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. We present a typical synergy between dynamic types (dynamics) and generalised algebraic datatypes (GADTs). The former provides a clean approach to integrating dynamic typing in a statically typed language. It allows values to be wrapped together with their type in a uniform package, deferring type unification until run time using a pattern match annotated with the desired type. The latter allows for the explicit specification of constructor types, as to enforce their structural validity. In contrast to ADTs, GADTs are heterogeneous structures since each constructor type is implicitly universally quantified. Unfortunately, pattern matching only enforces structural validity and does not provide instantiation information on polymorphic types. Consequently, functions that manipulate such values, such as a type-safe update function, are cumbersome due to boilerplate type representation administration. In this paper we focus on improving such functions by providing a new GADT annotation via a natural synergy with dynamics. We formally define the semantics of the annotation and touch on novel other applications of this technique such as type dispatching and enforcing type equality invariants on GADT values.

1 Introduction

In this paper we discuss a typical synergy between two concepts: dynamic types (dynamics) and generalised algebraic datatypes (GADTs).

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 dynamics, monomorphic [1] and polymorphic [2] values can be wrapped together with their type 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.

Algebraic datatypes (ADTs) in functional languages allow us to inductively define structures. Unfortunately, it does not allow us to enforce structural validity at compile time. With the arrival of generalised algebraic datatypes (GADTs) [6, 10, 14], this restriction is relieved by allowing constructors to explicitly dictate their types. On the one hand, this prevents us from constructing ill-structured (i.e., ill-typed) values, and on the other hand this ensures structural validity once a GADT value is pattern matched. In contrast to ADTs, GADTs are heterogeneous structures since each constructor occurrence is implicitly universally quantified. Pattern matching such a value only introduces information regarding the structure of the constructor types, leaving the type variables polymorphic. However, more information on their instantiation is often required, typically in functions that manipulate such values. Conventional approaches to this problem are cumbersome, due to boilerplate type representation administration.

The main contribution of this paper is to define a type-safe update function on GADT values via an annotation, achieved by a natural synergy between dynamics and GADTs.

Overview

This paper is organised as follows. First, we elaborate on both dynamics and GADTs (Section 2). We motivate the need for the synergy by defining an update function on λ-terms (Section 3). Then, we formally define a semantics for the new annotation via a synergy between dynamics and GADTs (Section 4). We conclude with related work (Section 5) and a discussion on future work and other applications of this technique (Section 6). In this paper we use Clean's dynamics and Haskell's GADTs. For the sake of presentation, our examples use Haskell syntax, augmented with Clean's notation for dynamics.

2 Preliminaries

We start by 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 typing sometimes does not suffice 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 [9], 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 together with their type in a dynamic [13], obtaining a value of type *Dynamic*:

$$
\begin{aligned}
&\mathit{wrapInt} :: \mathit{Int} \to \mathit{Dynamic} \\
&\mathit{wrapInt}\ x = \mathbf{dynamic}\ x
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{unwrapInt} :: \mathit{Dynamic} \to \mathit{Int} \\
&\mathit{unwrapInt}\ (x :: \mathit{Int}) \quad\ = x \\
&\mathit{unwrapInt}\ (x :: \mathit{String}) = \mathit{stringToInt}\ x \\
&\mathit{unwrapInt}\ \_ \qquad\qquad\ = 0
\end{aligned}
$$

The first arm of the function pattern matches on a value x of type *Int* in the dynamic. If this is the case, the value is returned unchanged. However, the value in the dynamic is possibly a string and has to be converted to an integer first. Due to run-time type unification, the dynamic pattern match can fail in case the wrapped value is not of the type *Int* or *String*. 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:

$$
\begin{aligned}
&\mathit{wrap} :: \mathit{TC}\ \alpha \Rightarrow \alpha \to \mathit{Dynamic} \\
&\mathit{wrap}\ x = \mathbf{dynamic}\ x
\end{aligned}
$$

Since this function is polymorphic in the argument type, we require the context to provide the type code (i.e., the value representation of the type) of α which is stored together with the value x, using Clean's built-in TC class constraint. A type code also contains the definition of the type it describes, because dynamics can be (de)serialised across modules and verifying name equivalence in a dynamic pattern match does not suffice. Consequently, TC instances are only available for nonabstract types.

Unlike Haskell, Clean supports type-dependent dynamics [11], which allows us to use pattern variables in the type of a dynamic pattern match:

$$
\begin{aligned}
&\mathit{unwrap} :: \mathit{TC}\ \alpha \Rightarrow \mathit{Dynamic} \to \alpha \\
&\mathit{unwrap}\ (x :: \alpha^{\wedge}) = x \\
&\mathit{unwrap}\ \_ \qquad\ = \mathit{error}\ \texttt{"unwrap: incorrect type"}
\end{aligned}
$$

We require x to be of type α and refer to the same variable in the result type of *unwrap* using the $^{\wedge}$ 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. The context in which this function is used determines which type code is provided.

Pattern variables can also be used to enforce type equality, for example, to define function application of dynamics:

$$
\begin{aligned}
&\mathit{apply} :: \mathit{TC}\ \alpha \Rightarrow \mathit{Dynamic} \to \mathit{Dynamic} \to \mathit{Maybe}\ \alpha \\
&\mathit{apply}\ (f :: \beta \to \alpha^{\wedge})\ (x :: \beta) = \mathit{Just}\ (f\ x) \\
&\mathit{apply}\ \_\qquad\qquad\ \_\qquad\ = \mathit{Nothing}
\end{aligned}
$$

The dynamic pattern matches in the first arm 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 α in the result type of *apply*, a type code is required for this type. As an example, consider the following expressions:

$$
\begin{aligned}
&\mathit{apply}\ (\mathbf{dynamic}\ \mathit{fst})\ (\mathbf{dynamic}\ (1, \texttt{"2"})) \quad \rightsquigarrow \quad \mathit{Just}\ 1 \\
&\mathit{apply}\ (\mathbf{dynamic}\ \mathit{fst})\ (\mathbf{dynamic}\ 1) \qquad\quad \rightsquigarrow \quad \mathit{Nothing}
\end{aligned}
$$

While the first expression succeeds, the second expression fails since the argument is not a pair. Finally, dynamics preserve lazy behaviour of functional programs:

$$
\mathit{apply}\ (\mathbf{dynamic}\ \mathit{fst})\ (\mathbf{dynamic}\ (1, \bot)) \quad \rightsquigarrow \quad \mathit{Just}\ 1
$$

Although the value $\bot$ is part of the tuple that is wrapped in a dynamic, it is not evaluated when (un)wrapped.

