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" [5] 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.

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Göteborg, Sweden, 2014
Technical Report No. 2014:16
ISSN 1652-926X

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:

```
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:

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

Evaluation can then be defined as follows:

```
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 $\texttt{eval}_\texttt{U}$ essentially tagless in GHC.[1] 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 $\texttt{eval}_\texttt{U}$ depend on the expression [2, 10]. This avoids the need for

[1] Its performance is similar to that of a tagless version of $\texttt{eval}_\texttt{U}$ in which we use `Int` to represent both integers and Booleans (see the function $\texttt{eval}_\texttt{I}$ in the paper's source code).

tagged unions in functions like $\mathtt{eval}_\mathsf{U}$. The standard way to do this in Haskell is to make `Exp` an indexed GADT [3, 11], i.e. an expression type $\mathtt{Exp}_\mathsf{T}$ that is indexed by the type it evaluates to, so the evaluator has the following type:

$$\mathtt{eval}_\mathsf{T} :: \ldots \to \mathtt{Exp}_\mathsf{T}\ \mathtt{a} \to \mathtt{a}$$

This means that an expression of type $\mathtt{Exp}_\mathsf{T}$ `Int` evaluates to an *actual* `Int` rather than a tagged `Int`, and since the type index may vary in the recursive calls of $\mathtt{eval}_\mathsf{T}$, there is no longer any need for a tagged union. In order to make evaluation for the original `Exp` type efficient, a partial conversion function from `Exp` to $\mathtt{Exp}_\mathsf{T}$ can be defined, and $\mathtt{eval}_\mathsf{T}$ can then be used to evaluate the converted expression. Such a conversion function is implemented in a blog post by Augustsson [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 $\mathtt{Exp}_\mathsf{T}$. If this data type is only going to be used for evaluation, this seems quite redundant.
- We still have to pattern match on $\mathtt{Exp}_\mathsf{T}$, 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 [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 `Exp` and `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 `Exp` and $\mathsf{Exp_T}$ compositional. However, a generic representation of $\mathsf{Exp_T}$ is quite different from that of `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 $\mathsf{Exp_T}$ 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" [5]. Their approach is essentially to replace $\mathsf{Exp_T}\ \mathsf{a}$ by a function $\mathsf{env} \rightarrow \mathsf{a}$ where `env` is the runtime environment. One may think of the technique as fusing the conversion from `Exp` to $\mathsf{Exp_T}$ with the evaluator $\mathsf{eval_T}$ so that the $\mathsf{Exp_T}$ representation disappears. Put simply, this approach avoids the problem of having to make a compositional version of $\mathsf{Exp_T}$. Not only does this lead to a simpler implementation; it also makes evaluation more efficient, as we get rid of the pattern matching on $\mathsf{Exp_T}$.

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.[2] 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:

```
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:

```
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]

```
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] https://github.com/emilaxelsson/tagless-eval/blob/master/Paper.lhs

[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.

```haskell
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))
```

```haskell
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.

```haskell
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:

```haskell
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)
```

```
(⊢) :: SymTab Type env → Exp → Maybe (CompExp Type env)

gamma ⊢ LitB b = return $ (λ_ → b) ::: BType
gamma ⊢ LitI i = return $ (λ_ → i) ::: IType
gamma ⊢ Var v  = lookup v gamma

gamma ⊢ Equ a b = do
    a' ::: ta ← gamma ⊢ a
    b' ::: tb ← gamma ⊢ b
    Wit       ← typeEq ta tb
    Wit       ← witEq ta
    let run e = a' e == b' e
    return $ run ::: BType

gamma ⊢ If c t f = do
    c' ::: BType ← gamma ⊢ c
    t' ::: tt    ← gamma ⊢ t
    f' ::: tf    ← gamma ⊢ f
    Wit          ← typeEq tt tf
    let run e = if c' e then t' e else f' e
    return $ run ::: tt
```

