Chapter 5

A Leaner Specification for Generalized Algebraic Data Types

Arie Middelkoop[1], Atze Dijkstra[1], S. Doaitse Swierstra[1]
Category: Research

Abstract: The type systems of current approaches for dealing with Generalized Algebraic Data Types (GADTs) tend to be more algorithmic than declarative in nature, and are incomplete in the sense that they deal with specific issues only. When implementing GADTs, this raises the question whether complex type system infrastructure is needed, and secondly, if all requirements were taken into account during implementation. We answer these questions by giving a declarative specification with less demands on infrastructure, and which deals with the key issues related to a GADT implementation.

5.1 INTRODUCTION

Generalized Algebraic Data Types (GADTs for short) allow for additional equalities to hold between type variables, witnessed by pattern matching on GADT constructors. These equalities are then used to coerce the type of an expression to an equivalent type. We give a more extensive introduction to GADTs in Section 5.3.

There are many applications that show that GADTs are a useful addition to languages. For example when dealing with transformations on typed abstract syntax for the implementation of domain-specific languages [2]. Therefore, we added support for GADTs in EHC, the Haskell compiler that we are developing at Utrecht University. We describe our implementation in terms of some specification. However, we were unable to find a specification that is sufficiently simple

[1]Universiteit Utrecht, The Netherlands; {ariem,atze,doaitse}@cs.uu.nl

to match our implementation, because of two reasons. We sketch these reasons here, and defer a more thorough discussion to Section 5.2.

The first reason is that the type systems given by most approaches are complex. A reason for this is that algorithmic type systems are given, which rely on advanced type system infrastructures for the implementation. This makes it harder to incorporate ideas when the infrastructures differ.

Secondly, the GADT-aspect of an approach is often not discussed in isolation, but alongside with other language features. This makes it difficult to determine if one requirement of an approach relates to the GADT-aspect, or is a requirement related to another language feature.

Therefore, we conclude that a specification is needed that is less complex and only takes the key issues of GADTs into account. The techniques we use are not novel: we take the constraint-based approach of Sulzmann et al. [11] and combine this with the unification-based approach of Peyton Jones et al. [6]. However, the resulting type system is easier and less tied to a particular implementation.

More concretely, we give:

- A declarative type system (Section 5.4) for an explicitly typed System F-like language extended with GADTs, using qualified types with equalities as qualifiers. Type conversions are restricted without having to resort to rigid types [6] or type shapes [8].

- A denotational semantics (Section 5.5) by giving a translation to a subset of System F_C, which is a System F-like language with explicit type coercions, defined by Sulzmann et al. [10].

5.2 RELATED WORK

An approach for GADTs consists of two components: a type information propagation component and a type conversion component. A type information propagation strategy determines what is known type information based on user-supplied type signatures, and what is inferred type information. A type conversion strategy deals with the construction of coercion terms.

5.2.1 Type information propagation strategies

Pottier et al. [8] use shape inference, where a shape represents known type information based on user-supplied signatures. These shapes are spread throughout the syntax tree in order to locate possible coercions. Their approach uses complex algorithms to spread the shapes as far as possible (depending on some quality versus performance tradeoff parameter). The essential part concerning GADTs is that incompatible shapes are normalized with respect to some equation system, which is not made explicit in their work. From an implementer's point of view, this description is a concrete description of the equation system, and it requires infrastructure for spreading shapes.

Similarly, Peyton Jones et al. [6, 7] define a notion of wobbly types to combine type checking and type inference, which is based on earlier work on boxy types [12]. A rigid type represents known type information based on user-supplied type information, whereas a wobbly type is based on inferred information. The idea is then that type conversions are only applied on rigid types, for reasons of predictability and most-general typing. Aside from wobbly types, the authors use concepts such as "fresh most general unifiers" and lexically scoped type variables in their presentation. It is hard to distinguish which of these concepts are really required, and which of these concepts are actually related to other language features that are covered by their approach (such as type families).