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 λ-terms could be defined as follows:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Lam} = \mathit{Undef} \\
&\qquad\qquad \mid\ \mathit{Const}\ \mathit{Value} \\
&\qquad\qquad \mid\ \mathit{App}\ \mathit{Lam}\ \mathit{Lam}
\end{aligned}
$$

The *Undef* constructor has no fields, while the *Const* constructor has a single field for a value. The *App* constructor has two fields, which both can be any term. The values are enumerated by another ADT:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Value} = \mathit{VInt}\ \mathit{Int} \\
&\qquad\qquad \mid\ \mathit{VFun}\ (\mathit{Value} \to \mathit{Value})
\end{aligned}
$$

Next, we define an evaluation function:

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Lam} \to \mathit{Value} \\
&\mathit{eval}\ \mathit{Undef}\ \ \ = \bot \\
&\mathit{eval}\ (\mathit{Const}\ x) = x \\
&\mathit{eval}\ (\mathit{App}\ f\ x) = \mathbf{case}\ \mathit{eval}\ f\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{VFun}\ f \to f\ (\mathit{eval}\ x) \\
&\qquad\qquad\qquad\qquad \_\qquad\ \to \mathit{error}\ \texttt{"eval: not a function"}
\end{aligned}
$$

The arms for *Undef* and *Const* are straightforward. However, since nothing prevents us from constructing ill-typed terms, the arm for *App* has to ensure that its first field actually evaluates to a function.

With the arrival of generalised abstract datatypes, we are able to enforce structural validity by providing an explicit type signature to each constructor. Consequently, a GADT imposes a heterogeneous structure since all constructors are implicitly universally quantified. We illustrate the use of GADTs by defining the *Lam* type again, this time describing typed λ-terms:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Lam} :: \star \to \star\ \textbf{where} \\
\quad \mathit{Undef} :: \mathit{Lam}\ \alpha \\
\quad \mathit{Const} :: \alpha \to \mathit{Lam}\ \alpha \\
\quad \mathit{App} \quad :: \mathit{Lam}\ (\beta \to \alpha) \to \mathit{Lam}\ \beta \to \mathit{Lam}\ \alpha
\end{array}
$$

The *Lam* type is parameterised 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. The explicit constructor types prevent us from constructing ill-typed terms. Consider the following examples:

$$\bot\ 1 \quad \equiv \quad \mathit{App}\ \mathit{Undef}\ (\mathit{Const}\ 1)$$

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 $\mathit{Lam}\ \alpha$. When we provide a term that does not return a function, the term becomes ill typed:

$$0\ 1 \quad \equiv \quad \mathit{App}\ (\mathit{Const}\ 0)\ (\mathit{Const}\ 1)$$

A more useful example actually applies a function, for example the absolute value of an integer:

$$\mathit{abs}\ 1 \quad \equiv \quad \mathit{App}\ (\mathit{Const}\ \mathit{abs})\ (\mathit{Const}\ 1)$$

This term is well typed and returns a value of type $\mathit{Lam}\ \mathit{Int}$.

Type information described in the type of the constructors is also employed when the constructors are pattern matched in a function definition. Since only well-typed terms can be constructed, we can now safely and concisely define the evaluation function:

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Lam}\ \alpha \to \alpha \\
\mathit{eval}\ \mathit{Undef} \quad\ = \bot \\
\mathit{eval}\ (\mathit{Const}\ x) = x \\
\mathit{eval}\ (\mathit{App}\ f\ x) = \mathit{eval}\ f\ (\mathit{eval}\ x)
\end{array}
$$

The result type of the function depends on the term that is evaluated. Each constructor dictates the type of its fields as well as the result type. For example, evaluating the first field of the *App* constructors returns a function, which can safely be applied to its evaluated second field. However, be aware that the exact types of these fields are not known since we are dealing with a heterogeneous structure.

3 Motivation

In this section, we motivate the need for the typical synergy between dynamics and GADTs in the context of update functions on GADTs (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 using a new GADT annotation (Section 3.3).

3.1 Setting the Scene

As the running example, we use the definition from Section 2.2 that represents typed λ-terms. Our goal is to define an update function that takes such a term, and updates a field of a constructor at a specified position with a new value. Then, the desired type of the update function becomes:

$$update :: Lam\ \alpha \to Path \to \beta \to Lam\ \alpha$$

The argument and result type of the function are the same since we only consider updates that do not affect the top-level type of the term. However, an update can change the structure. The path depicts the location of the update in the heterogeneous structure:

$$\textbf{type}\ Path = [Int]$$

The path is represented as a list of integers. The length of the list indicates the recursive level (where the empty list is the root) of the target and each value the field (where 0 is the first field) that must be considered. Since the path possibly dictates an update anywhere in the heterogeneous structure, the type of the new value is unrestricted. Hence, the challenge we face lies in only allowing type-safe updates.

3.2 Conventional Approach

The conventional approach to this problem makes extensive use of equality types [3, 5]. By comparing the value representations of the type of the old and new value, a proof of type equality can be obtained to ensure only type-safe updates.

First, we modify our original *Lam* definition from Section 2.2 to the following:

$$
\begin{aligned}
&\textbf{data } \mathit{Lam}_R :: \star \to \star \textbf{ where}\\
&\quad \mathit{Undef}_R :: \mathit{Lam}_R\ \alpha\\
&\quad \mathit{Const}_R :: \mathit{RepOf}\ \alpha \to \mathit{Lam}_R\ \alpha\\
&\quad \mathit{App}_R \quad :: \mathit{RepOf}\ (\mathit{Lam}_R\ (\beta \to \alpha)) \to \mathit{RepOf}\ (\mathit{Lam}_R\ \beta) \to \mathit{Lam}_R\ \alpha
\end{aligned}
$$

The difference is that the types of the constructor fields now include a type representation:

$$
\textbf{type } \mathit{RepOf}\ \alpha = (\alpha, \mathit{Rep}\ \alpha)
$$

The *Rep* type enumerates the possible types, including the integer type, the function type, and the Lam_R type:

$$
\begin{aligned}
&\textbf{data } \mathit{Rep} :: \star \to \star \textbf{ where}\\
&\quad \mathit{RInt} \quad :: \mathit{Rep}\ \mathit{Int}\\
&\quad \mathit{RFun} \quad :: \mathit{Rep}\ \alpha \to \mathit{Rep}\ \beta \to \mathit{Rep}\ (\alpha \to \beta)\\
&\quad \mathit{RLam}_R :: \mathit{Rep}\ \alpha \to \mathit{Rep}\ (\mathit{Lam}_R\ \alpha)
\end{aligned}
$$

