Efficient Evaluation for Untyped and Compositional Representations of Expressions

Emil Axelsson

January 12, 2015

Abstract

This report gives a simple implementation of A. Baars and S.D. Swierstra’s “Typing Dynamic Typing” using modern (GHC) Haskell features, and shows that the technique is especially beneficial in a compositional setting, where parts of the expression are defined separately.

Evaluating expressions that are represented as algebraic data types typically requires using tagged unions to represent values. Tagged unions can introduce runtime overhead due to tag checking, and this overhead is unnecessary if the evaluated expression is well-typed. Likewise, pattern matching on the constructors of the expression causes overhead which is unnecessary if the same expression is evaluated multiple times. Typing Dynamic Typing solves both of these problems by deferring all tag checking to an initial “dynamic compilation” phase after which evaluation proceeds without any tag checking or pattern matching. The problems of tag checking and pattern matching are worse in a compositional setting, and our measurements show that the technique gives especially good performance gains for compositional expressions.
1 Introduction

Writing interpreters in strongly typed languages often requires using tagged unions to account for the fact that different sub-expressions of the interpreted language may result in values of different types. Consider the following Haskell data type representing expressions with integers and Booleans:

```haskell
data Exp where
    LitB :: Bool -> Exp -- Boolean literal
    LitI :: Int -> Exp -- Integer literal
    Equ :: Exp -> Exp -> Exp -- Equality
    Add :: Exp -> Exp -> Exp -- Addition
    If :: Exp -> Exp -> Exp -> Exp -- Condition
    Var :: Name -> Exp -- Variable

type Name = String
```

In order to evaluate `Exp`, we need a representation of values – Booleans and integers – and an environment mapping bound variables to values:

```haskell
data Uni = B !Bool | I !Int
type Env a = [(Name,a)]
```

Evaluation can then be defined as follows:

```haskell
evalU :: Env Uni -> Exp -> Uni
evalU env (LitB b) = B b
evalU env (LitI i) = I i

evalU env (Equ a b) = case (evalU env a, evalU env b) of
    (B a', B b') -> B (a'==b')
    (I a', I b') -> B (a'==b')

evalU env (Add a b) = case (evalU env a, evalU env b) of
    (I a', I b') -> I (a'+b')

evalU env (If c t f) = case (evalU env c, evalU env t, evalU env f) of
    (B c', B t', B f') -> B $ if c' then t' else f'
    (B c', I t', I f') -> I $ if c' then t' else f'

evalU env (Var v) = fromJust $ lookup v env
```

One thing stands out in this definition: There is a lot of pattern matching going on! Not only do we have to match on the constructors of `Exp` – we also have to eliminate and introduce type tags (`B` and `I`) of interpreted values for every single operation. Such untagging and tagging can have a negative effect on performance.

It should be noted that modern compilers are able to handle some of the tagging efficiently. For example the strict fields in the `Uni` type makes `evalU` essentially tagless in GHC. Yet, as our measurements show (Sec. 5), the effect of pattern matching on the `Exp` type can still have a large impact on performance. We will also see that dealing with type tags becomes much more expensive in a compositional setting, where the `Uni` type is composed of smaller types.

1.1 Avoiding Type Tags

In a dependently typed programming language, type tags can be avoided by letting the return type of `evalU` depend on the expression \[1, 10\]. This avoids the need for

\[1\] Its performance is similar to that of a tagless version of `evalU` in which we use Int to represent both integers and Booleans (see the function `evalI` in the paper’s source code).
tagged unions in functions like \texttt{eval}. The standard way to do this in Haskell is to make \texttt{Exp} an indexed GADT \cite{1}, i.e. an expression type \texttt{Exp} that is indexed by the type it evaluates to, so the evaluator has the following type:
\[
\text{eval\_T} :: \ldots \rightarrow \text{Exp\_T} \rightarrow a \rightarrow a
\]

This means that an expression of type \texttt{Exp\_T} \texttt{Int} evaluates to an actual \texttt{Int} rather than a tagged \texttt{Int}, and since the type index may vary in the recursive calls of \texttt{eval\_T}, there is no longer any need for a tagged union. In order to make evaluation for the original \texttt{Exp} type efficient, a partial conversion function from \texttt{Exp} to \texttt{Exp\_T} can be defined, and \texttt{eval\_T} can then be used to evaluate the converted expression. Such a conversion function is implemented in a blog post by Augustsson \cite{1} (though not for the purpose of evaluation).

Although going through a type-indexed expression does get rid of the tags of interpreted values, there are at least two problems with this solution:

- It requires the definition of an additional data type \texttt{Exp\_T}. If this data type is only going to be used for evaluation, this seems quite redundant.
- We still have to pattern match on \texttt{Exp\_T}, which introduces unnecessary overhead.

### 1.2 Avoiding Tags Altogether

The motivation behind this report is to implement expression languages in a compositional style using W. Swierstra’s Data Types à la Carte \cite{12}. The idea is to specify independent syntactic constructs as separate composable types, and to define functions over such types using extensible type classes (see Sec. 3.1). However, a compositional implementation is problematic when it comes to evaluation:

- **Tagged evaluation** for compositional types requires making both \texttt{Exp} and \texttt{Uni} compositional types. With Data Types à la Carte, construction and pattern matching for compositional types is generally linear in the degree of modularity, which means that the problems with tagging and pattern matching become much worse as the language is extended.
- **Tagless evaluation** for compositional types requires making both \texttt{Exp} and \texttt{Exp\_T} compositional. However, a generic representation of \texttt{Exp\_T} is quite different from that of \texttt{Exp}, and having to combine two different data type models just for the purpose of evaluation would make things unnecessarily complicated. Moreover, pattern matching on a compositional \texttt{Exp\_T} gets more expensive as the language is extended.

