

Draft
A Typical Synergy
Dynamic Types and Generalised Algebraic Datatypes

Thomas van Noort and Peter Achten and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

`{thomas, p.achten, rinus}@cs.ru.nl`

Abstract. Static typing is not always sufficient in strongly typed functional programming languages since some types will only be known at run-time. Especially systems that require user input have to deal with this issue. Dynamic types provide a clean approach to integrating dynamic typing in a statically typed language. Dynamic systems, such as structure editors, transform complex internal structures which represent complex domain-specific languages. These are modeled using generalised algebraic datatypes which capture additional type information, leading to precise definitions. However, the interaction between polymorphic values wrapped in dynamics and polymorphic constructors of generalised algebraic datatypes is often cumbersome since these concepts live in separate worlds. In this paper we motivate the need for a typical synergy between these two concepts and sketch its semantics.

1 Introduction

Types play an important role in strongly typed functional programming languages such as Clean and Haskell. Using static type-checking, erroneous behaviour at run-time is prevented. Moreover, more efficient code can be generated using the knowledge provided by the types at compile-time.

However, in dynamic systems that deal with user input, some types will only be known at run-time. Using dynamic types (dynamics), monomorphic [1] and polymorphic [2] values can be wrapped in a black box. The dynamic is unwrapped by pattern matching on the required type in a function definition, instead of specifying the type explicitly in its signature. This approach defers part of the type-checking process until run-time, exactly when the final required type information is made available. Fortunately, this does not take place at the cost of the advantage of static typing since the type system guarantees that when pattern matching succeeds, the unwrapped dynamic can be used safely as dictated by the specified type. Of course, pattern matching can fail and cause a run-time error, but this is not different from conventional pattern matching.

Dynamic systems, such as structure editors, often update complex internal structures, based on the interaction with the user. Such structures are inductively defined using algebraic datatypes (ADTs). Unfortunately, ADTs restrict the result of all its constructors to be of the same type. With the arrival of generalised algebraic datatypes (GADTs) [5, 7, 9], this restriction is relieved by allowing constructors to explicitly specify their return types. Once such a constructor is pattern matched, the additional type information is made available at its use sites, allowing more specific functions to be defined.

The concepts of dynamics and GADTs have a striking similarity. Via pattern matching, the former consumes type information while the latter produces type information. Typically, structure editors require a relation between the information provided by these concepts. The user input is wrapped in a dynamic and is used to update the internal GADT representation. Unfortunately, the type information that needs to be related live in separate worlds. Especially in the case of polymorphic dynamics and polymorphic GADT constructors, enforcing such a relation is quite cumbersome since this requires a proof of type equality. Although both concepts have been studied in depth separately, a combination of the two has never been described before. The main contribution of this paper is to motivate and describe the typical synergy between dynamics and GADTs, allowing us to relate their polymorphic type information naturally.

Related Work

Related work will be included in the final version of this paper.

Overview

This paper is organized as follows. First, we discuss the preliminaries (Section 2) to become acquainted with the participants of the synergy. We motivate the need for the synergy by describing a structure editor (Section 3). Then, we sketch a semantics for the interaction between dynamics and GADTs (Section 4). We end this paper with a brief discussion and future work in the conclusion (Section 5).

2 Preliminaries

We start by briefly introducing dynamics (Section 2.1) and GADTs (Section 2.2).

2.1 Dynamic Types

The advantage of statically typed languages is that types are verified at compile-time, preventing erroneous behaviour at run-time due to ill-typed values. However, static types sometimes is not enough since a type might only be known at run-time. Using dynamic types, values are wrapped in a black box, not exposing the type of the contents to the outside world. But unlike existential types [6],

both the value and its type are unwrapped by pattern matching the black box, thereby obtaining a value of the matched content type.

In Clean, the keyword **dynamic** provides the mechanism to wrap values in a dynamic [8], obtaining a value of type *Dynamic*:

```
wrapInt :: Int → Dynamic
wrapInt x = dynamic x
```

Unwrapping the integer value is achieved by pattern matching on the dynamic value using the type annotation ::, thereby providing a required type:

```
unwrapInt :: Dynamic → Int
unwrapInt (x :: Int) = x
unwrapInt _          = 0
```

The first arm of the function pattern matches on a value *x* of type *Int* in the dynamic. As with regular pattern matching, the dynamic pattern match can fail in case the wrapped value is not of the required type *Int*. It is our responsibility to provide a catch-all arm which either returns a default value or a run-time error message.