The *Rep* type is only a witness of a type, for example, the type $\mathit{Lam}_R\ (\mathit{Int} \to \mathit{Int})$ is witnessed by the value $\mathit{RLam}_R\ (\mathit{RFun}\ \mathit{RInt}\ \mathit{RInt})$. For the sake of brevity, this representation type only reflects monomorphic types. 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:

$$
\begin{aligned}
&\textbf{data } \mathit{Equal} :: \star \to \star \to \star \textbf{ where}\\
&\quad \mathit{Refl} :: \mathit{Equal}\ \alpha\ \alpha
\end{aligned}
$$

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 that performs a point-wise comparison of type representations, using Haskell's **do** notation:

$$
\begin{aligned}
&\mathit{eq}_R :: \mathit{Rep}\ \alpha \to \mathit{Rep}\ \beta \to \mathit{Maybe}\ (\mathit{Equal}\ \alpha\ \beta)\\
&\mathit{eq}_R\ \mathit{RInt}\ \mathit{RInt} = \mathit{Just}\ \mathit{Refl}\\
&\mathit{eq}_R\ (\mathit{RFun}\ x_1\ x_2)\ (\mathit{RFun}\ y_1\ y_2) = \textbf{do}\ \mathit{Refl} \leftarrow \mathit{eq}_R\ x_1\ y_1\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathit{Refl} \leftarrow \mathit{eq}_R\ x_2\ y_2\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ \mathit{Refl}\\
&\mathit{eq}_R\ (\mathit{RLam}_R\ x)\ \ (\mathit{RLam}_R\ y) = \textbf{do}\ \mathit{Refl} \leftarrow \mathit{eq}_R\ x\ y\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ \mathit{Refl}\\
&\mathit{eq}_R\ \_\ \_ = \mathit{Nothing}
\end{aligned}
$$

Given two *Rep* values, this function either returns *Just Refl* if the type representations are the same, thereby implicitly indicating that the types α and β are the same as well, or *Nothing*. In the arms for *RFun* and RLam_R we have to explicitly pattern match the result of the recursion as to obtain its type equality proof. Finally, we define a catch-all arm which returns *Nothing* for *Rep* values that are not equal.

Then, using the modified *Lam* definition and a type representation added to the new value, we are finally able to define our update function:

$$
\begin{array}{lll}
update_R :: Lam_R\ \alpha \to Path \to RepOf\ \beta \to Lam_R\ \alpha & & \\
update_R\ Undef_R & [\,] \quad \_ & = Undef_R \\
update_R\ (Const_R\ (x, rx))\ [0] & (y, ry) & = \textbf{case}\ eq_R\ rx\ ry\ \textbf{of} \\
 & & \quad Just\ Refl \to Const_R\ (y, ry) \\
 & & \quad Nothing \to Const_R\ (x, rx) \\
update_R\ (App_R\ (f, rf)\ x)\ [0] & (y, ry) & = \textbf{case}\ eq_R\ rf\ ry\ \textbf{of} \\
 & & \quad Just\ Refl \to App_R\ (y, ry)\ x \\
 & & \quad Nothing \to App_R\ (f, rf)\ x \\
update_R\ (App_R\ f\ (x, rx))\ [1] & (y, ry) & = \textbf{case}\ eq_R\ rx\ ry\ \textbf{of} \\
 & & \quad Just\ Refl \to App_R\ f\ (y, ry) \\
 & & \quad Nothing \to App_R\ f\ (x, rx) \\
update_R\ (App_R\ (f, rf)\ x)\ (0 : p)\ y & & = App_R\ (update_R\ f\ p\ y, rf)\ x \\
update_R\ (App_R\ f\ (x, rx))\ (1 : p)\ y & & = App_R\ f\ (update_R\ x\ p\ y, rx) \\
update_R\ x \quad \_ \quad \_ & & = x
\end{array}
$$

In the arm for $Undef_R$ there is nothing left to do, we only have to make sure that the path is fully consumed. The $Const_R$ is the first 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. Then, in the arms for App_R we use the same approach and either replace its first or second field, or dispatch on the head of the path and continue to recurse in either of its fields. Finally, a catch-all arm is included to return the original term once the provided path is incorrect. Whenever the function is applied, all the type representations need to be provided explicitly:

$$
\begin{array}{l}
update_R\ (Const_R\ (abs, RFun\ RInt\ RInt))\ [0]\ (neg, RFun\ RInt\ RInt) \\
\quad \rightsquigarrow \\
Const_R\ (neg, RFun\ RInt\ RInt)
\end{array}
$$

Although this approach guarantees type-safe updates, it is not a very elegant definition. First of all, the invasive inclusion of *Rep* values in the datatype clutters the update function with type equality witnesses and manual proofs. Moreover, the types of the values that are updated have to be known beforehand since these are enumerated in the *Rep* type and traversed in the type equality function. 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 type representations which are used to convince the type checker of type equality. When we look back at Section 2.1, we notice that this is actually what Clean's *TC* type class provides. We propose to adapt the original *Lam* definition from Section 2.2 again:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Lam}_T :: \star \to \star\ \textbf{where} \\
\quad \mathit{Undef}_T :: \qquad\qquad\quad\ \ \mathit{Lam}_T\ \alpha \\
\quad \mathit{Const}_T :: \mathit{TC}\ \alpha \qquad\ \Rightarrow \alpha \to \mathit{Lam}_T\ \alpha \\
\quad \mathit{App}_T \quad :: (\mathit{TC}\ \beta, \mathit{TC}\ \alpha) \Rightarrow \mathit{Lam}_T\ (\beta \to \alpha) \to \mathit{Lam}_T\ \beta \to \mathit{Lam}_T\ \alpha
\end{array}
$$