We incorporated an equivalent notion of wobbly and rigid types in our specification: in an explicitly typed System F-variant, the skolemnized type constants are rigid types, and we only allow type conversions on those.

The exact choice of propagation strategy is an orthogonal issue. By allowing type conversions only on known type information, a better propagation strategy just means that less explicit types have to be given by the programmer. This is the reason why we have chosen an explicitly typed source language in the first place. In fact, for our own implementation of this specification, we piggy backed on the infrastructure of the implementation of a higher-ranked impredicate type system. It has two propagation strategies, of which the advanced one has a concept of hard and soft type context, which can be compared to rigid and boxy types [4].

5.2.2 Type conversion strategies

Peyton Jones et al. [6, 7] use a unification-based strategy, where type conversion is called type refinement. A fixpoint substitution is constructed that, for each type equation introduced by pattern matching, contains a mapping for each (non-wobbly) type component of the left hand side of an equation, to the corresponding type component on the right hand side of an equation. The substitution is then used to normalize the types under consideration. The presentation is intertwined with the type propagation strategy, which makes it hard to separate these concepts. More recent continuation of this work by Sulzmann et al. [10] puts the GADT aspect in relative isolation.

Wazny [13] uses a constraint-based strategy. The GADT aspect of this strategy is covered separately by Sulzmann et al. [11, 9]. They also formulate the typing problems in terms of solving constraints with CHRs. The difference is that we restrict ourselves to equality constraints, and do not need the machinery required to solve implication constraints. Furthermore, their typing/translation rules do not mention how to deal with existential data types, which may be transparent to the given approach, but is of interest to a reader because GADT examples [2] often use them. Finally, in contrast to Peyton Jones et al., restrictions on type conversions are not mentioned.

We rely on their encoding for GADTs as qualified types in our approach (i.e. data constructors take a list of equality constraints). This encoding makes explicit which type variables are equated to what types, which gives slightly more infor-

mation than an encoding with type signature notation (although these encodings can be mapped to each other).

As an overall observation concerning the related work, we use a constraint-based strategy similar to that of Wazny, but use the unification-based strategy of Peyton Jones et al. for the specific part of the implementation that deals with the decomposition of equality constraints.

5.3 MOTIVATION AND EXAMPLES

A typical implementation of an embedded domain specific language consists of some combinators to construct an abstract syntax tree, and some functionality in the host language to manipulate this abstract syntax tree. After analysis and transformation, the abstract syntax tree is translated to some denotation in the host language in order to use it.

For example, assume that we use Haskell as a host language and embed an expression language containing only tuples and numbers, using the following abstract syntax:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr} \\
\quad = \mathit{Num}\ \mathit{Int} \\
\quad \mid \mathit{Tup}\ \ \mathit{Expr}\ \mathit{Expr}
\end{array}
$$

However, the following straightforward translation of the expression to a tuple in the host language does not type check, because the inferred types for the case alternatives are not the same:

$$
\begin{array}{l}
\mathit{eval}\ e = \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad \mathit{Num}\ i \ \ \rightarrow i \\
\qquad\qquad \mathit{Tup}\ p\ q \rightarrow (\mathit{eval}\ p, \mathit{eval}\ q)
\end{array}
$$

We can bypass this restriction imposed by the Haskell type system by using a typed abstract syntax and encode a proof that the generated tuples are type correct. For that, we add a type parameter t to the abstract syntax, which represents the type of the expression, and embed in the constructors a proof (with type $\mathit{Equal}\ t\ t'$) that states that this t is equal to the real type t' of this specific expression (an *Int* for a *Num* and some tuple type for a *Tup*):

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr}\ t \\
\quad = \qquad \mathit{Num}\ (\mathit{Equal}\ t\ \mathit{Int}) \quad \mathit{Int} \\
\quad \mid \forall a\ b\ .\ \mathit{Tup}\ \ (\mathit{Equal}\ t\ (a,b))\ (\mathit{Expr}\ a)\ (\mathit{Expr}\ b)
\end{array}
$$