```
gamma ⊢ Add a b = do
    a' ::: ta ← gamma ⊢ a
    b' ::: tb ← gamma ⊢ b
    Wit       ← typeEq ta tb
    Wit       ← witNum ta
    let run e = a' e + b' e
    return $ run ::: ta
```

Figure 2: Definition of typed compilation (⊢).

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 `Equ`, we recursively compile the arguments and bind their run functions and types. In order to use `==` 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 `Eq`. We do this by pattern matching on the witnesses returned from `typeEq` and `witEq`. The use of **`do`** notation and the `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 `Add` and `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 `Nothing`. However, lookup failure can only happen at "compile time" (i.e. in $\vdash$). If we do get a result from $\vdash$, this `CompExp` will evaluate variables by their run function applied to the runtime environment of type `env`. If `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 `env` a typed heterogeneous collection, such that no lookup failures can occur at run time.

2.3 Local Variables

Although the `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:[4]

---

[4] 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 → Exp → Exp → Exp          -- Let binding
    Iter :: Name → Exp → Exp → Exp → Exp   -- Iteration
```

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

```
ex₁ = 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 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:

```
ext :: (Name, t a) → SymTab t env → SymTab t (a,env)
ext (v,t) gamma = (v, fst ::: t) : map shift gamma
  where shift (w, get ::: u) = (w, (get ∘ snd) ::: u)
```

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

Next, the compilation of `Let` and `Iter`:

```
gamma ⊢ Let v a b = do
    a' ::: ta ← gamma            ⊢ a
    b' ::: tb ← ext (v,ta) gamma ⊢ b
    return $ (λe → b' (a' e, e)) ::: tb

gamma ⊢ Iter v l i b = do
    l' ::: IType ← gamma            ⊢ l
    i' ::: ti    ← gamma            ⊢ i
    b' ::: tb    ← ext (v,ti) gamma ⊢ b
    Wit          ← typeEq ti tb
    return $ (λe → iter (l' e) (i' e) (λs → b' (s,e))) ::: tb
```

In both cases, the body `b` is compiled with an extended symbol table containing the local variable. Likewise, when using the compiled body `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 `Iter` will be used in the benchmarks in Sec. 5, the semantic function `iter` is defined as a strict tail-recursive loop:

```
iter :: Int → a → (a → a) → a
iter l i b = go l i
  where go !0 !a = a
        go !n !a = go (n-1) (b a)
```

2.4 Evaluation

Using the typed compiler, we can now define the evaluator for `Exp`:

```
eval_T :: Type a → Exp → Maybe a
eval_T 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_T IType ex_1
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_U` 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, `eval_U` (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 `Exp`, and a closed set of types, represented by `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 = In_L (f a) | In_R (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 $\mathtt{In}_\mathsf{L}$ and `g` nodes with $\mathtt{In}_\mathsf{R}$.

A redefinition of our `Exp` type using Data Types à la Carte can look as follows:

```
data Arith_F a   = LitI_F Int | Add_F a a
data Logic_F a   = LitB_F Bool | Equ_F a a | If_F a a a
data Binding_F a = Var_F Name | Let_F Name a a | Iter_F Name a a a
```

$\textbf{type}\ \mathtt{Exp_C = Term\ (Arith_F\ {:}{+}{:}\ Logic_F\ {:}{+}{:}\ Binding_F)}$

7

```
class f :<: g where
    inj :: f a → g a
    prj :: g a → Maybe (f a)

instance f :<: (f :+: g) where
    inj          = In_L
    prj (In_L f) = Just f
    prj _        = Nothing

instance f :<: f where
    inj = id
    prj = Just

instance (f :<: h) ⇒ f :<: (g :+: h) where
    inj          = In_R ∘ inj
    prj (In_R f) = prj f
    prj _        = Nothing
```

Figure 3: Automated tagging/untagging of nested sums (Data Types à la Carte). The code uses overlapping instances; see the paper [12] for details.