An excellent solution to these problems is given in A. Baars and S.D. Swierstra’s “Typing Dynamic Typing” \cite{5}. Their approach is essentially to replace \texttt{Exp\_T} \texttt{a} by a function \texttt{env} \rightarrow \texttt{a} where \texttt{env} is the runtime environment. One may think of the technique as fusing the conversion from \texttt{Exp} to \texttt{Exp\_T} with the evaluator \texttt{eval\_T} so that the \texttt{Exp\_T} representation disappears. Put simply, this approach avoids the problem of having to make a compositional version of \texttt{Exp\_T}. Not only does this lead to a simpler implementation; it also makes evaluation more efficient, as we get rid of the pattern matching on \texttt{Exp\_T}.

The rest of the report is organized as follows: Sec. 2 gives a simple implementation of Typing Dynamic Typing using modern (GHC) Haskell features. Sec. 3 implements the technique for compositional data types based on Data Types à la Carte. Sec. 4
generalizes the compositional implementation using a novel representation of open type representations. Finally, Sec. 5 presents a comparison of different implementations of evaluation in terms of performance.

The source of this report is available as a literate Haskell file. Certain parts of the code are elided from the report, but the full definitions are found in the source code. A number of GHC-specific extensions are used.

2 Typing Dynamic Typing

Typing Dynamic Typing is a technique for evaluating untyped representations of expressions without any checking of type tags or pattern matching, other than in an initial “typed compilation” stage [5]. In this section, we will present the technique using the Exp type from Sec. 1.

2.1 Type-Level Reasoning

We will start by building a small toolbox for type-level reasoning needed to implement evaluation. At the core of this reasoning is a representation of the types in our expression language:

```haskell
data Type a where
  BType :: Type Bool
  IType :: Type Int
```

Type is an indexed GADT, which means that pattern matching on its constructors will refine the type index to Bool or Int depending on the case [11].

Consider the following function that converts a value to an integer:

```haskell
toInt :: Type a → a → Int
toInt BType a = if a then 1 else 0
        
toInt IType a = a
```

In the first case, matching on BType refines the type index to Bool, and similarly, in the second case, the index is refined to Int. On the right-hand sides, the refined types can be used as local assumptions, which means that the result expressions are type-correct even though a has different types in the two cases.

Next, we define a type for witnessing type-level constraints [3]

```haskell
data Wit c where
    Wit :: c ⇒ Wit c
```

Similarly to how pattern matching on BType/IType introduces local constraints in the right-hand sides of toInt, pattern matching on Wit introduces c as a local constraint. A constraint can be e.g. a class constraint such as (Num a) or an equality between two types (a ~ b).

Equality witnesses for types in our language are constructed by this function:

---

[2] 3 The c parameter of Wit has kind Constraint, which is allowed by the recent GHC extension ConstraintKinds. Wit is available as the type Dict in the constraints package: http://hackage.haskell.org/package/constraints
class TypeRep t where
  typeEq :: t a → t b → Maybe (Wit (a ~ b))
  witEq :: t a → Maybe (Wit (Eq a))
  witNum :: t a → Maybe (Wit (Num a))

instance TypeRep Type where
  typeEq BType BType = Just Wit
  typeEq IType IType = Just Wit
  typeEq _ _ = Nothing
  witEq BType = Just Wit
  witEq IType = Just Wit
  witNum IType = Just Wit
  witNum BType = Nothing

Figure 1: Overloaded witnessing functions.

typeEq :: Type a → Type b → Maybe (Wit (a ~ b))
typeEq BType BType = Just Wit
typeEq IType IType = Just Wit
typeEq _ _ = Nothing

Type equality gives us the ability to coerce types dynamically:

  coerce :: Type a → Type b → a → Maybe b
  coerce ta tb a = do
    Wit ← typeEq ta tb
    return a

Note how pattern matching on Wit allows us to assume that a and b are equal for the rest of the do block. This makes it possible to return a as having type b. Such coercions are at the core of the compiler that will be defined in Sec. 2.2.

To prepare for the compositional implementation in Sec. 3, we overload typeEq on the type representation. The code is shown in Fig. 1 where we have also added functions for witnessing Eq and Num constraints.

2.2 Typed Compilation

The technique in Typing Dynamic Typing is to convert an untyped expression to a run function env → a, where env is the runtime environment (holding values of free variables) and a is the result type. This run function will perform completely tagless evaluation – no checking of type tags and no pattern matching on expression constructors. Thus, the conversion function can be seen as a “compiler” that compiles an expression to a tagless run function.

Since the result type of the run function depends on the expression at hand, we have to hide this type using existential quantification. Compiled expressions are thus defined as:

  data CompExp t env where
    (:=) :: (env → a) → t a → CompExp t env

The argument t a paired with the run function is meant to be a type representation allowing us to coerce the existential type a.