Instead of defining a function for each value type that is turned into a dynamic, we define a single function:

```
wrap :: α → Dynamic | TC α
wrap x = dynamic x
```

Since the value of type α is wrapped in a dynamic, we require the type code (i.e., the value representation of the type) of α using the built-in *TC* class constraint. This type representation is silently wrapped together with the value in the dynamic. Unlike Haskell, Clean supports polymorphic dynamics, which allows us to use type variables in the dynamic pattern match:

```
unwrap :: Dynamic → α | TC α
unwrap (x :: α^) = x
```

We require *x* to be of type α and refer to the same type variable in the result type of *unwrap* using the annotation [^]. This causes both types to be coerced automatically at run-time. Therefore, a type code is required for α such that it can be compared with the type code obtained from the dynamic pattern match. To further illustrate the power of polymorphic dynamics, we define function application of dynamics¹:

```
apply :: Dynamic → Dynamic → Maybe β | TC β
apply (f :: α → β^) (x :: α) = Just (f x)
apply _ _ = Nothing
```

¹ For the sake of clarity, we curry the Clean types just as in Haskell.

The dynamic pattern matches in the first arm of *apply* share the same scope. Therefore, they only succeed once the argument type of the function matches the type of the argument. Because the result type of the function in the first dynamic pattern match refers to β in the result type of *apply*, a type code is required for this type.

2.2 Generalised Algebraic Datatypes

Algebraic datatypes are an oft-used abstraction in functional languages since they provide an inductive approach to defining complex structures by enumerating the alternatives of a type and the associated fields. For example, in Haskell, an ADT representing a term in the λ -calculus is defined as follows:

```
data Lam = Undef
         | Const Value
         | App Lam Lam
```

The *Undef* constructor is empty, while the *Const* constructor has a single field for a value. The *App* constructor has two fields, which both can be any lambda term. The values are enumerated by another ADT:

```
data Value = VInt Int
           | VBool Bool
           | VFun (Value  $\rightarrow$  Value)
```

Next, we define an evaluation function:

```
eval :: Lam  $\rightarrow$  Value
eval Undef           =  $\perp$ 
eval (Const x)      = x
eval (App (Const (VFun f)) x) = f (eval x)
eval (App _ _       ) = error "eval.App: not a function"
```

While the arms for *Undef* and *Const* are straightforward, the arm for *App* has to ensure that its first field is actually a function. Even worse, nothing prevents us from constructing ill-typed terms. Ideally, we would like to expose the result type of a term using a parameter. Unfortunately, the constructors of an ADT share the same result type while the constructors of *Lam* ideally do not.

With the arrival of generalised abstract datatypes, we are freed from this restriction by allowing us to provide an explicit type signature to each constructor. We illustrate the use of GADTs by defining the *Lam* type again:

```
data Lam :: *  $\rightarrow$  * where
  Undef :: Lam  $\alpha$ 
  Const ::  $\alpha \rightarrow$  Lam  $\alpha$ 
  App   :: Lam ( $\alpha \rightarrow \beta$ )  $\rightarrow$  Lam  $\alpha \rightarrow$  Lam  $\beta$ 
```

The *Lam* type is parameterized by the result type of the term once it is evaluated. With each constructor, we explicitly specify its result type. The *Undef* constructor represents an undefined value. Since its result type α is free and not bound by any fields, it can be unified with any other type. The *Const* constructor lifts any value to the *Lam* type. The *App* constructor is more explicit about the types of its two field. The argument type of the function term must match the type of the argument term. Then, its result type is the result type of the function term.

Because of the use of a GADT, we are prevented from constructing ill-typed terms. Consider the following example:

$\perp 1$ is expressed as `App Undef (Const 1)`

Since the return type of the *Undef* constructor can be anything, it is instantiated to a function as *App* requires, thereby returning a value of type *Lam* β . When we provide a term that does not return a function, the term becomes ill-typed:

`0 1` is expressed as `App (Const 0) (Const 1)`

A more useful example is the following:

`($\lambda x \rightarrow (x, x)$) 1` is expressed as `App (Const ($\lambda x \rightarrow (x, x)$)) (Const 1)`

This term is well-typed and returns a value of type *Lam* (*Int*, *Int*).

