\) is of type
Arrow a Bool \rightarrow Arrow a Bool\ which\ does\ not\ match\ function\ apply’s\ signature\ \(a_1 \rightarrow a_2\).\ This\ is\ the\ very\ and\ only\ place\ (i.e.\ the\ definition\ of\ apply)\ where\ we\ deploy\ rule\ (Eq)\ to\ achieve\ type\ preservation\ in\ the\ GADT-based\ defunctionalization\ scheme.\ By\ construction,\ pattern\ \(x\)\ for\ function\ \(\lambda\ x \rightarrow e’\)\ always\ carries\ the\ assertion\ \(a_1 = [t_1]\)\ and\ \(a_2 = [t_2]\),\ which\ allows\ rule\ (Eq)\ to\ coerce\ from\ \(a_1\)\ to\ \([t_1]\)\ and\ from\ \([t_2]\)\ to\ \(a_2\)\ when\ \(e’\)\ is\ typed.\ In\ the\ case\ of\ \(\lambda\ x3 \rightarrow X4\ x2\ x3\),\ pattern\ matching\ \(X3\)\ yields\ the\ constraints\ \(a_1 = Arrow\ a\ Bool\)\ and\ \(a_2 = Arrow\ a\ Bool\),\ which\ justifies\ the\ type\ \(a_1 \rightarrow a_2\).\ Note\ that\ in\ defunctionalization,\ uses\ of\ local\ type\ equality\ constraints\ are\ always\ “straightforward”:\ coercions\ are\ only\ between\ types\ where\ a\ equality\ constraint\ of\ the\ two\ syntactically\ appears\ in\ the\ store.\ Thus,\ the\ logic\ implication\ (\(\Rightarrow\))\ used\ in\ rule\ (Eq)\ can\ be\ simplified\ as\ set\ inclusion\ (\(\subseteq\)).\ Applied\ to\ our\ Haskell\ 98-based\ type-directed\ translation,\ this\ property\ guarantees\ that\ either\ rule\ (Eq1)\ or\ (Eq2)\ can\ be\ used\ in\ place\ of\ (Eq)\ when\ needed,\ as\ proof\ witnesses\ of\ correct\ types\ should\ be\ directly\ available\ from\ the\ environment.

The\ last\ step\ of\ defunctionalization\ is\ a\ post-translation\ process\ which\ completes\ the\ definition\ of\ function\ apply’.\ For\ each\ lambda\ abstraction\ in\ the\ source\ program\ whose\ defunctionalization\ derivation\ ends\ with

\[
\begin{align*}
\mathcal{G} & \vdash t e (\lambda x. e) \\
\mathcal{G} & \vdash \forall (x : t_1) \quad \mathcal{G} \vdash t_i \\
\mathcal{G} & \vdash \mathcal{H} \quad y_i : t_i & & \text{for } i = 1, \ldots, n \\
\mathcal{G} & \vdash \mathcal{D} \quad X’ : (a_1 := [t_1]) \rightarrow (a_2 := [t_2]) \rightarrow [t] \rightarrow Arrow a_1 a_2 \\
\mathcal{C}, \mathcal{G} \cup \{ x : a_1, ep_1 : a_1 := t_1, ep_2 : a_2 := t_2 \} & \vdash \mathcal{D} \quad e : t_2 \rightsquigarrow e’
\end{align*}
\]

we\ generate\ a\ pattern\ clause\ patcl\ of\ the\ form

\[
\begin{align*}
X1 & \cdot ep1 \cdot ep2 \cdot \bar{y} \Rightarrow \text{let } x = \text{arg in } (\text{to ep2}) e’ \\
\text{We\ collect\ all\ such\ clauses\ to\ build}\ & \\
\text{apply} & : : \text{Arrow a1 a2 } \rightarrow \text{a1 } \rightarrow \text{a2} \\
\text{apply} & = \lambda f \rightarrow \lambda \text{arg } \rightarrow \\
\text{case } f \text{ of } & \text{patcl}
\end{align*}
\]