To be able to compile variables, we need a symbol table giving the run function for each variable in scope. For this, we use the previously defined Env type:

  type SymTab t env = Env (CompExp t env)
\((\vdash) \::\ SymTab \ Type \ env \rightarrow \ Exp \rightarrow \ Maybe \ (CompExp \ Type \ env)\)

\[
\begin{align*}
gamma \vdash \ LitB \ b &= \ return \ (\lambda_\_ \rightarrow b) :: \ BType \\
gamma \vdash \ LitI \ i &= \ return \ (\lambda_\_ \rightarrow i) :: \ IType \\
gamma \vdash \ Var \ v &= \ lookup \ v \ gamma \\
gamma \vdash \ Equ \ a \ b &= \ do \\
&\quad a' :: ta \leftarrow \ gamma \vdash a \\
&\quad b' :: tb \leftarrow \ gamma \vdash b \\
&\quad \text{Wit} \leftarrow \ typeEq \ ta \ tb \\
&\quad \text{Wit} \leftarrow \ witEq \ ta \\
&\quad \text{let} \ \text{run} \ e = a' \ e == b' \ e \\
&\quad \ return \$ \ \text{run} :: \ BType \\
gamma \vdash \ Add \ a \ b &= \ do \\
&\quad a' :: ta \leftarrow \ gamma \vdash a \\
&\quad b' :: tb \leftarrow \ gamma \vdash b \\
&\quad \text{Wit} \leftarrow \ typeEq \ ta \ tb \\
&\quad \text{Wit} \leftarrow \ witNum \ ta \\
&\quad \text{let} \ \text{run} \ e = a' \ e + b' \ e \\
&\quad \ return \$ \ \text{run} :: \ ta \\
gamma \vdash \ If \ c \ t \ f &= \ do \\
&\quad c' :: BType \leftarrow \ gamma \vdash c \\
&\quad t' :: tt \leftarrow \ gamma \vdash t \\
&\quad f' :: tf \leftarrow \ gamma \vdash f \\
&\quad \text{Wit} \leftarrow \ typeEq \ tt \ tf \\
&\quad \text{let} \ \text{run} \ e = \ if \ c' \ e \ then \ t' \ e \ else \ f' \ e \\
&\quad \ return \$ \ \text{run} :: \ tt
\end{align*}
\]

Figure 2: Definition of typed compilation \((\vdash)\).

The compiler is defined in Fig. 2. Compilation of literals always succeeds, and the result is simply a constant function paired with the appropriate type representation.

For \textit{Equ}, we recursively compile the arguments and bind their run functions and types. In order to use \textit{==} on the results of these run functions, we first have to prove that the arguments have the same type and that this type is a member of \textit{Eq}. We do this by pattern matching on the witnesses returned from \textit{typeEq} and \textit{witEq}. The use of \textit{do} notation and the \textit{Maybe} monad makes it convenient to combine witnesses for multiple constraints: If any function fails to produce a witness, the whole definition fails to produce a result. If the witnesses are produced successfully, we have the necessary assumptions for \(a' \ e == b' \ e\) to be a well-typed expression. Compilation of \textit{Add} and \textit{If} follows the same principle.

For variables, the result is obtained by a lookup in the symbol table. Note that this lookup may fail, in which case the compiler returns \textit{Nothing}. However, lookup failure can only happen at “compile time” (i.e. in \((\vdash)\)). If we do get a result from \((\vdash)\), this \textit{CompExp} will evaluate variables by their run function applied to the runtime environment of type \textit{env}. If \textit{env} is a lookup table, we can also get lookup failures at run time. But as we will see in the next section, it is possible to make \textit{env} a typed heterogeneous collection, such that no lookup failures can occur at run time.

### 2.3 Local Variables

Although the \textit{Exp} type contains a constructor for variables, there are no constructs that bind local variables. In order to make the example language more interesting, we add two such constructs:

\footnote{In order to keep the types simple, we avoid adding object-level functions to the language, but the technique in this report also works for functions.}
data Exp where
  ...
  Let :: Name \rightarrow Exp \rightarrow Exp \rightarrow Exp  -- Let binding
  Iter :: Name \rightarrow Exp \rightarrow Exp \rightarrow Exp \rightarrow Exp  -- Iteration

The expression \texttt{Let "x" \ a \ b} binds "x" to the expression \texttt{a} in the body \texttt{b}. The expression \texttt{Iter "x" \ l \ i \ b} will perform \texttt{l} iterations of the body \texttt{b}. In each iteration, the previous state is held in variable "x" and the initial state is given by \texttt{i}. For example, the following expression computes \(2^8\) by iterating \(\lambda x \rightarrow x+x\) eight times:

\[
\text{ex}_1 = \text{Iter "x" (LitI 8) (LitI 1) (Var "x" 'Add' Var "x")}
\]

So far, we have left the runtime environment completely abstract. The symbol table determines how each variable extracts a value from this environment. In order to compile the constructs that introduce local variables, we need to be able to extend the environment. The following function adds a new name to the symbol table and extends the runtime environment by pairing it with a new value:

\[
\text{ext :: (Name, t a)} \rightarrow \text{SymTab t env} \rightarrow \text{SymTab t (a, env)}
\]

\[
\text{ext (v,ta) gamma} = (v, \text{fst} :: t) : \text{map shift gamma}
\]

\[
\text{where shift (w, get :: u)} = (w, (\text{get o snd}) :: u)
\]

For the new variable, the extraction function is \texttt{fst}, and for every other variable, we compose the previous extraction function with \texttt{snd}. Extending the environment multiple times leads to a runtime environment of the form \((a,(b,(\ldots, env)))\), which we can see as a list of heterogeneously typed values.