The idea behind GADTs is that the additional type information described in the result type of the constructors can be employed when the constructors are pattern matched in a function definition. Since only well-typed terms can be constructed, we can now safely define the evaluation function:

$$\begin{aligned} \text{eval} &:: \text{Lam } \alpha \rightarrow \alpha \\ \text{eval } \text{Undef} &= \perp \\ \text{eval } (\text{Const } x) &= x \\ \text{eval } (\text{App } f \ x) &= \text{eval } f \ (\text{eval } x) \end{aligned}$$

Notice how the type of the result changes with each arm. Each constructor specializes α to the type dictated by its result type.

3 Motivation

In the previous section we have seen two, at first sight seemingly different, concepts: Clean's dynamics and Haskell's GADTs. However, these concepts have more in common than you might think. On the one hand we discussed how dynamics are unwrapped in a dynamic pattern match, *consuming* type information at its use site. On the other hand, we argued that GADTs allow for more specific result types of constructors in a datatype, which is employed in a GADT pattern match, *producing* type information at its use site.

In this section, we motivate the need for a synergy between these complementary concepts. We start by describing the context in which this all takes place and the problems encountered (Section 3.1). Next, we discuss why the conventional approach is not suited to this problem (Section 3.2) and how the synergy elegantly improves on these issues (Section 3.3).

3.1 Context

Structure editors allow users to edit values based on their structure, thereby preventing ill-structured (i.e., ill-typed) values. One of the approaches allows the user to request a value of a certain type from the editor, after which a template value is provided containing default values. Then, the user proceeds by updating the structure at each desired position. Since these systems rely on user input, the types of the values that are updated are not known until run-time. Therefore, updates are stored in dynamics and provided to the update function as such. The structures that are edited are often complex and require a strongly typed representation. As we have seen in Section 2.2, ADTs are less suited than GADTs to this job. To elegantly define the update function, a natural interaction between dynamics and GADTs is required.

As a running example, we use the GADT definition from Section 2.2 that expresses the λ -calculus. Our goal is to define an update function that takes such a term, and updates a polymorphic value at a specified position with a new value. The target of the update is specified by a path:

```
type Path = [Int]
```

The path is represented by a list of integers. The length of the list indicates the recursive level (where 0 is the root) of the target and each value the fields (where 0 is the first field) that are recursed.

3.2 Conventional Approach

Unfortunately, Haskell does not support polymorphic dynamics. Therefore, we have to resort to ordinary type variables to express that the new value is provided by the user and can be of any type. This gives us the following type signature for the update function:

```
update :: Lam  $\alpha$   $\rightarrow$  Id  $\rightarrow$   $\beta$   $\rightarrow$  Lam  $\alpha$ 
```

The new value of type β is not restricted by any of the argument types or the result type. Unfortunately, this relaxation bites us in the tail once we update the value at the right position, since the old value type α and the new value type β are considered different types and the function will be ill-typed.

This is where equality types come in [3, 4]. By making clever use of GADTs, we are able to construct a proof of type equality. First, we define a value representation of types:

```

data Rep :: * → * where
  RInt  :: Rep Int
  RBool :: Rep Bool
  RFun  :: Rep α → Rep β → Rep (α → β)

```

The *Rep* type is only a witness of a type, meaning, the *RInt* constructor only carries the fact that it is of type *Int*. Given such witnesses, we are able to construct the actual proof that the types of such *Rep* values are the same. Such a proof is constructed by the following GADT:

```

data Equal :: * → * → * where
  Refl :: Equal α α

```

The *Equal* type consists of a single constructor *Refl*, one that proves that both of the type arguments are the same. Then, we define a type equality function:

```

eqRep :: Rep α → Rep β → Maybe (Equal α β)
eqRep RInt      RInt      = Just Refl
eqRep RBool    RBool    = Just Refl
eqRep (RFun x1 y1) (RFun x2 y2) = case (eqRep x1 x2, eqRep y1 y2) of
                                     (Just Refl, Just Refl) → Just Refl
                                     _                    → Nothing
eqRep _         _         = Nothing

```

Given two *Rep* values, this function either returns *Just Refl* if the values are the same, thereby implicitly indicating that the types α and β are the same as well, or *Nothing*. In the arm for *RFun*, both the argument and result types need to be equal. Finally, we define a catch-all arm which returns *Nothing* for values that are not equal.