It\ follows\ straightforwardly\ that\ defunctionalized\ programs\ produced\ by\ our\ scheme\ are\ operationally\ equivalent\ to\ the\ target\ program\ generated\ in\ [19].\ The\ only\ difference\ is\ the\ additional\ occurrence\ of\ coercions\ to\ refl\ or\ from\ refl\ which\ evaluate\ to\ the\ identity.\ Recall\ that\ coercions\ among\ new,\ isomorphic\ types\ can\ be\ implemented\ without\ execution\ time\ overhead\ because\ newtype\ does\ not\ change\ the\ underlying\ representation\ of\ an\ object\ [17].

The\ important\ result\ is\ that\ defunctionalized\ programs\ are\ well-typed\ Haskell\ 98\ programs.\ Recall\ that\ \(\vdash^{HM}\)\ refers\ to\ the\ Hindley/Milner\ system\ extended\ with\ newtypes.

**Theorem 1 (Well-Typed).**\ Let\ \(\mathcal{G} \vdash^{HM} e : t\)\ and\ \(e\)\ is\ defunctionalized\ to\ \(e’\).\ Then\ \([\mathcal{G}] \vdash^{HM} e’ : [t]\).

We\ can\ also\ state\ completeness.\ That\ is,\ all\ well-typed\ programs\ can\ be\ defunctionalized.
Theorem 2 (GADTless Defunctionalization Completeness). Let $\Gamma \vdash_{HM}^t e : t$. Then $\Gamma \vdash^{DF} e : t \rightsquigarrow e'$ for some $e'$.

Proofs of the above results are given in Appendix B.

6 Further GADTless Programming Examples

The GADT examples we have seen so far could straightforwardly be encoded via newtypes. In this section, we observe that for many other GADT examples the explicit coercion encoding approach appears to be more suitable.

Let’s attempt an encoding of the trie example found in [4]. A trie is a finite map from keys to values whose structure depends on the type of keys, here encoded as products and sums in GADT variants:

```haskell
data Either a b where
  Left :: a \rightarrow Either a b
  Right :: b \rightarrow Either a b

data Trie k v where
  TUnit ::
    Maybe v \rightarrow Trie () v
  TSum :: forall k1 k2.
    Trie k1 v \rightarrow Trie k2 v \rightarrow Trie (Either k1 k2) v
  TProd :: forall k1 k2.
    Trie k1 (Trie k2 v) \rightarrow Trie (k1, k2) v
```

A trie for a unit type is maybe one value, a trie for a sum is a product of tries, and a trie for a product is a composition of tries. An important operation on tries is the merging of two maps with the same domain and co-domain.

```haskell
merge :: (v \rightarrow v \rightarrow v)
  \rightarrow Trie k v \rightarrow Trie k v \rightarrow Trie k v
merge c (TUnit Nothing ) (TUnit Nothing ) = TUnit Nothing
merge c (TUnit Nothing ) (TUnit (Just v')) = TUnit (Just v')
merge c (TUnit (Just v)) (TUnit Nothing ) = TUnit (Just v)
merge c (TUnit (Just v)) (TUnit (Just v')) = TUnit (Just (c v v'))
merge c (TSum ta tb) (TSum ta' tb') = TSum (merge c ta ta') (merge c tb tb')
merge c (TProd ta) (TProd ta') = TProd (merge (merge c) ta ta')
```

The second two last function clauses are interesting. The patterns of the first and second argument constrain $k$ to Either $k1 k2$ and Either $k1' k2'$, respectively. Hence, we have

$$
\text{Either } k1 k2 = k = \text{Either } k1' k2'
$$

from which we can follow $k1 = k1'$ and $k2 = k2'$.  

12
The point is that in the GADT type system we can deduce $t_1 = t'_1, \ldots, t_n = t'_n$ from $T t_1 \ldots t_n = T t'_1 \cdots t'_n$ for any $n$-ary type constructor $T$. This is the standard “decomposition” law for Herbrand type constructors. In terms of the encoding schemes described in Section 4, this means that we need to decompose the proof term associated to $T t_1 \ldots t_n = T t'_1 \cdots t'_n$ into a proof term associated to $t_i = t'_i$. But it seems impossible to define such a decomposition law in Haskell 98 using newtypes.