Next, the compilation of \texttt{Let} and \texttt{Iter}:

\[
\text{gamma} \vdash \text{Let v a b = do}
\]

\[
a' :: \text{ta} \leftarrow \text{gamma} \vdash a
\]

\[
b' :: \text{tb} \leftarrow \text{ext (v,ta) gamma} \vdash b
\]

\[
\text{return$ (\lambda e \rightarrow b' (a' e, e))} :: \text{tb}
\]

\[
\text{gamma} \vdash \text{Iter v l i b = do}
\]

\[
l' :: \text{IType} \leftarrow \text{gamma} \vdash l
\]

\[
i' :: \text{ti} \leftarrow \text{gamma} \vdash i
\]

\[
b' :: \text{tb} \leftarrow \text{ext (v,ti) gamma} \vdash b
\]

\[
\text{Wit} \leftarrow \text{typeEq ti tb}
\]

\[
\text{return$ (\lambda e \rightarrow \text{iter l' e} (i' e) (\lambda s \rightarrow b' (s,e)))} :: \text{tb}
\]

In both cases, the body \texttt{b} is compiled with an extended symbol table containing the local variable. Likewise, when using the compiled body \texttt{b'} in the run function, it is applied to an extended runtime environment with the value of the local variable added to the original environment.

Since \texttt{Iter} will be used in the benchmarks in Sec. 5, the semantic function \texttt{iter} is defined as a strict tail-recursive loop:

\[
\text{iter :: Int \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a}
\]

\[
\text{iter l i b = go l i}
\]

\[
\text{where go !0 !a = a}
\]

\[
\text{go !n !a = go (n-1) (b a)}
\]

\subsection*{2.4 Evaluation}

Using the typed compiler, we can now define the evaluator for \texttt{Exp}:
eval :: Type a → Exp → Maybe a

```
    eval t exp = do
        a ::: ta ← ([] ::: SymTab Type ()) ⊢ exp
        Wit ⊢ typeEq t ta
        return $ a ()
```

The caller has to supply the anticipated type of the expression. This type is used to coerce the compilation result. The expression is assumed to be closed, so we start from an empty symbol table and runtime environment.

Finally, we can try evaluation in GHCi:

```
*Main> eval IType ex1
Just 256
```

Let us take a step back and ponder what has been achieved so far. The problem was to get rid of the tag checking in the eval function. We have done this by breaking evaluation up in two stages: (1) typed compilation, and (2) running the compiled function. But since the compiler still has to check the types of all sub-expressions, have we really gained anything from this exercise? The crucial point is that since the language contains iteration, the same sub-expression may be evaluated many times, while the compiler only traverses the expression once. In contrast, evalU (extended with Iter) has to perform tag checking and pattern matching at every loop iteration.

3 Implementation for Compositional Data Types

So far, we have only considered a closed expression language, represented by \( \text{Exp} \), and a closed set of types, represented by \( \text{Type} \). However, the method developed is general and works for any language of similar structure. As in our previous research [4], a main aim is to provide a generic library for EDSL implementation. The library should allow modular specification of syntactic constructs and manipulation functions so that an EDSL implementation can be done largely by assembling reusable components.

3.1 Compositional Data Types

In order to achieve a compositional implementation of tagless evaluation, we need to use open representations of expressions and types. For this, we use the representation in Data Types à la Carte [12]:

```
data Term f = In (f (Term f))
data (f :+: g) a = InL (f a) | InR (g a)
```

infixr :+::

Term is a fixed-point operator turning a base functor \( f \) into a recursive data type with a value of \( f \) in each node. Commonly, \( f \) is a sum type enumerating the constructors in the represented language. The operator :+: is used to combine two functors \( f \) and \( g \) to a larger one by tagging \( f \) nodes with \( \text{InL} \) and \( g \) nodes with \( \text{InR} \).

A redefinition of our \( \text{Exp} \) type using Data Types à la Carte can look as follows:

```
data ArithF a = LitI Int | AddF a a
data LogicF a = LitB Bool | EquF a a | IfF a a a
data BindingF a = VarF Name | LetF Name a a | IterF Name a a a
type ExpC = Term (ArithF :+: LogicF :+: BindingF)
```
Here, we made the somewhat arbitrary choice to divide the constructors into three subgroups. This will allow us to demonstrate the modularity aspect of the implementation.

A problem with nested sums like \texttt{Arith} \mathbin{:+:} \texttt{Logic} \mathbin{:+:} \texttt{Binding} is that the constructors in this type have several layers of tagging. For example, a simple expression representing the variable "x" is constructed as follows:

\[
\text{vExp} = \text{In} \ $ \ \text{InR} \ $ \ \text{InR} \ $ \ \text{Var} \ \text{F} \ "x" :: \text{Exp}_C
\]

Similarly, pattern matching on nested sums quickly becomes unmanageable. The solution provided in Data Types à la Carte is to automate tagging and untagging by means of the type class in Fig. 3. Informally, a constraint \( f :<: g \) means that \( g \) is a nested sum in which \( f \) appears as a term. This class allows us to write the above example as follows:

\[
\text{vExp}' = \text{In} \ $ \ \text{In} \ $ \ \text{Var} \ \text{F} \ "x" :: \text{Exp}_C
\]

Importantly, this latter definition is immune to changes in the \texttt{Exp}_C type (such as adding a new functor to the sum).