Baars et al. [1] give a definition of this *Equal* data type and some operations, including a function *coerce* that converts the type to a proved equivalent type, and some combinators to construct equality proofs:

$$
\begin{array}{ll}
\mathit{coerce} & :: \mathit{Equal}\ a\ b \rightarrow a \rightarrow b \\
\texttt{sym} & :: \mathit{Equal}\ a\ b \rightarrow \mathit{Equal}\ b\ a \\
\mathit{refl} & :: \mathit{Equal}\ a\ a
\end{array}
$$

Now, we can modify the *eval* function in such a way that the case alternatives have the same type (namely the t in *Expr t*):

$$
\begin{array}{l}
eval :: Expr\ t \to t \\
eval\ e = \textbf{case}\ e\ \textbf{of} \\
\qquad\qquad Num\ ass\ i \quad\ \to coerce\ (sym\ ass)\ i \\
\qquad\qquad Tup\ \ ass\ p\ q \to coerce\ (sym\ ass)\ (eval\ p, eval\ q)
\end{array}
$$

The assumptions used by *eval* need to be proved when constructing values of type *Expr t*, which we achieve by using *refl*:

$$
Tup\ refl\ (Num\ refl\ 4)\ (Num\ refl\ 2) \qquad :: Expr\ (Int, Int)
$$

The important observation to make at this point is that the proofs are a *static* property of the program. Hence, the goal is to construct these proofs automatically, at those places explicitly indicated by the programmer using type signatures. In case of the example, at those places where the type t shows up.

In order not to tie ourselves to Haskell or to a specific implementation of a type system, we use as source language an explicitly typed lambda calculus, called System F_A [10], where the equalities are not encoded as values, but with qualified type notation similar to Stuckey et al. [9]:

$$
\begin{array}{l}
\textbf{data}\ Expr\ t = \\
\quad |\qquad (t \doteq Int) \quad\ \Rightarrow Val\ Int \\
\quad | \exists a\ b\ .\ (t \doteq (a,b)) \Rightarrow Tuple\ (Expr\ a)\ (Expr\ b) \\
;\textbf{let}\ (eval :: \forall t\ .\ Expr\ t \to t) \\
\quad = \Lambda t \to \lambda (e :: Expr\ t) \to \\
\qquad (\textbf{case}\ e\ \textbf{of} \\
\qquad\quad Num\ (x :: Int) \to x \\
\qquad\quad (Tup\ p\ q)\ a\ b \to (,)\ (eval\ a\ p)\ (eval\ b\ q) \\
\qquad)\ ::\ t \\
\textbf{in}\ eval\ (Tup\ (Num\ 4)\ (Num\ 2))
\end{array}
$$

Similarly, instead of translating to Haskell, we translate to an explicitly typed lambda calculus with native support for equality proofs, called System F_C [10]:

$$
\begin{array}{l}
\textbf{data}\ Expr\ t = \\
\quad |\qquad Num\ (t{\sim}Int) \quad\ Int \\
\quad | \exists a\ b\ .\ Tup\ \ (t{\sim}(a,b))\ (Expr\ a)\ (Expr\ b) \\
;\textbf{let}\ (eval :: \forall t\ .\ Expr\ t \to t) \\
\quad = \Lambda t \to \lambda (e :: Expr\ t) \to \\
\qquad (\textbf{case}\ e\ \textbf{of} \\
\qquad\quad Num\ eqInt\ (x :: Int) \ \to x \qquad\qquad\qquad\qquad \triangleright \texttt{sym}\ eqInt \\
\qquad\quad (Tup\ eqTup\ p\ q)\ a\ b \to (,)\ (eval\ a\ p)\ (eval\ b\ q) \triangleright \texttt{sym}\ eqTup \\
\qquad)\ ::\ t \\
\textbf{in}\ eval\ (Tup\ (Int, Int)\ (Num\ Int\ 4)\ (Num\ Int\ 2))
\end{array}
$$

Each constructor contains equality proofs of type *t1*∼*t2* (a proof that *t1* and *t2* are equal), called *coercions*, which can be used after pattern matching on them. The ▷ operator corresponds with the *coerce* function given above. Note that `sym` in this case is not a function, but a construction in the target language that operates on coercions, and that a type t as coercion represents the reflexivity $t{\sim}t$.