The explicit coercion encoding approach seems more flexible when it comes to decomposition. For simplicity, we only give parts of the definition of the decomposition law for the `Either a b` data type.

```haskell
decomp1 :: (Either a b -> Either c d) -> (a->c)
decomp f = \ a -> case (f (Left a)) of
  Left c -> c
```

We inject the `a` value into the `Either a b` data type, apply the incoming coercing function and then extract the `c` value. Notice that if (input) function `f` behaves like the identity function, the resulting (output) function behaves like the identity function as well. We can therefore argue that decomposition for `Either a b` is (correctly) definable in Haskell 98. Thus, we can rewrite the “trie” example in an equivalent form which is accepted by Haskell 98. Due to space limitations we provide the details in Appendix A.

There are many other examples which can be translated using the explicit coercion approach. A comprehensive list of examples can be found under ³

http://users.ox.ac.uk/~wolf2335/projects/translate-gadt/

In fact, it almost seems that all practical examples can be encoded. Though, not every decomposition function is definable. Here is the (contrived) critical example.

```haskell
data Foo a where
  K :: Foo a

data Erk a b c where
  I :: c -> Erk a a c
  f :: Erk (Foo a) (Foo Int) a -> a
  f (I x) = x + 1
```

First, we convince ourselves that the above program is well-typed in the GADT system. The pattern `I x` in combination with the type annotation implies that `Foo a = Foo Int`. By decomposition, we conclude that `a = Int`. Thus, the program text `x+1` can be given type `Int`. Hence, the above is well-typed. To translate the above, we need to define a function of type `EQ (Foo a) (Foo Int) -> EQ a Int`. We claim it is impossible to define such a function with satisfies the invariant. It suffices to show that a function

```haskell
decompFoo :: (Foo a->Foo Int)->(a->Int)
```

³ Examples are also part of the technical report version [27].
with the property that \texttt{decompFoo} (\texttt{\ x->x}) evaluates to \texttt{\ x->x} is not definable.

The problem here is that a value of type \texttt{a} cannot be injected into a value of
type \texttt{Foo a}. So, clearly the incoming function of type \texttt{Foo a->Foo Int} is useless. Effectively, we could omit the function parameter altogether. Parametricity tells us that any function of type \texttt{a->Int} must be a constant function. Hence, \texttt{decompFoo} applied to any function of type \texttt{Foo a->Foo Int} yields a constant function. Hence, an encoding of the above critical example is impossible.

The above suggests that in order to represent the decomposition law via
the explicit coercion approach, (1) we must demand that for each data type
\texttt{T a1...a_n}, all type parameters \texttt{a1,...,a_n} appear in the argument type of at least
one constructor. This immediately rules out examples such as the above phantom
type \texttt{Foo a} and also abstract types such as \texttt{IO a}. In addition, (2) we require
that type parameters \texttt{a1, ..., a_n} appear in a positive position in the argument of
a constructor. We say that a data type \texttt{T a1...a_n} is \textit{decomposable} iff the above
conditions (1) and (2) are satisfied.

\textbf{Theorem 3 (GADTless Programming via Explicit Coercions).} Every
\textit{GADT} program which only uses decomposable types can be encoded via explicit coercions.

A proof sketch of the above result is given in Appendix C.

\section{Conclusion and Related Work}

The main result of this paper is that Pottier’s and Gauthier’s [19] GADT-based
polymorphic typed defunctionalization can actually be expressed in Haskell 98
using newtypes to encode GADTs (Section 5.2). Thus, a type-preserving polyn-
morphic defunctionalization becomes possible for many systems without having
to extend the typed-intermediate language with GADTs (or something equiva-
 lent).