### 3.2 Compositional Evaluation

To make the typed compiler from Sec. 2 compositional, we simply introduce a type class parameterized on the type representation and the base functor:

\[
\text{class} \ \text{Compile} \ t \ f \ \text{where} \\
\text{compile} \ F :: \text{Compile} \ t \ g \\
\rightarrow \SymTab \ t \ env \\
\rightarrow \text{f} \ (\text{Term} \ g) \\
\rightarrow \text{Maybe} \ (\text{CompExp} \ t \ env)
\]

The constraint \text{Compile} \ t \ g makes it possible to recursively compile the sub-terms. Sub-terms have type \text{Term} \ g rather than \text{Term} \ f. This is so that \( f \) can be a smaller functor that appears as a part of \( g \). For example, in the instance for \text{Logic} \mathbin{:+:} \text{Bindings} \mathbin{:+:} \text{Arith}, \text{g} can be the sum \text{Arith}:\mathbin{:+:} \text{Logic} \mathbin{:+:} \text{Bindings}.

The main compilation function just unwraps the term and calls \text{compile}:

\[
\text{compile} :: \text{Compile} \ t \ f \\
\rightarrow \SymTab \ t \ env \\
\rightarrow \text{Term} \ f \\
\rightarrow \text{Maybe} \ (\text{CompExp} \ t \ env)
\]

\[
\text{compile} \ \gamma \ (\text{In} \ f) = \text{compile}_F \ \gamma \ f
\]

The reason for parameterizing \text{Compile} on the type representation is to be able to compile using different type representations and not be locked down to a particular set of types (such as \text{Type}). In order to be maximally flexible in the set of types, we can use Data Types à la Carte also to compose type representations. To do this, we break up \text{Type} into two smaller types:
instance TypeRep BType where
  typeEq BType C = Just Wit
  witEq BType C = Just Wit
  witNum BType C = Nothing

instance TypeRep IType where
  typeEq IType C = Just Wit
  witEq IType C = Just Wit
  witNum IType C = Just Wit

instance (TypeRep t1, TypeRep t2) ⇒ TypeRep (t1 :+: t2) where
  typeEq (InL t1) (InL t2) = typeEq t1 t2
  typeEq (InR t1) (InR t2) = typeEq t1 t2
  typeEq _ _ = Nothing
  witEq (InL t) = witEq t; witEq (InR t) = witEq t
  witNum (InL t) = witNum t; witNum (InR t) = witNum t

Figure 4: TypeRep instances for compositional type representations.

data BType a where BType :: BType Bool

data IType a where IType :: IType Int

The relevant TypeRep instances are given in Fig. 4. Using :+: , we can compose those into a representation that is isomorphic to Type:

type TypeC = BType :+: IType

The Compile instance for ArithF looks as follows:

instance (TypeRep t, IType :<: t) ⇒ Compile t ArithF where
  compile gamma (LitI fi) = return $ const i :: inj IType
  compile gamma (Add a b) = do
    a' :: ta ← compile gamma a
    b' :: tb ← compile gamma b
    Wit ← typeEq ta tb
    return $ (λe → a' e + b' e) :: ta

The close similarity to the code in Fig. 2 should make the instance self-explanatory. Note how the t parameter is left polymorphic, with the minimal constraint IType :<: t. This constraint is fulfilled by any type representation that includes IType. The remaining Compile instances are omitted from the report, but can be found in the source code.

We can now define an evaluation function for compositional expressions similar to the definition of eval:

evalC :: ∀t f a . (Compile t f, TypeRep t) ⇒ t a → Term f → Maybe a
evalC t exp = do
  a :: ta ← compile ([]) :: SymTab t () exp
  Wit ← typeEq t ta
  return $ a ()

4 Supporting Type Constructors

This section might be rather technical, especially for readers not familiar with the generic data type model used. However, it is possible to skip this section and move

---

5This instance requires turning on the UndecidableInstances extension. In this case, however, the undecidability poses no problems.
straight to the results without missing the main points of the report.

The compositional type representations introduced in Sec. 3.2 have one severe limitation: they do not support type constructors. For example, although it is possible to add a representation for lists of Booleans,

```haskell
data ListBType a where ListBType :: ListBType [Bool]
```

it is not possible to add a general list type constructor that can be applied to any other type.

In a non-compositional setting, what we would need is something like this:

```haskell
data Type a where
  BType :: Type [Bool]
  IType :: Type [Int]
  LType :: Type a -> Type [a]
```

This is a recursive GADT, and to make such a type compositional, we need a compositional data type model that supports type indexes. One such model is *generalized compositional data types*; see section 5 of Bahr and Hvitved [7]. Another one is the applicative model by Axelsson [4]. Here we will choose the latter, since it directly exposes the structure of data types without the need for generic helper functions, which leads to a more direct programming style. However, we expect that the former model would work as well.