A specification for GADTs consists of two parts: a type system for the source language (Section 5.4), and a translation that constructs the equality proofs and inserts the coercions (Section 5.5).

5.4 TYPE SYSTEM

Source language We use an explicitly typed source language extended with GADTs . This source language is a minor variation on System F_A [10], which we call System F'_A . The key difference is support for existential quantification and a slightly more uniform representation of pattern matches. The syntax of System F'_A is given in Figure 5.1.

A v is an identifier of which the denotation is a unique type constant. Although these identifiers can appear where a type or type variable is expected, they play an important role later. They are introduced at two places: when opening an existential using a pattern match (P.APP.EXIST) and with a universal abstraction (E.UNIV.ABS).

Each equality is encoded as a qualified type. The LHS is – without loss of generality – syntactically restricted to be a (bound) type variable. Existentials for a data constructor are introduced with an $\exists$ instead of the $\forall$ that is written in Haskell there.

The pattern language is similar to the expression language. Identifiers are bound to fields of a constructor using applications of variables. Likewise, universally quantified variables are instantiated by applying a type, and existentials are opened by applying a unique type constant.

Notation First some notation before we give a type system for the source language. The semicolon in the rules for patterns acts as a separator to indicate that Γ and Δ are separate environments. Juxtaposition of environments (i.e. $\Delta\ \Gamma$) represents environment concatenation. $\Gamma\ (x) = t$ states that x is bound to t in Γ. Furthermore, we use an overbar to indicate a list. A single value at the position where a list is expected is implicitly assumed to be a singleton list. Juxtaposition of lists represents list concatenation. The components of the list are accessible with a subscript i. A type $\tau\ [\overline{\tau}]$ is obtained by replacing some components of τ in some fixed way with types $\overline{\tau}$. With $\tau^a = \tau\ [\overline{\tau^a}]$ and $\tau^b = \tau\ [\overline{\tau^b}]$, we express that τ^a and τ^b have a common structure τ, with corresponding differences in τ^a_i and τ^b_i respectively.

Type System The type rules for expressions are given in Figure 5.3, with the meaning that in environment Γ, the expression e has type τ. The type rules for a

$e \in Expr$

$\rightarrow$	C	(E.CON)
$\mid$	x	(E.VAR)
$\mid$	$e\ e$	(E.APP.EXPR)
$\mid$	$e\ \tau$	(E.APP.UNIV)
$\mid$	$\lambda p\ .\ e$	(E.LAM.ABS)
$\mid$	$\Lambda v\ .\ e$	(E.UNIV.ABS)
$\mid$	$\textbf{let}\ \overline{p = e}\ \textbf{in}\ e$	(E.LET)
$\mid$	$\textbf{case}\ e\ \textbf{of}\ \overline{p \rightarrow e} :: \tau$	(E.CASE)

$d \in Decl$

$$\rightarrow\ \textbf{data}\ D\ \overline{\alpha} = \overline{\mid \exists \overline{\beta}\ .\ \overline{x \doteq \tau} \Rightarrow C\ \overline{\tau}}$$

$t \in Program$

$$\rightarrow\ \overline{d}; e \qquad \text{(TOPLEVEL)}$$

$x, \alpha, \beta \in Var \qquad D \in TyCon$
$v \in TyConst \qquad C \in ValCon$
$TyConst \subseteq Var$

$p \in Pattern$

$\rightarrow$	C	(P.CON)
$\mid$	$x :: \tau$	(P.VAR)
$\mid$	$p\ p$	(P.APP.PAT)
$\mid$	$p\ \tau$	(P.APP.UNIV)
$\mid$	$p\ v$	(P.APP.EXIST)