The idea of using newtypes to encode GADTs originates from the work by
Baars and Swierstra [1], Hinze and Cheney [3], and possibly many others. But
many GADT programs can only be encoded using explicit coercion functions.
The idea of using explicit coercion functions to mimic GADT style program appears
first in the work by Yang [29]. His method has been re-invented later by
Chen, Zhu and Xi [2]. In our second main result, we establish for the first time
sufficient conditions under which an encoding of GADTs with explicit coercions
is possible in Haskell 98 (Section 6). The disadvantage of the explicit coercion
compared to the newtypes approach is that explicit coercions impose a run-time
penalty. Furthermore, there are (albeit contrived) GADT programs which cannot
seem to be encoded via explicit coercions. This happens if one cannot “decom-
pose” type equations (respectively their associated proof terms/coercions).

Interestingly, the “decomposition” problem also arises when translating type
class programs [7].

\footnote{A phantom type has a type parameter which does not appear as an argument of a
constructor. The constructors of an abstract type are not visible to the user.}
class Foo a where foo :: a->Int
instance Foo a => Foo [a] where
    foo [] = 1
    foo _ = 2
bar :: Foo [a] => a->Int
bar = foo

Based on the System F-style translation scheme described in [7], we are unable to translate function bar. The program text demands a dictionary for Foo a but the annotation only supplies a dictionary for Foo [a]. This is the wrong way around. The instance declaration tells us how to construct Foo [a] given Foo a but the other direction does not hold in general.

Many other programming language features appear to be (roughly) equivalent in terms of expressive power to GADTs. For example, type classes, as pioneered by Weirich [28], can also be used to encode GADT style behavior. Kiselyov [10] provides numerous examples of ingenious type class encodings of GADT programs. The gist of his idea is to turn each (value) pattern clause into an (type class) instance declaration. A drawback of the type class encoding is that the original GADT program has to go under some substantial rewrites. Furthermore, GADTs are closed whereas type class instances are open. Hence, both concepts seem to complement, rather than substitute, each other.

In conclusion, we could show that some non-trivial GADT-based transformations and programs can actually be expressed in Haskell 98. Our results help to get a better understanding of GADTs and under which conditions we can replace them with features available in Haskell 98.

Acknowledgments

We thank Gregory Duck, Manuel M. T. Chakravarty, James Cheney, Oleg Kiselyov, Simon Peyton Jones and Jeremy Wazny for their comments on previous versions of this paper.

References

7. C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in
10. O. Kiselyov. Typed lambda-expressions without gadts.
http://www.haskell.org/~pipermail/haskell-cafe/2005-January/008212.html,
12. K. Läufer and M. Odersky. An extension of ML with first-class abstract types. In
13. Robin Milner, Mads Tofte, and David Macqueen. The Definition of Standard ML.
15. M. Odersky, M. Sulzmann, and M Wehr. Type inference with constrained types.
Health & Science University, OGI School of Science & Engineering, September
2004.
18. S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-
based type inference for GADTs. In Proc. of ICFP ’06, pages 50–61. ACM Press,
2006.
Workshop on ML, volume 148 of Electronic Notes in Theoretical Computer Science,
21. John C. Reynolds. Definitional interpreters for higher-order programming lan-
23. Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified
with type equality coercions. In Proc. of ACM SIGPLAN Workshop on Types in
types to existential types. Technical Report TR22/04, The National University of
A  GADT Encoding Examples via Explicit Coercions

type EQ a b = (a->b,b->a)

sym :: EQ a b -> EQ b a
sym (f,g) = (g,f)

data Trie k v where
  TUnit ::
    EQ k () -> Maybe v -> Trie k v
  TSum :: forall k1 k2.
    EQ k (Either k1 k2) -> Trie k1 v -> Trie k2 v -> Trie k v
  TProd :: forall k1 k2.
    EQ k (k1,k2) -> Trie k1 (Trie k2 v) -> Trie k v