Instead of including *Rep* values, we include TC class constraints with the constructors that can be updated. Then, we define the update function using the new $::^{\mathcal{G}}$ annotation on the field of a GADT constructor:

$$
\begin{array}{lll}
\mathit{update}_T :: \mathit{TC}\ \beta \Rightarrow \mathit{Lam}_T\ \alpha \to \mathit{Path} \to \beta \to \mathit{Lam}_T\ \alpha & & \\
\mathit{update}_T\ \mathit{Undef}_T & [\,] & \_ = \mathit{Undef}_T \\
\mathit{update}_T\ (\mathit{Const}_T\ (x ::^{\mathcal{G}} \beta\hat{}))\ [0] & & y = \mathit{Const}_T\ y \\
\mathit{update}_T\ (\mathit{App}_T\ (f ::^{\mathcal{G}} \beta\hat{})\ x)\ [0] & & y = \mathit{App}_T\ y\ x \\
\mathit{update}_T\ (\mathit{App}_T\ f\ (x ::^{\mathcal{G}} \beta\hat{}))\ [1] & & y = \mathit{App}_T\ f\ y \\
\mathit{update}_T\ (\mathit{App}_T\ f\ x) & (0 : p) & y = \mathit{App}_T\ (\mathit{update}_T\ f\ p\ y)\ x \\
\mathit{update}_T\ (\mathit{App}_T\ f\ x) & (1 : p) & y = \mathit{App}_T\ f\ (\mathit{update}_T\ x\ p\ y) \\
\mathit{update}_T\ x & \_ & \_ = x
\end{array}
$$

Let us take a look at the differences between this update function and the conventional definition update_R from Section 3.2. First of all, this function operates on the Lam_T type that is decorated with TC constraints, and its type contains a TC constraint to obtain a type code for the new value of type β. Although the update function was intended to be polymorphic at first, this constraint only forbids abstract types to occur as new values, as discussed earlier in Section 2.1. Another difference is that the function is no longer cluttered with verbose type equality witnesses and manual proofs. Instead, the fields of the constructors are annotated using the $::^{\mathcal{G}}$ annotation, accessing the instantiated polymorphic type information. For example, in the arm for Const_T, the annotation denotes that x is of type β, or even more specific, the type of the new value as determined by the context in which this function is used. Note that the catch-all arm now also takes care of any failing tests for type equality. Comparing the use of this update function to the conventional approach emphasises the elegance of our approach:

$$
\mathit{update}_T\ (\mathit{Const}_T\ \mathit{abs})\ [0]\ \mathit{neg} \quad \rightsquigarrow \quad \mathit{Const}_T\ \mathit{neg}
$$

Instead of explicitly providing type representations and equality proofs, it is now the context that implicitly determines which fields are eligible for an update.

4 Semantics

In this section we present the formal semantics of the synergy. We formally define a core functional language and the GADT annotation extension (Section 4.1). Then, we describe the idea behind the translation from the extended language to the core language by means of an example (Section 4.2), followed by a formal approach (Section 4.3).

$$
\begin{array}{lll}
(\textit{program}) & \pi & ::= \overline{\delta}\ \overline{\phi} \\
\\
(\textit{datatype declaration}) & \delta & ::= \mathbf{type}\ T\ \overline{\alpha} = \tau \\
 & & \mid\ \mathbf{data}\ T\ \overline{\alpha} = \overline{C\ \tau} \\
 & & \mid\ \mathbf{data}\ T :: \kappa\ \mathbf{where}\ \overline{C :: \sigma} \\
(\textit{qualified type}) & \sigma & ::= \overline{TC\ \alpha} \Rightarrow \tau \\
(\textit{base type}) & \tau & ::= \alpha \mid \mathit{Int} \mid T \\
 & & \mid\ \tau_1\ \tau_2 \mid \tau_1 \rightarrow \tau_2 \\
 & & \mid\ \mathit{Dynamic} \\
(\textit{annotation type}) & \omega & ::= \alpha \mid \alpha^{\wedge} \mid \mathit{Int} \mid T \\
 & & \mid\ \omega_1\ \omega_2 \mid \omega_1 \rightarrow \omega_2 \\
 & & \mid\ \mathit{Dynamic} \\
(\textit{kind}) & \kappa & ::= \star \mid \kappa_1 \rightarrow \kappa_2 \\
\\
(\textit{function declaration}) & \phi & ::= \mathbf{fix}\ f :: \sigma = \epsilon \\
(\textit{expression}) & \epsilon & ::= \bot \mid i \mid x \mid C \\
 & & \mid\ \epsilon_1\ \epsilon_2 \mid \lambda x \rightarrow \epsilon \mid \mathbf{case}\ \epsilon_s\ \mathbf{of}\ \overline{\rho \rightarrow \epsilon} \\
 & & \mid\ \mathbf{dynamic}\ e :: \omega \\
(\textit{nested pattern}) & \rho & ::= \varrho \mid C\ \overline{\rho} \\
(\textit{base pattern}) & \varrho & ::= \_ \mid i \mid x \\
 & & \mid\ x :: \omega
\end{array}
$$

Fig. 1. The core language FC

4.1 Formal Language

The functional core language FC, which forms the basis of our semantics, is depicted in Fig. 1. It is a common subset of Clean and Haskell, extended with Clean's dynamics and Haskell's GADTs. An FC program consists of zero or more datatype declarations and function declarations. A datatype is either a type synonym, an ADT, or a GADT. A type comes in three flavours: a qualified type, a base type, and an annotation type. A qualified type only includes the TC constraint, as to facilitate dynamics, where we write τ as a shorthand for the qualified type $\cdot \Rightarrow \tau$ with no constraints. Second, a base type comprises the polymorphic types. Very much alike a base type, we define a separate annotation type, but one that also allows the use of the $^{\wedge}$ annotation. A named function is defined by its type and body. Amongst the well-known expressions, our language supports the case construct to pattern match values, typically the arguments of a function, and dynamic values. In the language of patterns we distinguish a nested pattern from a base pattern, as to prepare for the language extension. Finally, we do not explicitly include lists and tuples of arbitrary arity in the language of expressions, patterns and types, since these can easily be realised through predefined ADTs. We do not provide operational semantics and typing for the core language since these have been studied in-depth elsewhere [2, 4, 6].

$$
\begin{array}{lll}
(\textit{expression}) & \epsilon ::= \cdots & \\
& \quad | \quad \cdots \mid \textbf{case}\ \epsilon_s\ \textbf{of}\ \overline{\theta \rightarrow \epsilon} & \\
& \quad | \quad \cdots & \\
(\textit{pattern}) & \theta ::= \varrho \mid C\ \overline{\vartheta} & \\
(\textit{field pattern}) & \vartheta ::= \rho \mid x ::^{\mathcal{G}} \omega &
\end{array}
$$

Fig. 2. The extended language FC$^+$

Next, we define the extended language FC$^+$ that allows us to use the GADT annotation, as shown in Fig. 2. For the sake of simplicity, we only allow the new annotation to occur on the top level of a constructor field pattern. However, nested patterns can be easily achieved by nesting case expressions. We redefine patterns in FC case expressions to be either a base pattern or a constructor with field patterns. Then, a pattern in a constructor field is either an original nested pattern, or an identifier annotated with a type.