$\tau \in Type$

$\rightarrow$	C	(T.CON)
$\mid$	x	(T.VAR)
$\mid$	$\tau\ \tau$	(T.APP)
$\mid$	$\forall x\ .\ \tau$	(T.FORALL)
$\mid$	$\exists x\ .\ \tau$	(T.EXISTS)
$\mid$	$\overline{\tau \doteq \tau} \Rightarrow \tau$	(T.EQS)

$\Gamma, \Delta \in Env$

$\rightarrow$	$x :: \tau,\ \Gamma$
$\mid$	$x,\ \Gamma$
$\mid$	$\tau \doteq \tau,\ \Gamma$

Figure 5.1. Syntax of System F'_A

pattern given in Figure 5.4 denote that in environment Γ, the pattern p has type τ, with bindings for variables in Δ.

When we ignore the special rules E.APP.EQS, E.COERCE, and P.APP.EQS for the moment, the rules are a minor variation on a type system for System F. The differences are:

- Scoping is made explicit by collecting bindings for a pattern in a local environment Δ.
- An existential is opened with a pattern match using P.APP.EXIST by instantiation to a fresh fixed type variable v (a type constant). Such a v may not escape the scope of the pattern match, which is enforced by demanding that the only variables that may escape are those that are bound in the global environment Γ in rules E.LAM.ABS, E.LET, and E.CASE.
- There are two type signatures in the environment for a constructor: a constructor signature $\Gamma\ (C)$ and a deconstructor signature $\Gamma\ (C^{\diamond})$. The constructor signature defines which which equalities need to be proved and which values have to be supplied. The deconstructor signature gives the dual definition: which values can be extracted when pattern matching against this constructor and which equalities can be assumed to be proved. Finally, Figure 5.2 defines how these signatures are derived from a data type declaration.

$$\boxed{\Gamma \vdash_d d}$$

$$\frac{\begin{array}{c}\Gamma\ (C_i^{\diamond}) = \forall\overline{\alpha}\ \exists\overline{\beta}_i\ .\ (\overline{x^l \doteq \tau^r}) \Rightarrow \tau_{i,0} \to \ldots \to \tau_{i,n_i} \to D\ \overline{\alpha}_i \\ \Gamma\ (C_i) = \forall\overline{\alpha}\forall\overline{\beta}_i\ .\ (\overline{x^l \doteq \tau^r}) \Rightarrow \tau_{i,0} \to \ldots \to \tau_{i,n_i} \to D\ \overline{\alpha}_i \\ D \in \Gamma\end{array}}{\Gamma \vdash_d \textbf{data}\ D\ \overline{\alpha} = |\ \exists\overline{\beta}.\ (\overline{x^l \doteq \tau^r}) \Rightarrow C\ \overline{\tau}}\ \text{ADT}_G$$

Figure 5.2. Data definition type rules (G)

Type Conversions Without the three special rules, typing the example of Section 5.3 fails. We demand that the right-hand sides of the case alternatives are of type t, but the first case alternative is of type *Int*. The pattern match against the *Num* constructor gives us the proof that the *Int* is actually equal to the t. We exploit this knowledge using rule E.COERCE, by substituting the t for *Int* when typing the case alternative. Only those type constants (like t) may be substituted; we discuss this later. This rule uses the entailment relation $\Vdash$, which states that in environment Γ, the two types are proved to be equal.