### 4.1 A Generic Applicative Data Type Model

In previous work [4], we developed a generic data type model that represents data types as primitive symbols and applications:

```haskell
data AST sym sig where
  Sym :: sym sig -> AST sym sig
  (::$) :: AST sym (a -> sig) -> AST sym (Full a) -> AST sym sig
```

Symbols are introduced from the type `sym` which is a parameter. By using different `sym` types, a wide range of GADTs can be modeled. The `sig` type index gives the symbol’s signature, which captures its arity as well as the indexes of its arguments and result. Signatures are built using the following two types:

```haskell
data Full a
data a -> sig; infixr ->
```

For example, a symbol with signature `Int -> Full Bool` represents a unary constructor whose argument is indexed by `Int` and whose result is indexed by `Bool`.

We demonstrate the use of `AST` by defining typed symbols for the arithmetic sub-language of `Exp` with the expression `1+2` as an example:

```haskell
data Arith sig where
  LitI :: Int -> Arith (Full Int)
  Add :: Arith (Int -> Int) -> Full Int

  ex2 = Sym Add ::$ Sym (LitI 1) ::$ Sym (LitI 2) :: AST Arith (Full Int)
  -- 1+2
```

The `AST` type is similar to `Term` in the sense that it makes a recursive data type from a non recursive one. This observation leads to the insight that we can actually use `::`: to compose symbols just like we used it to compose base functors. We can, for example, split the `Arith` type above into two parts,
\[
\text{data LitI sig where} \quad \text{LitI} :: \text{Int} \rightarrow \text{LitI} (\text{Full Int}) \\
\text{data Add sig where} \quad \text{Add} :: \text{Add} (\text{Int} \rightarrow \text{Int} :\rightarrow \text{Full Int})
\]
giving us the following isomorphism:

\[
\text{AST Arith sig} \simeq \text{AST (LitI :+: Add)} \text{ sig}
\]

### 4.2 Compositional Type Representations

Since the AST type is an indexed GADT, pattern matching on its constructors together with the symbol gives rise to type refinement just as for the Type representation used earlier. This means that we can use AST to get compositional type representations, including support for type constructors. For our old friends, \text{Bool} and \text{Int}, the representations look as before, with the addition of \text{Full} in the type parameter:

\[
\text{data BType\_T2 sig where} \quad \text{BType\_T2} :: \text{BType\_T2} (\text{Full Bool}) \\
\text{data IType\_T2 sig where} \quad \text{IType\_T2} :: \text{IType\_T2} (\text{Full Int})
\]

Now we can also add a representation for the list type constructor, corresponding to \text{LType\_T} above:

\[
\text{data LType\_T2 sig where} \quad \text{LType\_T2} :: \text{LType\_T2} (a :\rightarrow \text{Full } [a])
\]

To see how type refinement works for such type representations, we define a generic sum function for representable types:

\[
\text{type Type\_T2 a} = \text{AST (BType\_T2 :+: IType\_T2 :+: LType\_T2)} (\text{Full a}) \\
\text{gsum :: Type\_T2 a} \rightarrow \text{a} \rightarrow \text{Int} \\
\text{gsum (Sym itype) i} | \text{Just IType\_T2} ← \text{prj itype} = i \\
\text{gsum (Sym ltype :$ t) as} | \text{Just LType\_T2} ← \text{prj ltype} = \text{sum} $ \text{map} (\text{gsum} t) \text{ as} \\
\text{gsum _ _} = 0
\]

Integers are returned as they are. For lists, we recursively \text{gsum} each element and then sum the result. We test \text{gsum} on a doubly-nested list of integers:

\[
\text{listListInt :: Type\_T2 }[[\text{Int}]] \\
\text{listListInt} = \text{Sym (inj LType\_T2) :$ (Sym (inj LType\_T2) :$ Sym (inj IType\_T2))}
\]

\[
\text{<Main>} \text{ gsum listListInt }[[1],[2,3],[4,5,6]] \\
21
\]

Unfortunately, the compositional implementation of the methods from the TypeRep class for the AST type is a bit too involved to be presented here. However, it is available in the open-typerep package on Hackage.\footnote{http://hackage.haskell.org/package/open-typerep-0.1} Part of the API is listed in Fig. 5.

To demonstrate the use of the list type, we extend our language with constructs for introducing and eliminating lists:

\[
\text{data List\_F a = Single a | Cons a a | Head a | Tail a}
\]

We use a singleton constructor instead of a constructor for empty lists, because we have to be able to assign a monomorphic type representation to each sub-expression, and the empty list gives no information about the type of its elements. (The alternative would be to use a \text{Nil} constructor that takes a type representation as argument.)
newtype TypeRepENW t a = TypeRep (AST t (Full a))

-- Constructing type reps
boolType :: (BoolType :<: t) ⇒ TypeRep t Bool
intType :: (IntType :<: t) ⇒ TypeRep t Int
listType :: (ListType :<: t) ⇒ TypeRep t a → TypeRep t [a]

-- Type equality
typeEq :: TypeEq ts ts ⇒ TypeRep ts a → TypeRep ts b → Maybe (Wit (a ~ b))

-- List the arguments of a type constructor representation
matchConM :: Monad m ⇒ TypeRep t c → m [E (TypeRep t)]

-- Existential quantification
data E e where E :: e a → E e

Figure 5: Parts of the open-typerep API.