merge :: (v -> v -> v) -> Trie k v -> Trie k v -> Trie k v
merge c (TUnit (g1,h1) Nothing ) (TUnit (g2,h2) Nothing ) =
  TUnit (g1,h1) Nothing
merge c (TUnit (g1,h1) Nothing ) (TUnit (g2,h2) (Just v')) =
  TUnit (g1,h1) (Just v')
merge c (TUnit (g1,h1) (Just v)) (TUnit (g2,h2) Nothing ) =
  TUnit (g1,h1) (Just v)
merge c (TUnit (g1,h1) (Just v)) (TUnit (g2,h2) (Just v')) =
  TUnit (g1,h1) (Just (c v v'))
merge c (TSum (g1,h1) ta tb) (TSum (g2,h2) ta' tb') =
  TSum (g1,h1) (merge c ta ((comp ((decompE1 (g1.h2)),(decompE1 (g2.h1)))) ta'))
  (merge c tb ((comp ((decompE2 (g1.h2)),(decompE2 (g2.h1)))) tb'))
merge c (TProd (g1,h1) ta) (TProd (g2,h2) ta') =
  let k1'k1 = ((decompP1 (g1.h2)),(decompP1 (g2.h1)))
    k2'k2 = ((decompP2 (g1.h2)),(decompP2 (g2.h1)))
  in  TProd (g1,h1) (merge (merge c ta (comp2 k1'k1 k2'k2 ta')))}

comp :: EQ a b -> Trie a v -> Trie b v
comp (f',g') (TUnit (f,g) v) = TUnit (f.g', f'.g) v
comp (f',g') (TSum (f,g) t1 t2) = TSum (f.g', f'.g) t1 t2
comp (f',g') (TProd (f,g) t1) = TProd (f.g', f'.g) t1

comp' :: EQ v v' -> Trie k v -> Trie k v
comp' (f',g') (TUnit eq v) = TUnit eq (compM f' v)
comp' eq' (TSum eq t1 t2) = TSum eq (comp' eq' t1) (comp' eq' t2)
comp' eq' (TProd eq t1) = TProd eq (comp' ((comp' eq'), (comp' (sym eq')))) t1)

comp2 :: EQ a b -> EQ a' b' -> Trie a (Trie a' v) -> Trie b (Trie b' v)
comp2 eq1 eq t = comp' (comp eq, comp (sym eq)) (comp eq1 t)

compM :: (a->b) -> Maybe a -> Maybe b
compM f Nothing = Nothing
compM f (Just x) = Just (f x)

decompP1 :: ((a,b) -> (c,d)) -> (a->c)
decompP1 \( f = \lambda a \to \text{case } (f \ (a, \text{undefined})) \) of
\( (c, \_ ) \to c \)

decompP2 :: ((a, b) \to (c, d)) \to (b \to d)
decompP2 \( f = \lambda b \to \text{case } (f \ (\text{undefined}, b)) \) of
\( (_, d ) \to d \)

decompE1 :: (Either a b -> Either c d) -> (a->c)
decompE1 \( f = \lambda a \to \text{case } (f \ (\text{Left } a)) \) of
\( \text{Left } c \to c \)

decompE2 :: (Either a b -> Either c d) -> (b->d)
decompE2 \( f = \lambda b \to \text{case } (f \ (\text{Right } b)) \) of
\( \text{Right } d \to d \)

B  Defunctionalization Proofs

Theorem 1 follows straightforwardly from the two lemmas below. We use \( \sigma_{\text{apply'}} \) as an abbreviation of function \( \text{apply'} \)'s type \( \text{Arrow'} \ a_1 a_2 \to a_1 \to a_2 \). We assume types of \( X' \), \( \text{refl}, t_0 \) and \( \text{from} \) are known and are omitted from the environment.

**Lemma 1.** Let \( \Gamma \vdash^{HM} e : t \) and \( \Gamma \vdash^{DF} e : t \rightsquigarrow e' \). Then \( \llbracket \Gamma \rrbracket \cup \text{apply'} : \sigma_{\text{apply'}} \vdash^{HM} e' : [t] \).