As an example, we define the $update_T$ function from Section 3.3 in the FC$^+$ language:

$$
\begin{array}{l}
\textbf{fix}\ update_T :: TC\ \beta \Rightarrow Lam_T\ \alpha \rightarrow Path \rightarrow \beta \rightarrow Lam_T\ \alpha = \\
\quad \lambda x \rightarrow \lambda p \rightarrow \lambda y \rightarrow \textbf{case}\ (x, p)\ \textbf{of} \\
\quad\quad (Undef_T\ \ , [\,]) \qquad\quad \rightarrow Undef_T \\
\quad\quad (x \qquad\ \ , (0 : [\,])) \rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad Const_T\ (x ::^{\mathcal{G}} \beta^{\wedge}) \rightarrow Const_T\ y \\
\quad\quad\quad\quad\quad\quad\quad\quad App_T\ (f ::^{\mathcal{G}} \beta^{\wedge})\ x \rightarrow App_T\ y\ x \\
\quad\quad\quad\quad\quad\quad\quad\quad \_ \qquad\qquad\qquad\quad \rightarrow x \\
\quad\quad (x \qquad\ \ , (1 : [\,])) \rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad App_T\ f\ (x ::^{\mathcal{G}} \beta^{\wedge}) \rightarrow App_T\ f\ y \\
\quad\quad\quad\quad\quad\quad\quad\quad \_ \qquad\qquad\qquad\quad \rightarrow x \\
\quad\quad (App_T\ f\ x, (0 : p)) \rightarrow App_T\ (update_T\ f\ p\ y)\ x \\
\quad\quad (App_T\ f\ x, (1 : p)) \rightarrow App_T\ f\ (update_T\ x\ p\ y) \\
\quad\quad \_ \qquad\qquad\qquad\quad \rightarrow x
\end{array}
$$

While being slightly more verbose than the original definition, a translation from a Clean or Haskell definition is easily made. Note that the definitions of the *Path* and Lam_T type from Section 3.1 and Section 3.3 respectively do not change in the formal model.

4.2 Intuition

The general idea behind the translation is to take each GADT and translate it to an extended parallel definition in which only constructor fields that are annotated in the program, are decorated with additional type information. A conversion function takes care of inserting type information in the original definition and the $::^{\mathcal{G}}$ annotations are translated such that it accesses this information.

For example, the Lam_T type from Section 3.3 translates to the following:

$$\begin{array}{l}\textbf{data } Lam_T^\circ :: \star \to \star \textbf{ where} \\ \quad Undef_T^\circ :: \qquad\qquad\qquad\quad Lam_T^\circ\,\alpha \\ \quad Const_T^\circ :: TC\ \alpha \qquad\quad \Rightarrow TypeOf\ \alpha \to Lam_T^\circ\,\alpha \\ \quad App_T^\circ \quad :: (TC\ \beta, TC\ \alpha) \Rightarrow TypeOf\ (Lam_T\ (\beta \to \alpha)) \to TypeOf\ (Lam_T\ \beta) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \to Lam_T^\circ\,\alpha\end{array}$$

The extended definition, as well as its constructors, is given a new name. Since all fields of the constructors are annotated in the update function from Section 3.3, all fields of the $Const_T^\circ$ and App_T° constructor now contain a typed value. Note that in order to only have to translate patterns instead of complete functions later on, the addition of type information is nonrecursive:

$$\textbf{type } TypeOf\ \alpha = (\alpha, Dynamic)$$

A typed value is simply the original value paired with its type stored in a dynamic. As a *Dynamic* can contain a value of any type, not necessarily the type α, we use the following function to obtain correctness by construction:

$$\begin{array}{l}\textbf{fix } typeOf :: TC\ \alpha \Rightarrow \alpha \to TypeOf\ \alpha = \\ \quad \lambda x \to (x, \textbf{dynamic } \bot :: \alpha^\wedge)\end{array}$$

Since we only need the type of a value, it suffices to wrap $\bot$ instead of an actual value. As described in Section 2.1, the $^\wedge$ annotation refers to context-dependent type information. Meaning, the context in which *typeOf* is used determines the type that is stored in the dynamic. Note that constructors can contain GADT values, like App_T°, which requires such types to be stored in a dynamic. Unfortunately, type code facilities are yet to be defined for GADTs. Although GADTs greatly complicate the type inference process [10, 12], we hypothesise that storing such values in dynamics is not different from ADT values since it does not affect the unification of type codes which describe a GADT.

Then, the conversion from the original to the extended definition injects the type information in constant time using the function *typeOf*:

$$\begin{array}{l}\textbf{fix } toLam_T^\circ :: Lam_T\ \alpha \to Lam_T^\circ\,\alpha = \\ \quad \lambda x \to \textbf{case } x \textbf{ of} \\ \qquad Undef_T \quad \to Undef_T^\circ \\ \qquad Const_T\ x \to Const_T^\circ\,(typeOf\ x) \\ \qquad App_T\ f\ x \to App_T^\circ\,(typeOf\ f)\ (typeOf\ x)\end{array}$$

The conversion only renames the $Undef_T$ constructor since it has no fields. The fields of the $Const_T$ and App_T constructor are extended with their types. As the function *typeOf* dictates, this requires a type code for the field types. The translation relies critically on this assumption, which is enforced by only considering a FC$^+$ program well typed, if and only if, each constructor has TC constraints on every type variable occurring in its annotated fields. Fortunately, as mentioned before in Section 2.1, the TC constraint is easily discharged for

$$\boxed{[\![\pi_{\mathsf{FC}^+}]\!] \equiv \pi_{\mathsf{FC}}}$$

$$\frac{\overline{[\![\delta]\!]} \equiv \overline{\overline{\delta'}\ \phi^\circ} \qquad \overline{[\![\phi]\!]} \equiv \overline{\phi'}}{\overline{[\![\overline{\delta}\ \overline{\phi}]\!]} \equiv \overline{\overline{\delta'}\ \phi^\circ\ \overline{\phi'}}}\ (\textsc{t-prog})$$

Fig. 3. Translation of programs

any nonabstract type, which only forbids the use of the GADT annotation in combination with abstract types.