Using this type equality function we are one step closer to defining our update function. The conventional approach forces us to include a *Rep* value for each value that we would like update. This requires us to modify our original *Lam* definition to the following:

```

data LamR :: * → * where
  UndefR :: LamR α
  ConstR :: (α, Rep α) → LamR α
  AppR    :: LamR (α → β) → LamR α → LamR β

```

In this case, only the *ConstR* stores values which can be updated, but the impact on larger and more complex datatypes is easily imagined. Using the modified datatype, we are finally able to define our update function:

```

update :: LamR α → Path → (β, Rep β) → LamR α
update UndefR      []      _      = UndefR
update (ConstR (x, rx)) [0] (y, ry) = case eqRep rx ry of
                                     Just Refl → ConstR (y, ry)
                                     Nothing   → ConstR (x, rx)

```

$$\begin{array}{lll}
\text{update } (\text{AppR } f \ x) & (0 : p) \ y & = \text{AppR } (\text{update } f \ p \ y) \ x \\
\text{update } (\text{AppR } f \ x) & (1 : p) \ y & = \text{AppR } f \ (\text{update } x \ p \ y) \\
\text{update } x & - & = x
\end{array}$$

In the arm for *UndefR* there is nothing left to do, we only have to make sure that the path is fully consumed. The *ConstR* is the interesting case, since we have to verify that the types match by testing the equality of the *Rep* values. Once these values are the same, we provide a proof that α and β are equal types by pattern matching on the *Refl* constructor. The arm for *AppR* dispatches on the head of the path, either recursing in its first or second field. Finally, a catch-all case is included to return the original term once the provided path is incorrect.

Although this approach works, it is not very elegant. Because of the *Rep* values, the original datatype needs to be adapted, introducing yet another datatype. Moreover, the types of the values that are updated have to be known beforehand since these have to be constructors in the *Rep* type. Furthermore, the update function becomes cluttered with type equality witnesses, and above all, this approach does not scale up to more complex structures and update functions.

3.3 The Synergy

The conventional approach requires us to carry around value representations of types which are used to convince the type checker of polymorphic type equality at the right time. When we look back at Section 2.1, we notice that this is actually what Clean's polymorphic dynamics provide. Values are wrapped together with a type code, which can be unwrapped at the right time using a dynamic pattern match. Therefore, we propose to combine Clean's dynamics with Haskell's GADTs to elegantly define the update function:

$$\begin{array}{lll}
\text{update} :: \text{Lam } \alpha \rightarrow \text{Path} \rightarrow \text{Dynamic} \rightarrow \text{Lam } \alpha & & \\
\text{update } \text{Undef} & [] & - = \text{Undef} \\
\text{update } (\text{Const } (x ::^+ \alpha)) & [0] & (y ::^- \alpha) = \text{Const } y \\
\text{update } (\text{App } f \ x) & (0 : p) \ y & = \text{App } (\text{update } f \ p \ y) \ x \\
\text{update } (\text{App } f \ x) & (1 : p) \ y & = \text{App } f \ (\text{update } x \ p \ y) \\
\text{update } x & - & - = x
\end{array}$$

Let us take a look at the differences between this update function and the conventional definition from Section 3.2. First of all, this definition uses the original datatype *Lam* instead of the adapted *LamR* type and it takes a dynamic as a new value. Furthermore, the arm for *Const* is freed from the verbose type equality witnesses. Instead, a new GADT constructor annotation $::^+$ is used to denote that the constructor produces information that value x is of type α . Its counterpart, the dynamic annotation $::^-$, is used to denote that it consumes the type α of value y . Since these type annotations share the same scope, this leads to type equality and y can safely be updated by x . We introduce new type annotations to disambiguate Clean's dynamic pattern match annotation from Haskell's ordinary type annotation. Note that the catch-all case now also takes care of any failing dynamic pattern match.

4 Semantics

In this section we give a brief sketch the semantics of the synergy. We discuss the challenges in typing the GADT constructor annotation and dynamic annotation (Section 4.1) and its operational semantics (Section 4.2). For the moment, we do not consider the annotation \wedge from Section 2.1. A semantics based on first-class phantom types [5] will be included in the final version of this paper.