Compilation of ListF can be defined as follows:

instance (ListType :<: t, TypeEq t t) ⇒ Compile (TypeRep t) ListF where
  compile, gamma (Single a) = do
    a' ::: ta ← compile gamma a
    return $ (λe → [a' e]) ::: listType ta
  compile, gamma (Head as) = do
    as' ::: tas ← compile gamma as
    [E ta] ← matchConM tas
    Wit ← typeEq tas (listType ta)
    return $ (λe → head (as' e)) ::: ta

Note how the code does not mention the AST type or its constructors. Type representations are constructed using functions like listType, and pattern matching is done using a combination of matchConM and typeEq. For example, in the Head case, the matchConM line checks that tas is a type constructor of one parameter called ta. The next line checks that tas is indeed a list of ta, which gives the type refinement necessary for the run function.

5 Results

To verify the claim that the method in this report achieves efficient evaluation, we have performed some measurements of the speed of the different evaluation functions: evalU, evalT, and evalC. However, out of these functions, only evalC is compositional. So to make the comparison meaningful, we have added a compositional version of evalU:

7The Compile instance for ListF is not directly compatible with the other instances in this report. The List instance uses (TypeRep t) as the type representation, while the other instances just use a constrained t. To make the instances compatible, the other instances would have to be rewritten to use the open-typerep library.

8In the measurements, evalU has been extended with a case for Iter; see the source code for details. Note that evalU is different from the other evaluation functions in that it throws an error rather than return Nothing when something goes wrong. However, our measurements show only small differences in time if evalU is rewritten to return Maybe.
\[ \text{eval}_{C3} = \text{eval}_C :: \text{Type}_3 \rightarrow \text{Exp}_3 \rightarrow \text{Maybe } a \]
\[ \text{eval}_{C10} = \text{eval}_C :: \text{Type}_{10} \rightarrow \text{Exp}_{10} \rightarrow \text{Maybe } a \]
\[ \text{eval}_{C30} = \text{eval}_C :: \text{Type}_{30} \rightarrow \text{Exp}_{30} \rightarrow \text{Maybe } a \]
\[ \text{eval}_{UC3} = \text{eval}_{UC} [] :: \text{Exp}_3 \rightarrow \text{Uni}_3 \]
\[ \text{eval}_{UC10} = \text{eval}_{UC} [] :: \text{Exp}_{10} \rightarrow \text{Uni}_{10} \]
\[ \text{eval}_{UC30} = \text{eval}_{UC} [] :: \text{Exp}_{30} \rightarrow \text{Uni}_{30} \]

\[ \text{data } X \ a \text{ instance } \text{TypeRep } X ; \text{ instance } \text{Compile } t \ X ; \text{ instance } \text{Eval}_{UC} u \ X \ g \]

\[ \text{type } \text{Exp}_3 = \text{Term} (\text{Arith}_F :+: \text{Logic}_F :+: \text{Binding}_F) \quad \text{-- 3 terms} \]
\[ \text{type } \text{Exp}_{10} = \text{Term} (X :+: X :+: \ldots :+: \text{Arith}_F :+: \text{Logic}_F :+: \text{Binding}_F) \quad \text{-- 10 terms} \]
\[ \text{type } \text{Exp}_{30} = \text{Term} (X :+: X :+: \ldots :+: \text{Arith}_F :+: \text{Logic}_F :+: \text{Binding}_F) \quad \text{-- 30 terms} \]
\[ \text{type } \text{Type}_3 = X :+: \text{BType} :+: \text{IType} \]
\[ \text{type } \text{Uni}_3 = \text{Term} (X :+: \text{B}_F :+: \text{I}_F) \quad \text{-- Etc. for Type}_{10}, \text{Type}_{30}, \text{Uni}_{10}, \text{Uni}_{30} \]

\[ \text{class } \text{Eval}_{UC} u \ f \ g \text{ where} \]
\[ \text{eval}_{UCF} :: \text{Env} (\text{Term } u) \rightarrow f (\text{Term } g) \rightarrow \text{Term } u \]
\[ \text{eval}_{UC} :: \text{Eval}_{UC} u \ f \ f \Rightarrow \text{Env} (\text{Term } u) \rightarrow \text{Term } f \rightarrow \text{Term } u \]
\[ \text{eval}_{UC} \text{ env } (\text{In } f) = \text{eval}_{UCF} \text{ env } f \]

Here we have replaced \text{Uni} by \text{Term } u, where \text{u} is a class parameter so that the universal type can be extended. The \text{f} parameter is the functor of the current node, and \text{g} is the functor of the sub-terms (and \text{f} is meant to be part of \text{g}).

Here we just give the instance for \text{Arith}. The remaining instances are in the source code.

\[ \text{instance } (\text{I}_F ::= u, \text{Eval}_{UC} u \ g \ g) \Rightarrow \text{Eval}_{UC} u \ \text{Arith}_F g \text{ where} \]
\[ \text{eval}_{UCF} \text{ env } (\text{LitI}_F \ i) = \text{In } \text{ inj } \text{ inj } \text{ I}_F \ i \]
\[ \text{eval}_{UCF} \text{ env } (\text{Add}_F \ a \ b) = \text{case } (\text{eval}_{UC} \text{ env } a, \text{eval}_{UC} \text{ env } b) \text{ of} \]
\[ (\text{In } a', \text{In } b') | \text{ Just } (\text{I}_F \ a'') \leftrightarrow \text{ prj } a' \]
\[ , \text{ Just } (\text{I}_F \ b'') \leftrightarrow \text{ prj } b' \]
\[ \rightarrow \text{In } \text{ inj } \text{ inj } \text{ I}_F \ (a''+b'') \]

This definition is similar to the \text{Add} case in \text{eval}_{U}, but here we use an open universal type, so tagging and untagging is done using \text{inj} and \text{prj} from Fig. 3. The \text{Uni} type has been decomposed into the following functors:

\[ \text{data } \text{B}_F \ a = \text{B}_F \ !\text{Bool} \]
\[ \text{data } \text{I}_F \ a = \text{I}_F \ !\text{Int} \]

As the degree of modularity increases, the functions \text{eval}_{C} and \text{eval}_{UC} become more expensive. To test this behavior, Fig. 6 defines specialized evaluation functions for varying sizes of the functor sums. The empty type \text{X} is introduced just to be able to make large functor sums.