Finally, we define the translation of the actual $::^{\mathcal{G}}$ annotation, accessing the inserted type information. For example, the FC^+ function $update_T$, as defined in Section 4.1, is translated to FC:

$$\begin{array}{l}
\textbf{fix}\ update_T :: TC\ \beta \Rightarrow Lam_T\ \alpha \to Path \to \beta \to Lam_T\ \alpha = \\
\quad \lambda x \to \lambda p \to \lambda y \to \textbf{case}\ (x, p)\ \textbf{of} \\
\qquad \ldots \\
\qquad (x, (0 : [\,])) \to \textbf{case}\ toLam_T^\circ\, x\ \textbf{of} \\
\qquad\qquad\qquad Const_T^\circ\, (x, _ :: \beta^\wedge) \to Const_T\ y \\
\qquad\qquad\qquad App_T^\circ\, (f, _ :: \beta^\wedge)\ (x, _) \to App_T\ y\ x \\
\qquad\qquad\qquad _ \to x \\
\qquad (x, (1 : [\,])) \to \textbf{case}\ toLam_T^\circ\, x\ \textbf{of} \\
\qquad\qquad\qquad App_T^\circ\, (f, _)\ (x, _ :: \beta^\wedge) \to App_T\ f\ y \\
\qquad\qquad\qquad _ \to x \\
\qquad \ldots
\end{array}$$

The conversion from the original to the extended GADT is applied to the scrutinee of the case expression. This provides type information in the pattern match, allowing it to interact naturally like a conventional dynamic, in this case with the type of the function using the $^\wedge$ annotation. Note that since the conversion function is specific to a program, and not to each case expression, the fields that do not use the annotation must discard the inserted type information, such as in the case for App_T°.

4.3 Formal Translation

We continue by defining the formal translation from the extended language FC^+ to the core language FC. We conjecture that the translation is sound, every well-typed FC^+ program is translated to a well-typed FC program.

Let us begin by translating programs, as depicted by Rule T-PROG in Fig. 3. A program in the FC^+ language is translated to the FC language by translating both the datatype declarations and the function declarations.

In Fig. 4 we define the translation of datatype declarations. Type synonyms and ADTs are left unchanged, as defined by Rules T-DATA-TSYN and T-DATA-ADT. We distinguish GADTs by using the metafunction $\mathsf{annotated}(T)$ to test if it is

$$\boxed{[\![\delta_{\mathsf{FC}^+}]\!] \equiv \overline{\delta}_{\mathsf{FC}}; \phi_{\mathsf{FC}}}$$

$$\frac{}{[\![\mathbf{type}\ T\ \overline{\alpha} = \tau]\!] \equiv \mathbf{type}\ T\ \overline{\alpha} = \tau; \cdot}\ (\textsc{t-data-tsyn})$$

$$\frac{}{[\![\mathbf{data}\ T\ \overline{\alpha} = \overline{C\ \tau}]\!] \equiv \mathbf{data}\ T\ \overline{\alpha} = \overline{C\ \tau}; \cdot}\ (\textsc{t-data-adt})$$

$$\frac{\neg\ \mathsf{annotated}(T)}{[\![\mathbf{data}\ T :: \kappa\ \mathbf{where}\ \overline{C :: \sigma}]\!] \equiv \mathbf{data}\ T :: \kappa\ \mathbf{where}\ \overline{C :: \sigma}; \cdot}\ (\textsc{t-data-gadt-1})$$

$$\frac{\begin{array}{ll} & \mathsf{annotated}(T) \\ \overline{[\![\sigma]\!]_C} \equiv \overline{\sigma^\circ; \overline{x}; \overline{\epsilon^\circ}} & \delta^\circ \equiv \mathbf{data}\ T^\circ :: \kappa\ \mathbf{where}\ \overline{C^\circ :: \sigma^\circ} \\ & \phi^\circ \equiv\ \ \mathbf{fix}\ to T^\circ :: T\ \overline{\alpha} \to T^\circ\ \overline{\alpha} = \\ \mathsf{tarity}(T) \equiv \overline{\alpha} & \qquad \lambda x \to \mathbf{case}\ x\ \mathbf{of} \\ & \qquad\quad \overline{C\ \overline{x} \to C^\circ\ \overline{\epsilon^\circ}} \end{array}}{[\![\mathbf{data}\ T :: \kappa\ \mathbf{where}\ \overline{C :: \sigma}]\!] \equiv \mathbf{data}\ T :: \kappa\ \mathbf{where}\ \overline{C :: \sigma}\ \delta^\circ; \phi^\circ}\ (\textsc{t-data-gadt-2})$$

Fig. 4. Translation of datatypes

$$\boxed{[\![\sigma_{\mathsf{FC}^+}]\!]_C \equiv \sigma_{\mathsf{FC}}; \overline{x}; \overline{\epsilon}_{\mathsf{FC}}}$$

$$\frac{[\![\tau]\!]_{C;0} \equiv \tau^\circ; \overline{x}; \overline{\epsilon^\circ}}{[\![\overline{TC\ \alpha} \Rightarrow \tau]\!] \equiv \overline{TC\ \alpha} \Rightarrow \tau^\circ; \overline{x}; \overline{\epsilon^\circ}}\ (\textsc{t-qtype})$$

Fig. 5. Translation of qualified types

pattern matched somewhere in the program using the GADT annotation (e.g., $\mathsf{annotated}(\mathit{Lam}^\circ) \equiv \mathit{True}$). If not, the original definition is returned without any modifications, as defined by Rule T-DATA-GADT-1. However, an annotated GADT requires some effort. In Rule T-DATA-GADT-2, the translation results in the original definition, an extended definition δ° and a conversion function ϕ°. By translating the types of the constructors, parameterised by the respective constructor name, we obtain extended types together with corresponding pattern variables and expressions that extend these variables. The former is used to define the constructor types of the extended definition, the latter two to define the corresponding conversion function. The metafunction $\mathsf{tarity}(T)$ provides zero or more fresh type variables, determined by the arity of the type T (e.g., $\mathsf{tarity}(\mathit{Lam}_T) \equiv \alpha$).

The translation of qualified types, parameterised by a constructor name, is shown in Fig. 5. A qualified type propagates the translation to its base type, adding a parameter which represents the index of the constructor field type under translation.