**Proof.** By structural induction on the derivation of \( \Gamma \vdash^{DF} e : t \rightsquigarrow e' \).

\( \circ \) **Case** (Abs). The rule is

\[
\begin{align*}
\bar{y} &= f s(\lambda x.e) & \bar{y} &= y_1, \ldots, y_n & \Gamma \vdash^{HM} & y_i : t_i & \text{for } i = 1, \ldots, n \\
\Gamma \vdash^{HM} & X' : (a_1 : \llbracket t_1 \rrbracket) \to (a_2 : \llbracket t_2 \rrbracket) \to [t] \to \text{Arrow } a_1 a_2 \\
\Gamma \cup \{ x : a_1, e_1 : a_1 : \llbracket t_1 \rrbracket, e_2 : a_2 : \llbracket t_2 \rrbracket \} \vdash^{DF} & e : t_2 \rightsquigarrow e' \\
\hline
\Gamma \vdash^{DF} & \lambda x.e : t_1 \to t_2 \rightsquigarrow X' \text{ refl refl } \bar{y}
\end{align*}
\]

Given \( \Gamma \vdash^{HM} \text{refl } a : a : \llbracket t_1 \rrbracket \), by rule (App), we have \( \Gamma \vdash^{HM} X' \text{ refl refl } \bar{y} : \text{Arrow'} \ [t_1] [t_2] \). Since \( [t_1 \to t_2] = \text{Arrow'} [t_1] [t_2] \), we conclude \( \Gamma \vdash^{HM} X' \text{ refl refl } \bar{y} : [t_1 \to t_2] \).

\( \circ \) **Case** (App). The rule is

\[
\begin{align*}
\Gamma \vdash^{DF} & e_1 : t_2 \to t \rightsquigarrow e'_1 & \Gamma \vdash^{DF} & e_2 : t_2 \rightsquigarrow e'_2 \\
\hline
\Gamma \vdash^{DF} & e_1 e_2 : t \rightsquigarrow \text{apply'} e'_1 e'_2
\end{align*}
\]

By induction, we have

\[
\llbracket \Gamma \rrbracket . \text{apply'} : \sigma_{\text{apply'}} \vdash^{HM} e'_1 : \text{Arrow'} [t_2] [t]
\]
\[
\llbracket \Gamma \rrbracket . \text{apply'} : \sigma_{\text{apply'}} \vdash^{HM} e'_2 : [t_2]
\]

By rule (App), we conclude \( \llbracket \Gamma \rrbracket . \text{apply'} : \sigma_{\text{apply'}} \vdash^{HM} \text{apply'} e'_1 e'_2 : [t] \).
Lemma 2. \( \text{apply}' : \sigma_{\text{apply}'} \vdash^H \lambda f.\text{arg}.\text{case}\ f \text{ of } \text{patch}_x : \sigma_{\text{apply}'} \).

Proof. Let’s consider a lambda abstraction whose defunctionalization derivation ends with

\[
\bar{y} = f(\lambda x.e) \quad \bar{y} = y_1, \ldots, y_n \quad \Gamma \vdash^H y_i : t_i \quad \text{for } i = 1, \ldots, n
\]

\[
\Gamma \vdash^H X' : (a_1 ::= [t_1]) \rightarrow (a_2 ::= [t_2]) \rightarrow [f] \rightarrow \text{Arrow } a_1 a_2
\]

\[
\Gamma \cup \{ x : a_1, ep_1 : a_1 ::= t_1, ep_2 : a_2 ::= t_2 \} \vdash^D e : t_2 \rightsquigarrow e'
\]

Applying Lemma 1 to the premise yields

\[
[\Gamma] \cup \{ \text{apply}' : t_{\text{apply}'}, x : a_1, ep_1 : a_1 ::= [t_1], ep_2 : a_2 ::= [t_2] \} \vdash^H e' : [t_2]
\]

Applying rule (App), we have

\[
[\Gamma] \cup \{ \text{apply}' : t_{\text{apply}'}, x : a_1, ep_1 : a_1 ::= [t_1], ep_2 : a_2 ::= [t_2] \} \vdash^H \text{(to eq)} e' : a_2
\]

Note that \( \text{patch}_x \) is of the form \( X' \ e_{\text{p}1} \ e_{\text{p}2} \ \bar{y} \rightarrow \text{let } x=\text{arg} \text{ in } (\text{to eq}) e' \). Given the lambda variable \( \text{arg} \)'s type \( a_1 \). Pattern matching of \( X' \) produces the bindings \( ep_1 : a_1 ::= [t_1] \) and \( ep_2 : a_2 ::= [t_2] \). The let introduction gives us \( x : a_1 \).