The first benchmark is for a balanced addition tree of depth 18, where we get the following results.\footnote{All measurements were done on a Dell laptop with an Intel Core i7-4600U processor and GHC 7.8.3 with the -O2 flag.}
In this case, the expression is very large, and the cost of compiling the expression is proportional to the cost of evaluating it. In such cases, typed compilation does not give any benefits, and we are better off using eval_uc for compositional evaluation. However, such huge expressions are quite rare. It is much more common to have small expressions that are costly to evaluate.

Our next benchmark is a triply-nested loop with \( n \) iterations at each level:

\[
\text{loopNest} :: \text{Int} \to \text{Exp} \\
\text{loopNest} n = \text{iter} \ "x" \ (\text{LitI} \ n) \ (\text{LitI} \ 0) \ $ \\
\quad \text{iter} \ "y" \ (\text{LitI} \ n) \ (\text{Var} \ "x") \ $ \\
\quad \text{iter} \ "z" \ (\text{LitI} \ n) \ (\text{Var} \ "y") \ $ \\
\quad (\text{Var} \ "x" \ "Add" \ \text{Var} \ "y" \ "Add" \ \text{Var} \ "z" \ "Add" \ \text{LitI} \ 1)
\]

This is a small expression, but it is costly to evaluate. For the expression \( \text{loopNest} \ 100 \), the results are as follows:

<table>
<thead>
<tr>
<th>eval_u</th>
<th>loopNest: 0.14s</th>
<th>eval_c3</th>
<th>loopNest: 0.077s</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval_t</td>
<td>loopNest: 0.042s</td>
<td>eval_uc10</td>
<td>loopNest: 0.27s</td>
</tr>
<tr>
<td>eval_c3</td>
<td>loopNest: 0.073s</td>
<td>eval_c30</td>
<td>loopNest: 0.074s</td>
</tr>
<tr>
<td>eval_uc3</td>
<td>loopNest: 0.17s</td>
<td>eval_uc30</td>
<td>loopNest: 0.75s</td>
</tr>
</tbody>
</table>

Now we see in the first column that evaluation based on typed compilation is clearly superior. Even more interestingly, the second column shows what happens as we increase the degree of modularity: the timing for eval_c3 stays roughly constant, while the timing for eval_uc grows significantly with the modularity. The reason why typed compilation is immune to extension is that compositional data types are only present during the compilation stage, and this stage is very fast for a small expression like \( \text{loopNest} \ 100 \).

As a reference point, we have also measured the function \( \text{loopNest}_H :: \text{Int} \to \text{Int} \)

\[
\text{loopNest}_H n = \text{iter} n 0 \ $ \ \lambda x \to \text{iter} n x \ $ \ \lambda y \to \text{iter} n y \ $ \ \lambda z \to x+y+z+1
\]

which runs the expression corresponding to \( \text{loopNest} \) directly in Haskell. This function runs around 40\times faster than eval_t (for \( n = 100 \)). This is not surprising, given that \( \text{loopNest}_H \) is subject to GHC’s -O2 optimizations. We think of the typed compilation technique (Sec. 2.2) as a compiler in the sense that it removes interpretive overhead before running a function. However, the result of typed compilation is generated at run time, so it will of course not be subject to GHC’s optimizations. Interestingly, when \( \text{loopNest}_H \ 100 \) is compiled with -06, it runs at the same speed as eval_t compiled with -02. Thus, this benchmark shows that it is not unreasonable to expect an embedded evaluator to run on par with unoptimized, compiled Haskell code.

## 6 Conclusion and Related Work

We have presented an implementation of evaluation for compositional expressions based on Typing Dynamic Typing [5]. The overhead due to compositional types is only present in the initial compilation stage. After compilation, a tagless evaluation function is obtained which performs evaluation completely without pattern matching.
or risk of getting stuck. This makes the method suitable e.g. for evaluating embedded languages based on compositional data types.

The final tagless technique by Carette et al. [8] models languages using type classes, which makes the technique inherently tagless and compositional. However, in order to avoid tag checking of interpreted values, expressions have to be indexed on the interpreted type, just like in a GADT-based solution [11]. If, for some reason, we start with an untyped representation of expressions (e.g. resulting from parsing), the only way to get to a type-indexed tagless expression is by means of typed compilation, e.g. as in this report. Typed compilation to final tagless terms has been implemented by Kiselyov [9] (for non-compositional representations of source expressions).

Bahr has worked on evaluation for compositional data types [6]. However, his work focuses on evaluation strategies rather than tag elimination.

Acknowledgements

This work is funded by the Swedish Foundation for Strategic Research, under grant RAWFP. David Raymond Christiansen, Gabor Greif and the anonymous referees for TFP 2014 provided useful feedback on this report.

References