4.1 Typing

The driving force behind the synergy is that GADT constructor annotations produce type information which is consumed by dynamic annotations. We discuss what types can be provided as annotations and how multiple type annotations interact via scoping.

Types The types that we allow in the type annotations are either a primitive type, polymorphic type variable, type application, or function type. For the sake of simplicity, we do not allow existentials, higher-ranked types, GADTs, or any other exotic type extension in a type annotation.

Furthermore, there must be a unification for the types in the GADT constructor annotations such that it adheres to the types as specified in the definition of the GADT. For example, the *Const* constructor of type *Lam* from Section 2.2 has the following type:

$$\text{Const} :: \alpha \rightarrow \text{Lam } \alpha$$

Next, consider the following pattern:

$$f (\text{Const } (x ::^+ \text{String})) \dots = \dots$$

This type annotation suggests that we only successfully match values of x of type *String*. However, we only use the GADT constructor annotation to name the type information that is produced by the pattern match. Since there is no substitution that applied to *String* is equal to α , this annotation is not allowed.

Scoping All the type annotations in the same arm of a function share the same scope. This allows us to enforce type equality between GADTs and dynamics. However, GADT constructor annotations are not allowed to share type variables between each other. Consider the following patterns:

$$f (\text{Const } (x ::^+ \alpha)) (\text{Const } (y ::^+ \alpha)) \dots = \dots$$

This definition suggests that type equality is enforced between two GADT values. However, we only use this annotation to name the type information that is produced. This in contrast to the dynamic annotation where each is allowed to restrict the pattern match freely, since it is only type information that is consumed in such annotations. For the sake of simplicity we do not allow any of the type annotations to be arbitrarily nested.

4.2 Operational Semantics

In Section 2.1 we have seen how the use of dynamics sometimes requires a type code (i.e., the built-in *TC* class). This is due to the fact that at run-time, the provided type code from the context is compared with the type code stored in the dynamic. In our approach, every GADT constructor annotation possibly introduces such a requirement since this information influences the dynamic annotation. Since each arm can contain different annotations, so will the type code requirement differ for each arm.

Because the type of a constructor value is not uniform in a GADT, we cannot statically determine which specific type codes are required. Therefore, we propose to use a solution that is influenced by the conventional approach from Section 3.2, every field of a GADT constructor is decorated with its corresponding type code. When a GADT value is constructed, the types of the fields are known and the right type code can be inserted. Then, when we traverse a GADT, the type code is traversed in parallel and the right information is there when we need it. We realize that this is likely an inefficient solution, and we will study more complex analyses in the future.

5 Conclusion

In this paper we have motivated the need for a synergy between dynamics and GADTs. We introduced two new type annotations: naming type information that is produced by pattern matching a GADT constructor and constraining a dynamic pattern match using this information. Using the example of update functions in structure editors, we have shown that this approach frees us from maintaining a type representation administration. The GADT representation that is updated need not to be adapted and the update function itself is no longer cluttered with type equality proofs.

Acknowledgements

This work has been funded by the Technology Foundation STW through its project on "Demand Driven Workflow Systems" (07729).

References

- [1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.
- [2] Martín Abadi, Luca Cardelli, Benjamin Pierce, Didier Rémy, and Robert Taylor. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):81–110, 1994.
- [3] Arthur Baars and Doaitse Swierstra. Typing dynamic typing. In Simon Peyton Jones, editor, *Proceedings of the 7th International Conference on Functional Programming, ICFP '02*, pages 157–166. ACM, 2002.

- [4] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Proceedings of the 6th Haskell Workshop, Haskell '02*, pages 90–104. ACM, 2002.
- [5] James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, 2003.
- [6] Konstantin Läfer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.
- [7] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In Julia Lawall, editor, *Proceedings of the 11th International Conference on Functional Programming, ICFP '06*, pages 50–61. ACM, 2006.
- [8] Marco Pil. Dynamic types and type dependent functions. In Pieter Koopman and Chris Clack, editors, *Selected Papers of the 10th International Symposium on the Implementation of Functional Languages, IFL '99*, pages 169–185. Springer-Verlag, 1999.
- [9] Hongwei Xi, Chiyang Chen, and Gang Chen. Guarded recursive datatype constructors. In Greg Morrisett and Alex Aiken, editors, *Proceedings of the 30th Symposium on Principles of Programming Languages, POPL '03*, pages 224–235. ACM, 2003.