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 `Arith_F :+: Logic_F :+: Binding_F` 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:

```
vExp = In $ In_R $ In_R $ Var_F "x" :: 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:

```
vExp' = In $ inj $ Var_F "x" :: Exp_C
```

Importantly, this latter definition is immune to changes in the $\texttt{Exp}_{\texttt{C}}$ 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:

```
class Compile t f where
    compile_F :: Compile t g
              ⇒ SymTab t env → f (Term g) → Maybe (CompExp t env)
```

The constraint `Compile t g` makes it possible to recursively compile the sub-terms. Sub-terms have type `Term g` rather than `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 $\texttt{Logic}_{\texttt{F}}$, `g` can be the sum $\texttt{Arith}_{\texttt{F}}\texttt{:+:Logic}_{\texttt{F}}\texttt{:+:Binding}_{\texttt{F}}$.

The main compilation function just unwraps the term and calls $\texttt{compile}_{\texttt{F}}$:

```
compile :: Compile t f ⇒ SymTab t env → Term f → Maybe (CompExp t env)
compile gamma (In f) = compile_F gamma f
```

The reason for parameterizing `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 `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 `Type` into two smaller types:

∞

```
instance TypeRep BType where
    typeEq BType_C BType_C = Just Wit
    witEq  BType_C         = Just Wit
    witNum BType_C         = Nothing
```

```
instance TypeRep IType where
    typeEq IType_C IType_C = Just Wit
    witEq  IType_C         = Just Wit
    witNum IType_C         = Just Wit
```

```
instance (TypeRep t1, TypeRep t2) ⇒ TypeRep (t1 :+: t2) where
    typeEq (In_L t1) (In_L t2) = typeEq t1 t2
    typeEq (In_R t1) (In_R t2) = typeEq t1 t2
    typeEq _ _                 = Nothing
    witEq  (In_L t) = witEq t;  witEq  (In_R t) = witEq t
    witNum (In_L t) = witNum t; witNum (In_R t) = witNum t
```

Figure 4: TypeRep instances for compositional type representations.

```
data BType a where BType_C :: BType Bool
data IType a where IType_C :: 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 Type_C = BType :+: IType
```

The Compile instance for ArithF looks as follows:[5]

```
instance (TypeRep t, IType :<: t) ⇒ Compile t Arith_F where
    compile_F gamma (LitI_F i)  = return $ const i ::: inj IType_C
```

```
compile_F gamma (Add_F a b) = do
    a' ::: ta ← compile gamma a
    b' ::: tb ← compile gamma b
    Wit       ← typeEq ta tb
    Wit       ← witNum ta
    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 $\texttt{eval}_{\texttt{T}}$:

```
eval_C :: ∀t f a . (Compile t f, TypeRep t) ⇒ t a → Term f → Maybe a
eval_C 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 [4]. However, it is possible to skip this section and move

[5] This 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,

```
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:

```
data Type_T a where
    BType_T :: Type_T Bool
    IType_T :: Type_T Int
    LType_T :: Type_T a → Type_T [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:

```
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:

```
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 sublanguage of `Exp` with the expression `1+2` as an example:

```
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,

```
data LitI sig where LitI :: Int → LitI (Full Int)
data Add  sig where Add  :: Add (Int :→ Int :→ Full Int)
```

giving us the following isomorphism:

```
AST Arith sig  ≃  AST (LitI :+: Add) 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, `Bool` and `Int`, the representations look as before, with the addition of `Full` in the type parameter:

```
data BType_T2 sig where BType_T2 :: BType_T2 (Full Bool)
data IType_T2 sig where IType_T2 :: IType_T2 (Full Int)
```

Now we can also add a representation for the list type constructor, corresponding to `LType_T` above:

```
data LType_T2 sig where LType_T2 :: LType_T2 (a :→ Full [a])
```

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

```
type Type_T2 a = AST (BType_T2 :+: IType_T2 :+: LType_T2) (Full a)

gsum :: Type_T2 a → a → Int
gsum (Sym itype) i       | Just IType_T2 ← prj itype = i
```

```
gsum (Sym ltype :$ t) as | Just LType_T2 ← prj ltype = sum $ map (gsum t) as
gsum _ _ = 0
```

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