By rules (Let) and (Clause), we have

\[
[\Gamma] \cup \{ \text{apply}' : t_{\text{apply}'}, \text{arg} : a_1 \} \vdash^H \text{X'} \ e_{\text{p}1} \ e_{\text{p}2} \ \bar{y} \rightarrow \text{let } x = \text{arg} \text{ in } (\text{to eq}) e' : \text{Arrow'} a_1 a_2 \rightarrow a_2
\]

Since \( \bar{y} = f(\lambda x.e) \), we have

\[
\{ \text{apply}' : t_{\text{apply}'}, \text{arg} : a_1 \} \vdash^H \text{X'} \ e_{\text{p}1} \ e_{\text{p}2} \ \bar{y} \rightarrow \text{let } x = \text{arg} \text{ in } (\text{to eq}) e' : \text{Arrow'} a_1 a_2 \rightarrow a_2
\]

Because this holds for every lambda abstraction, we conclude

\[
\text{apply}' : \sigma_{\text{apply}'} \vdash^H \lambda f.\text{arg}.\text{case}\ f \text{ of } \text{patch}_x : \sigma_{\text{apply}'}
\]

Proof of Theorem 2.

Proof. By structural induction on the derivation of \( \Gamma \vdash^H e : t \).

\( \circ \) Case (Abs). The rule is

\[
\Gamma \cup \{ x : t_1 \} \vdash^H e : t_2
\]

\[
\Gamma \vdash^H \lambda x.e : t_1 \rightarrow t_2
\]

By induction, we have \( \Gamma \cup \{ x : t_1 \} \vdash^D e : t_2 \rightsquigarrow e' \). Since \( \{ep_1, ep_2\} \cap \text{fv} (e) = \emptyset \),
we have \( \Gamma \cup \{ x : t_1, ep_1 : a_1 ::= t_1, ep_2 : a_2 ::= t_2 \} \vdash^D e : t_2 \rightsquigarrow e' \).

If we apply rule (Eq1) with \( eq_1 \) to every occurrence of \( x \) in \( e \), we obtain

\[
\Gamma \cup \{ x : a_1, ep_1 : a_1 ::= t_1, ep_2 : a_2 ::= t_2 \} \vdash^D e : t_2 \rightsquigarrow e''
\]

where \( e'' \) is \( e' \) with every occurrence of \( x \) substituted by (from eq1) \( x \). Thus,
by rule (Abs), we conclude \( \Gamma \vdash^D \lambda x.e : t_1 \rightarrow t_2 \rightsquigarrow X' \ \text{refl} \ \bar{y} \).
C GADTless Explicit Coercion Proofs

The essence to GADTless encoding is the mimicking of GADT typing rule (Eq) with explicit coercions. A measurement of expressiveness of an encoding approach is how much valid (i.e. equivalent to identity) proof witnesses can be constructed comparing to constraint implication $\models$ (defined below) used in the GADT type system.

$$(\text{Sym}) \frac{t_1 = t_2 \in C}{\Gamma \models t_1 = t_2} \quad (\text{DComp}) \frac{\Gamma \models t_1 \rightarrow t_3 \rightarrow t_4}{C \models t_1 \rightarrow t_2 \quad C \models t_2 \rightarrow t_3 \quad C \models t_3 \rightarrow t_4}$$

$$(\text{Arrow}) \frac{\Gamma \models t_1 = t_2 \quad \Gamma \models t_3 = t_4}{\Gamma \models t_1 \rightarrow t_3 = t_2 \rightarrow t_4} \quad (\text{Comp}) \frac{\Gamma \models t_i = t'_i}{C \models T \cdot t_1 \cdots t_n = T \cdot t'_1 \cdots t'_n}$$