In Fig. 6 we define the parameterised translation of such types, resulting in an extended type, pattern variables and expressions that extends these vari-

$$\boxed{[\![\tau_{\mathsf{FC}^+}]\!]_{C;n} \equiv \tau_{\mathsf{FC}}; \overline{x}; \overline{\epsilon}_{\mathsf{FC}}}$$

$$\frac{}{[\![T]\!]_{C;n} \equiv T; \cdot; \cdot}\ (\textsc{t-type-data}) \qquad \frac{}{[\![\tau_1\ \tau_2]\!]_{C;n} \equiv \tau_1\ \tau_2; \cdot; \cdot}\ (\textsc{t-type-app})$$

$$\frac{\neg\ \mathsf{annotated}(C, n) \qquad [\![\tau_2]\!]_{C;n+1} \equiv \tau_2^\circ; \overline{x_2}; \overline{\epsilon_2^\circ}}{[\![\tau_1 \to \tau_2]\!]_{C;n} \equiv \tau_1 \to \tau_2^\circ; x_1\ \overline{x_2}; x_1\ \overline{\epsilon_2^\circ}}\ (\textsc{t-type-fun-1})$$

$$\frac{\mathsf{annotated}(C, n) \qquad [\![\tau_2]\!]_{C;n+1} \equiv \tau_2^\circ; \overline{x_2}; \overline{\epsilon_2^\circ}}{[\![\tau_1 \to \tau_2]\!]_{C;n} \equiv \mathit{TypeOf}\ \tau_1 \to \tau_2^\circ; x_1\ \overline{x_2}; (\mathit{typeOf}\ x_1)\ \overline{\epsilon_2^\circ}}\ (\textsc{t-type-fun-2})$$

Fig. 6. Translation of base types

$$\boxed{[\![\phi_{\mathsf{FC}^+}]\!] \equiv \phi_{\mathsf{FC}}}$$

$$\frac{[\![\epsilon]\!] \equiv \epsilon'}{[\![\mathbf{fix}\ f :: \sigma = \epsilon]\!] \equiv \mathbf{fix}\ f :: \sigma = \epsilon'}\ (\textsc{t-fun})$$

Fig. 7. Translation of functions

ables. Since we are only interested in the fields of a constructor type, and the type of an empty constructor is either a type constructor or a type application, Rules T-TYPE-DATA and T-TYPE-APP result in an unchanged type and no pattern variables or expressions. The function type is the interesting case. If a constructor field is not annotated in the program, as shown in Rule T-TYPE-FUN-1, it is returned unchanged together with a fresh pattern variable and expression that corresponds to the identity. Otherwise, the translation in Rule T-TYPE-FUN-2 extends the type of the constructor field with additional type information and ensures that the fresh pattern variable is extended as well. In both cases we recurse in the translation by incrementing the second parameter to denote the next constructor field.

Next, we define the translation of functions by Rule T-FUN in Fig. 7. A function is translated by translating its body expression, which localises the conversion and thus does not change the type of a function.

The translation of expressions is shown in Fig. 8. The basic building blocks of expressions: bottom, integers, identifiers, and constructors, are left unchanged, as can be seen in Rules T-EXP-BOT, T-EXP-INT, T-EXP-ID, and T-EXP-CON respectively. Translation of an application is defined by Rule T-EXP-APP and translates both its expressions and Rule T-EXP-ABS defines the translation of an abstraction by translating the body expression. For case expressions, we define two separate rules, testing if one of its patterns uses the GADT annotation. If not, it suffices to only translate the scrutinee and the expression of each pattern, as defined by Rule T-EXP-CASE-1. Otherwise, Rule T-EXP-CASE-2 defines that the conversion function must be applied to the translated scrutinee. The name of this function

$$\boxed{[\![\epsilon_{\mathsf{FC}^+}]\!] \equiv \epsilon_{\mathsf{FC}}}$$

$$\frac{}{[\![\bot]\!] \equiv \bot}\ (\textsc{t-exp-bot}) \qquad \frac{}{[\![i]\!] \equiv i}\ (\textsc{t-exp-int})$$

$$\frac{}{[\![x]\!] \equiv x}\ (\textsc{t-exp-id}) \qquad \frac{}{[\![C]\!] \equiv C}\ (\textsc{t-exp-con})$$

$$\frac{[\![\epsilon_1]\!] \equiv \epsilon_1' \qquad [\![\epsilon_2]\!] \equiv \epsilon_2'}{[\![\epsilon_1\ \epsilon_2]\!] \equiv \epsilon_1'\ \epsilon_2'}\ (\textsc{t-exp-app}) \qquad \frac{[\![\epsilon]\!] \equiv \epsilon'}{[\![\lambda x \to \epsilon]\!] \equiv \lambda x \to \epsilon'}\ (\textsc{t-exp-abs})$$

$$\frac{\begin{array}{c} x ::^{\mathcal{G}} \alpha\ \notin\ \overline{\theta} \\ [\![\epsilon_s]\!] \equiv \epsilon_s' \qquad \overline{\theta} \equiv \overline{\rho} \qquad \overline{[\![\epsilon]\!]} \equiv \overline{\epsilon'} \end{array}}{[\![\mathbf{case}\ \epsilon_s\ \mathbf{of}\ \overline{\theta \to \epsilon}]\!] \equiv \mathbf{case}\ \epsilon_s'\ \mathbf{of}\ \overline{\rho \to \epsilon'}}\ (\textsc{t-exp-case-1})$$

$$\frac{\begin{array}{c} x ::^{\mathcal{G}} \alpha\ \in\ \overline{\theta} \qquad \mathsf{btype}(\epsilon_s) \equiv T \\ [\![\epsilon_s]\!] \equiv \epsilon_s' \qquad \overline{[\![\theta]\!]} \equiv \overline{\rho} \qquad \overline{[\![\epsilon]\!]} \equiv \overline{\epsilon'} \end{array}}{[\![\mathbf{case}\ \epsilon_s\ \mathbf{of}\ \overline{\theta \to \epsilon}]\!] \equiv \mathbf{case}\ to T^{\circ}\ \epsilon_s'\ \mathbf{of}\ \overline{\rho \to \epsilon'}}\ (\textsc{t-exp-case-2})$$

$$\frac{[\![\epsilon]\!] \equiv \epsilon'}{[\![\mathbf{dynamic}\ \epsilon :: \omega]\!] \equiv \mathbf{dynamic}\ \epsilon' :: \omega}\ (\textsc{t-exp-dyn})$$

Fig. 8. Translation of expressions

$$\boxed{[\![\theta_{\mathsf{FC}^+}]\!] \equiv \rho_{\mathsf{FC}}}$$

$$\frac{}{\overline{[\![\varrho]\!]} \equiv \varrho}\ (\textsc{t-pat-base}) \qquad \frac{\overline{[\![\vartheta]\!]}_{C;\mathsf{index}} \equiv \overline{\rho}}{[\![C\ \overline{\vartheta}]\!] \equiv C^{\circ}\ \overline{\rho}}\ (\textsc{t-pat-con})$$

Fig. 9. Translation of patterns

is determined by the metafunction $\mathsf{btype}(\epsilon_s)$ which determines the base name of the type of the scrutinee ϵ_s (e.g., $\mathsf{btype}(\mathit{Const}_T\ 1) \equiv \mathit{Lam}_T$). Furthermore, each pattern is translated so that the actual use of the annotation is translated. As we will see in a moment, the translation of patterns takes care of renaming the constructors, which is required since the scrutinee is converted to the extended type. Finally, Rule T-EXP-DYN defines the translation of a dynamic, simply translating its expression.