$$\boxed{\Gamma \vdash_e e : \tau}$$

$$\frac{\Gamma \vdash_e e : \overline{[v := \tau]}\tau' \quad \Gamma \Vdash \tau_i \doteq v_i}{\Gamma \vdash_e e : \tau'} \text{E.COERCE}_G$$

$$\frac{\Gamma \Vdash \tau_i^l \doteq \tau_i^r \quad \Gamma \vdash_e e : (\overline{\tau^l \doteq \tau^r}) \Rightarrow \tau}{\Gamma \vdash_e e : \tau} \text{E.APP.EQS}_G \qquad \frac{\Gamma\,(C) = \tau}{\Gamma \vdash_e C : \tau} \text{E.CON}_G$$

$$\frac{\Gamma\,(x) = \tau}{\Gamma \vdash_e x : \tau} \text{E.VAR}_G \qquad \frac{\Gamma \vdash_e a : \tau^a \quad \Gamma \vdash_e f : \tau^a \to \tau}{\Gamma \vdash_e f\ a : \tau} \text{E.APP.EXPR}_G$$

$$\frac{\Gamma \vdash_e f : \forall \alpha.\ \tau}{\Gamma \vdash_e f\ \tau^a : [\alpha := \tau^a]\ \tau} \text{E.APP.UNIV}_G$$

$$\frac{\Gamma; \Delta \vdash_p p : \tau^a \quad \Delta\ \Gamma \vdash_e e : \tau \quad \texttt{ftv}\,(\tau) \cap \texttt{ftv}\,(\Delta) \subseteq \texttt{ftv}\,(\Gamma)}{\Gamma \vdash_e \lambda p\ .\ e : \tau^a \to \tau} \text{E.LAM.ABS}_G$$

$$\frac{v, \Gamma \vdash_e e : \tau \quad v \notin \texttt{ftv}\,(\Gamma)}{\Gamma \vdash_e \Lambda v\ .\ e : \forall v.\ \tau} \text{E.UNIV.ABS}_G \qquad \frac{\Gamma; \Delta \vdash_p p_i : \tau_i \quad \Delta\ \Gamma \vdash_e e : \tau \quad \Delta\ \Gamma \vdash_e e_i : \tau_i \quad \texttt{ftv}\,(\tau) \cap \texttt{ftv}\,(\Delta) \subseteq \texttt{ftv}\,(\Gamma)}{\Gamma \vdash_e \textbf{let}\ \overline{p = e}\ \textbf{in}\ e : \tau} \text{E.LET}_G$$

$$\frac{\Gamma; \Delta_i \vdash_p p_i : \tau^p \quad \Delta_i \Gamma \vdash_e e_i : \tau \quad \Gamma \vdash_e e^s : \tau^p \quad \texttt{ftv}\,(\tau^p) \cap \texttt{ftv}\,(\Delta_i) \subseteq \texttt{ftv}\,(\Gamma)}{\Gamma \vdash_e (\textbf{case}\ e^s\ \textbf{of}\ \overline{p \to e}) :: \tau : \tau} \text{E.CASE}_G$$

Figure 5.3. Expression type rules (G)

$$\boxed{\Gamma;\Delta \vdash_p p : \tau}$$

$$\frac{\Gamma\ (C^\diamond) = \tau}{\Gamma;\Delta \vdash_p C : \tau}\ \text{P.CON}_G \qquad \frac{\Delta\ (x) = \tau}{\Gamma;\Delta \vdash_p x :: \tau : \tau}\ \text{P.VAR}_G$$

$$\frac{\Gamma;\Delta \vdash_p a : \tau^a \quad \Gamma;\Delta \vdash_p f : \tau^a \to \tau}{\Gamma;\Delta \vdash_p f\ a : \tau}\ \text{P.APP.PAT}_G \qquad \frac{\Gamma;\Delta \vdash_p f : \forall\alpha.\ \tau}{\Gamma;\Delta \vdash_p f\ \tau^a : [\alpha := \tau^a]\ \tau}\ \text{P.APP.UNIV}_G$$

$$\frac{\Gamma;\Delta \vdash_p f : \exists\alpha\ .\ \tau \quad v \in \Delta \quad v \notin \texttt{ftv}\ (\Gamma)}{\Gamma;\Delta \vdash_p f\ v : [\alpha := v]\ \tau}\ \text{P.APP.EXIST}_G$$

$$\frac{\Gamma;\Delta \vdash_p f : (\overline{\tau^l \doteq \tau^r}) \Rightarrow \tau \quad \tau_i^l \doteq \tau_i^r \in \Delta}{\Gamma;\Delta \vdash_p f : \tau}\ \text{P.APP.EQS}_G$$

Figure 5.4. Pattern type rules (G)

The rule E.APP.EQS is used to actually prove that t is equal to *Int* when we construct a value with the constructor *Num*, and rule P.APP.EQS allow us to extract this proof from the *Num* constructor and introduce it as an assumption in the environment.