```
listListInt :: Type_T2 [[Int]]
listListInt = Sym (inj LType_T2) :$ (Sym (inj LType_T2) :$ Sym (inj IType_T2))
```

```
*Main> 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.[6] 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:

```
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 `Nil` constructor that takes a type representation as argument.)

[6] http://hackage.haskell.org/package/open-typerep-0.1

```
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 $\texttt{List}_\texttt{F}$ can be defined as follows:[7]

```
instance (ListType :<: t, TypeEq t t) ⇒ Compile (TypeRep t) List_F where
    compile_F gamma (Single a) = do
        a' ::: ta ← compile gamma a
        return $ (λe → [a' e]) ::: listType ta
    compile_F 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: $\mathtt{eval}_{\mathsf{U}}$[8], $\mathtt{eval}_{\mathsf{T}}$ and $\mathtt{eval}_{\mathsf{C}}$. However, out of these functions, only $\mathtt{eval}_{\mathsf{C}}$ is compositional. So to make the comparison meaningful, we have added a compositional version of $\mathtt{eval}_{\mathsf{U}}$:

---

[7] The `Compile` instance for $\mathtt{List}_{\mathsf{F}}$ is not directly compatible with the other instances in this report. The $\mathtt{List}_{\mathsf{F}}$ 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.

[8] In the measurements, $\mathtt{eval}_{\mathsf{U}}$ has been extended with a case for `Iter`; see the source code for details. Note that $\mathtt{eval}_{\mathsf{U}}$ 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 $\mathtt{eval}_{\mathsf{U}}$ is rewritten to return `Maybe`.

```
evalC3   = evalC     :: Type3 a  → Exp3  → Maybe a
evalC10  = evalC     :: Type10 a → Exp10 → Maybe a
evalC30  = evalC     :: Type30 a → Exp30 → Maybe a

evalUC3  = evalUC [] :: Exp3  → Uni3
evalUC10 = evalUC [] :: Exp10 → Uni10
evalUC30 = evalUC [] :: Exp30 → Uni30
```

```
data X a
instance TypeRep X; instance Compile t X; instance EvalUC u X g

type Exp3  = Term (ArithF :+: LogicF :+: BindingF)                  -- 3 terms
type Exp10 = Term (X:+:X:+: ... :+: ArithF :+: LogicF :+: BindingF) -- 10 terms
type Exp30 = Term (X:+:X:+: ... :+: ArithF :+: LogicF :+: BindingF) -- 30 terms

type Type3 = X :+: BType :+: IType
type Uni3  = Term (X :+: BF :+: IF)   -- Etc. for Type10, Type30, Uni10, Uni30
```

Figure 6: Specialized evaluation functions for variously-sized functors

```
class EvalUC u f g where
    evalUCF :: Env (Term u) → f (Term g) → Term u

evalUC :: EvalUC u f f ⇒ Env (Term u) → Term f → Term u
```

```
evalUC env (In f) = evalUCF env f
```

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

Here we just give the instance for `Arith`. The remaining instances are in the source code.

```
instance (IF :<: u, EvalUC u g g) ⇒ EvalUC u ArithF g where
    evalUCF env (LitIF i)  = In $ inj $ IF i
    evalUCF env (AddF a b) = case (evalUC env a, evalUC env b) of
        (In a', In b') | Just (IF a'') ← prj a'
                       , Just (IF b'') ← prj b'
                       → In $ inj $ IF (a''+b'')
```

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

```
data BF a = BF !Bool
data IF a = IF !Int
```

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

The first benchmark is for a balanced addition tree of depth 18, where we get the following results:[9]

[9] All measurements were done on a Dell laptop with an Intel Core i7-4600U processor and GHC 7.8.3 with the `-O2` flag.

$\mathtt{eval}_{\mathsf{U}}$ addTree: 0.0046s	$\mathtt{eval}_{\mathsf{C3}}$ addTree: 0.19s
$\mathtt{eval}_{\mathsf{T}}$ addTree: 0.034s	$\mathtt{eval}_{\mathsf{UC3}}$ addTree: 0.011s

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 $\mathtt{eval}_{\mathsf{UC}}$ 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:

```
loopNest :: Int → Exp
loopNest n = Iter "x" (LitI n) (LitI 0) $
               Iter "y" (LitI n) (Var "x") $
                 Iter "z" (LitI n) (Var "y") $
                   (Var "x" ‘Add‘ Var "y" ‘Add‘ Var "z" ‘Add‘ LitI 1)
```

This is a small expression, but it is costly to evaluate. For the expression `loopNest 100`, the results are as follows:

$\mathtt{eval}_{\mathsf{U}}$ loopNest: 0.14s	$\mathtt{eval}_{\mathsf{C10}}$ loopNest: 0.077s
$\mathtt{eval}_{\mathsf{T}}$ loopNest: 0.042s	$\mathtt{eval}_{\mathsf{UC10}}$ loopNest: 0.27s
$\mathtt{eval}_{\mathsf{C3}}$ loopNest: 0.073s	$\mathtt{eval}_{\mathsf{C30}}$ loopNest: 0.074s
$\mathtt{eval}_{\mathsf{UC3}}$ loopNest: 0.17s	$\mathtt{eval}_{\mathsf{UC30}}$ loopNest: 0.75s

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 $\mathtt{eval}_{\mathsf{C*}}$ stays roughly constant, while the timing for $\texttt{eval}_{\texttt{UC*}}$ 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 `loopNest 100`.

As a reference point, we have also measured the function

```
loopNest_H :: Int → Int
loopNest_H n = iter n 0 $ λx → iter n x $ λy → iter n y $ λz → x+y+z+1
```

which runs the expression corresponding to `loopNest` directly in Haskell. This function runs around 40× faster than $\texttt{eval}_{\texttt{T}}$ (for `n = 100`). This is not surprising, given that $\texttt{loopNest}_{\texttt{H}}$ 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 $\texttt{loopNest}_{\texttt{H}}$ `100` is compiled with `-O0`, it runs at the same speed as $\texttt{eval}_{\texttt{T}}$ compiled with `-O2`. 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

[1] Augustsson, L.: More LLVM. http://augustss.blogspot.se/2009/06/more-llvm-recently-someone-asked-me-on.html (2009)

[2] Augustsson, L., Carlsson, M.: An exercise in dependent types: A well-typed interpreter. In: Workshop on Dependent Types in Programming, Gothenburg (1999)

[3] Augustsson, L., Petersson, K.: Silly type families. Available at http://web.cecs.pdx.edu/~sheard/papers/silly.pdf (1994)

[4] Axelsson, E.: A generic abstract syntax model for embedded languages. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. pp. 323–334. ICFP '12, ACM (2012)

[5] Baars, A.I., Swierstra, S.D.: Typing dynamic typing. In: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming. pp. 157–166. ICFP '02, ACM (2002)

[6] Bahr, P.: Evaluation à la carte: non-strict evaluation via compositional data types. In: Nordic Workshop on Programming Theory. pp. 38–40 (2011)

[7] Bahr, P., Hvitved, T.: Compositional data types. In: Proceedings of the 7th ACM SIGPLAN workshop on Generic programming. pp. 83–94. WGP '11, ACM (2011)

[8] Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming 19(5), 509–543 (2009)

[9] Kiselyov, O.: http://okmij.org/ftp/tagless-final/#typed-compilation and http://okmij.org/ftp/tagless-final/course/index.html#type-checking, accessed: 2014-06-30

[10] Pašalić, E., Taha, W., Sheard, T.: Tagless staged interpreters for typed languages. In: Proceedings of the Seventh ACM SIGPLAN International Conference on

Functional Programming. pp. 218–229. ICFP ’02, ACM (2002)

[11] Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming. pp. 50–61. ICFP ’06 (2006)

[12] Swierstra, W.: Data types à la carte. Journal of Functional Programming 18, 423–436 (6 2008)