Patterns possibly provide access to the inserted type information, their translation is shown in Fig. 9. A base pattern is left untouched, as depicted in Rule T-PAT-BASE. In Rule T-PAT-CON, the constructor in a constructor pattern is renamed and its fields are all translated, parameterised by the name of the original constructor and a metavalue index that provides the index of each constructor field.

$$\boxed{[\![\vartheta_{\mathsf{FC}^+}]\!]_{C;n} \equiv \rho_{\mathsf{FC}}}$$

$$\frac{\neg\ \mathsf{annotated}(C, n)}{[\![\rho]\!]_{C;n} \equiv \rho}\ (\textsc{t-fpat-pat-1}) \qquad \frac{\mathsf{annotated}(C, n)}{[\![\rho]\!]_{C;n} \equiv (\rho, \_)}\ (\textsc{t-fpat-pat-2})$$

$$\frac{}{[\![x ::^{\mathcal{G}} \omega]\!]_{C;n} \equiv (x, \_ :: \omega)}\ (\textsc{t-fpat-ann})$$

Fig. 10. Translation of field patterns

In Fig. 10 we conclude the translation from FC^+ to FC by defining the translation of field patterns, being the language extension itself. Since the conversion function that inserts type information is specific to a program, we have to verify if the current field pattern is annotated somewhere in the program. Rule T-FPAT-PAT-1 defines that if a field is never annotated, it need not to be translated. Otherwise, the additional information is discarded, as defined by Rule T-FPAT-PAT-2. The core of the translation is captured by Rule T-FPAT-ANN. A GADT annotation is erased by translating it to a dynamic type annotation, yielding a pair that matches the original value and the type that is stored in the dynamic.

5 Related Work

The foundations of structured programming on GADTs [7] provide an elegant approach to defining algebras on GADTs. While such algebras provide an abstraction mechanism to define an update function, explicit type representations and equality types [3, 5] are still required. In Section 3.2, we discussed the disadvantages of such an approach. In our work, type representations and type equality proofs are implicitly provided by dynamics, which significantly improves the elegancy of the function definitions.

Another approach to heterogeneous structures reflects the structure of a value directly in its type [8]. For example, the type of a heterogeneous list is basically a structure of nested tuples. Then, functions are defined on such structures using the type class mechanism, dispatching on the type structure. To enforce type-safe updates, yet another type class is defined to reflect type equality. Consequently, this approach results in rather verbose definitions since all action takes place on the level of type classes. Since the structure of the types are available, direct manipulation enables type-changing functions. Looking at the type of the update function in Section 3.3, our approach seems to forbid any type-changing updates. However, subterms can be replaced by arbitrary complex terms, thereby changing the underlying type structure.

6 Conclusion

We have presented the typical synergy between dynamics and GADTs to elegantly define functions that manipulate GADTs, requiring instantiation information on polymorphic types. Our approach comprises a new GADT annotation and improves upon boilerplate type representation administration in conventional approaches, since the functions are not cluttered any more with type equality witnesses and manual proofs. Also, by using dynamics, we no longer need to maintain a closed enumeration of the used types. Above all, our approach scales up to more complex structures and functions due to its simplicity. We have shown that the language extension is easily translated to a functional core that supports both dynamics and GADTs.

One of the major limitations in our approach is that the use of type codes limits the use of the GADT annotation to non-abstract types. It remains future work to define type codes for such types, as well as investigating if dynamics can be implemented without type codes as class constraints. This would improve our approach considerably since it will no longer require us to decorate GADTs beforehand with type code constraints. Also, we plan to verify our hypothesis that storing GADTs in dynamics is no different from conventional ADTs.

Despite these limitations, the translation to dynamics provides novel opportunities, such as type dispatching and enforcing type equality invariants on GADTs. These opportunities require a more intricate translation than described in this paper, since this class of functions projects values instead of manipulating the values as such.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. 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, Pittsburgh, PA, USA*, pages 157–166. ACM, 2002.
4. Robert Cartwright and James Donahue. The semantics of lazy (and industrious) evaluation. In *Proceedings of the 2nd Symposium on LISP and Functional Programming, LFP '82, Pittsburgh, PA, USA*, pages 253–264. ACM, 1982.

5. James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Proceedings of the 6th Haskell Workshop, Haskell '02, Pittsburgh, PA, USA*, pages 90–104. ACM, 2002.
6. James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, 2003.
7. Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In George Necula and Philip Wadler, editors, *Proceedings of the 35th Symposium on Principles of Programming Languages, POPL '08, San Francisco, CA, USA*, pages 297–308. ACM, 2008.
8. Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In Henrik Nilsson, editor, *Proceedings of the 8th Haskell Workshop, Haskell '04, Snowbird, UT, USA*, pages 96–107. ACM, 2004.
9. Konstantin Läfer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.
10. 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, Portland, Oregon, USA*, pages 50–61. ACM, 2006.
11. 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, Lochem, The Netherlands*, pages 169–185. Springer-Verlag, 1999.
12. Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In Graham Hutton and Andrew Tolmach, editors, *Proceedings of the 14th International Conference on Functional Programming, ICFP '09, Edinburgh, Scotland*, pages 341–352. ACM, 2009.
13. Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Philip Trinder, Greg Michaelson, and Ricardo Pena, editors, *Selected Papers of the 14th International Symposium on the Implementation of Functional Languages, IFL '02, Edinburgh, UK*, pages 101–117. Springer-Verlag, 2003.
14. Hongwei Xi, Chiyan 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, New Orleans, LA, USA*, pages 224–235. ACM, 2003.