Entailment Figure 5.5 gives the entailment rules. A derivation of these rules is a proof of equality between two types. The first two rules represent symmetry and transitivity of an equality relation. Rule E.ASSUME uses an assumption from the environment. The subsumption rule decomposes an equality proof in proofs for components of type types. The congruence rules allows for proving an equality with sub components converted if there is an equality proof for it. Again, only type constants v need to be converted. The decomposition and subsumption rules are often needed when a type conversion has to be applied deep inside a type. Transitivity is not used often, but there are examples [2] with matches on more than one constructor where transitivity is needed to combine equality proofs.

Discussion The explicitly typed source language allows us to abstract from a particular choice of type checking and type inference strategy, which (although important) is a separate issue. There is, however, a concept we have to take into account. A successful GADT pattern match results in additional assumptions between the equality of types, allowing types to be converted. Not all types are allowed to be converted for reasons of predictability and most general typing (when

$$\boxed{\Gamma \Vdash \tau^l \doteq \tau^r}$$

$$\frac{\Gamma \Vdash \tau^r \doteq \tau^l}{\Gamma \Vdash \tau^l \doteq \tau^r}\,\text{E.SYM}_G \qquad \frac{\Gamma \Vdash \tau^l \doteq \tau^a \quad \Gamma \Vdash \tau^a \doteq \tau^r}{\Gamma \Vdash \tau^l \doteq \tau^r}\,\text{E.TRANS}_G$$

$$\frac{\tau^l \doteq \tau^r \in \Gamma}{\Gamma \Vdash \tau^l \doteq \tau^r}\,\text{E.ASSUME}_G \qquad \frac{\Gamma \Vdash v_i \doteq \tau_i \quad \Gamma \Vdash [\overline{v := \tau}]\ \tau^a \doteq \tau^b}{\Gamma \Vdash \tau^a \doteq \tau^b}\,\text{E.CONGR}_G$$

$$\frac{\Gamma \Vdash \tau\ [\overline{\tau^a}] \doteq \tau\ [\overline{\tau^b}]}{\Gamma \Vdash \tau^a_i \doteq \tau^b_i}\,\text{E.SUBSUME}_G$$

Figure 5.5. Entailment rules (G)

dealing with type inference). This is the reason why Peyton Jones et al. distinguish rigid types [6]. In our explicitly typed system, the concept of rigid types relates to type constants v introduced by universal abstraction and existential pattern matching. By allowing conversions only to such a constant, we obtain a specification that takes into account which types are allowed to be converted, while leaving the choice of type inference or propagation strategy up to the implementation.

So, the type constants v determine the positions where types can be converted. In the example of Section 5.3, the requirement that the case alternatives all need to be of type t (coming from *Expr t*), dictates that there needs to be a conversion from the actual type of the case alternative to this type t.

Furthermore, note that the three special rules are not syntax directed. An implementation decides where to apply these rules. For example, assuming that the target language is extended with some additional syntax, and that the example of Section 5.3 is lifted to some evaluation monad m:

```
data Expr t
   = Val (t~Int) t
   | ∀a b . Tuple (t~(a,b)) (Expr a) (Expr b)
;let (eval :: ∀t m . Monad m ⇒ Expr t → m t)
   = Λt m → λ(e :: Expr t) →
     (case e of
        Val      eqInt (x :: t)     → return m (x ▷ sym eqInt)
        (Tuple eqTup p q) a b →
                 do ep ← eval m a p
                    eq ← eval m b q
                    return m ((,) ep eq ▷ sym eqTup)) :: m t
```

In the above code, the type conversion is applied as deep as possible. Another possibility is to convert as shallow as possible. For example, by changing the first case alternative to *return m x* $\triangleright$ (*m* (`sym` *eqInt*)), where the coercion *m* (of type $m{\sim}m$) represents reflexivity, and the application of the coercions represent a coercion of type *m Int*$\sim$*m t*. Other possibilities are a mixture between shallow and deep. This choice should not have an effect on the outcome of the program, but since it affects the structure of the target expression, a particular choice may be more beneficial depending on a particular implementation.