For $i = 1, \ldots, n$

$$(\text{Trans}) \frac{\Gamma \models t_2 = t_3}{C \models t_1 \rightarrow t_3 = t_2 \rightarrow t_4} \quad (\text{DCompT}) \frac{\Gamma \models t_1 \rightarrow t_3 = t_2 \rightarrow t_4}{C \models T \cdot t_1 \cdots t_n = T \cdot t'_1 \cdots t'_n}$$

To prove the claim of Theorem 3, it is sufficient to show that all the above rules are expressible via explicit coercions given only decomposable types are used in decomposition. The following corresponding coercion functions constructed are equivalent to the identity operationally assuming the input functions are identity.

```
sym : : EQ a b -> EQ b a
sym (f,g) = (g,f)
```

```
trans : : EQ a b -> EQ b c -> EQ a c
trans (f1,g1) (f2,g2) = (f2.f1,g1,g2)
```

```
arrow : : EQ a1 b1 -> EQ a2 b2 -> EQ (a1->a2) (b1->b2)
arrows (f1,g1) (f2,g2) = (\ g -> f2.g.g1, \ g -> g2.g.f1)
```

W.l.o.g we assume each data constructor takes exactly one argument. No phantom variables are allowed given the decomposable requirement.

```
T a = T1 a1
    | T2 a2
    | ...
    | Tn an
```

```
comp : : EQ a1 t1 -> EQ a2 t2 -> ... -> EQ an t_n -> EQ (T a) (T \bar{t})
comp (f1,g1) (f2,g2) ... (f_n,g_n) = (f1',f2',\ldots,f_n'), (g1',g2',\ldots,g_n')
```

where

```
f1' :: T a1 \ldots a_n -> T t1 a2 \ldots an
f1' (T1 x_1) = T1 (f1 x_1)
f1' x = x
```

```
20
```
Given the decomposable assumption, we only consider parameters at positive positions.

\[ f_n' :: T \ a_1 \ a_2 \ \ldots \ a_n \rightarrow T \ a_1 \ \ldots \ a_{n-1} \ t_n \]
\[ f_n' \ (T_n \ x_n) = T_n \ (f_n \ x_n) \]
\[ f_n' \ x = x \]

\[ g_1' :: T \ t_1 \ a_2 \ \ldots \ a_n \rightarrow T \ a_1 \ \ldots \ a_n \]
\[ g_1' \ (T_1 \ x_1) = T_1 \ (g_1 \ x_1) \]
\[ g_1' \ x = x \]

\[ \ldots \]
\[ g_n' :: T \ a_1 \ \ldots \ a_{n-1} \ t_n \rightarrow T \ a_1 \ \ldots \ a_n \]
\[ g_n' \ (T_n \ x_n) = T_n \ (g_n \ x_n) \]
\[ g_n' \ x = x \]

Due to laziness, evaluations of the above coercion functions do not diverge. However, it does not work in a strict language.

Decomposition can be defined as well by: 1. injecting the value into a data type whose conversion functions are available; 2. coercing the data type 3. projecting the data type which gives back the original value with a different type.

\[ \text{decompA} :: \text{EQ} \ (a_1 \rightarrow a_2) \ (b_1 \rightarrow b_2) \rightarrow \text{EQ} \ a_2 \ b_2 \]
\[ \text{decompA} \ (f,g) = (\ \lambda \ x \rightarrow f \ (\ \lambda \ y \rightarrow x) \ \text{undefined}, \ \lambda \ x \rightarrow g \ (\ \lambda \ y \rightarrow x) \ \text{undefined}) \]

Note that though a projection function \( \text{project}_i \) is partial, it always works with data constructed by the corresponding injection function \( \text{inject}_i \). Thus, coercion functions constructed above are total.