5.5 TRANSLATION

We give a translation to System F_C in order to give a semantics to the GADTs in our source language.

Target Language We limit our explanation to the fragment of System F_C that we need. This fragment is given in Figure 5.6. See Sulzmann et al. [10] for a full explanation. There are some essential differences with respect to the source language:

- A proof of equality is made explicit as a *coercion* value γ with the type $\tau^1{\sim}\tau^2$, meaning that γ is a witness that the type τ^1 is equal to type τ^2. Constructors store coercions as additional fields, and require them to be passed (E.APP.COE) when constructing values with such a constructor. Pattern matching against such a constructor bind an identifier to the coercion, allowing referencing to this coercion (C.VAR).
- The *cast* operator $\triangleright$ takes an expression $\hat{e}$ of type τ^1, and a coercion of type $\tau^1{\sim}\tau^2$, and converts the type of $\hat{e}$ to τ^2.
- There are several language constructs to operate on coercions. Transitivity ($\gamma^1 \circ \gamma^2$) and symmetry (`sym`) have the usual interpretation. The `left` construct decomposes a coercion on type applications to a coercion of only the function part. Similarly, the `right` construct decomposes a coercion to the argument part. Coercion application $\gamma^1\ \gamma^2$ creates a coercion that applies γ^1 to the function part of a type application and γ^2 to the argument part. Reflexivity of type $\tau{\sim}\tau$ is encoded as the coercion (not type!) τ. The other two constructs deal with quantors in types.

See Sulzmann et al. [10] for a type system of the target language.

Coercion construction In environment Γ, the source expression e with type τ is translated to the same expression $\hat{e}$ in the target language, with two exceptions (Figure 5.7). The key idea here is that a derivation of entailment is an equality proof out of which a coercion is constructed. The entailment relation expresses here that in environment Γ, τ^l is equal to τ^r, witnessed by the coercion γ (of type $\tau^l{\sim}\tau^r$).

$$
\begin{array}{lll}
\hat{e} \to & e & \\
\mid & \hat{e} \rhd \gamma & \text{(E.CAST)} \\
\mid & \hat{e}\ \gamma & \text{(E.APP.COE)} \\
 & & \\
\hat{d} \to & \textbf{data}\ D\ \overline{\alpha} = \overline{\mid \exists \overline{\beta}\ .\ C\ \overline{\gamma}\ \overline{\hat{\tau}}} & \\
 & & \\
\hat{\tau} \to & \tau \setminus \{\doteq\Rightarrow\} & \\
\mid & \gamma \to \hat{\tau} & \\
 & & \\
\hat{p} \to & p & \\
\mid & \hat{p}\ \gamma & \text{(P.APP.COE)}
\end{array}
$$

$$
\begin{array}{lll}
\gamma \in \mathit{Coercion} & & \\
\to & x & \text{(C.VAR)} \\
\mid & \gamma\ \gamma & \text{(C.APP)} \\
\mid & \hat{\tau} & \text{(C.REFL)} \\
\mid & \mathtt{sym}\ \gamma & \text{(C.SYM)} \\
\mid & \gamma \circ \gamma & \text{(C.TRANS)} \\
\mid & \mathtt{left}\ \gamma & \text{(C.LEFT)} \\
\mid & \mathtt{right}\ \gamma & \text{(C.RIGHT)} \\
\mid & \forall \alpha\ .\ \gamma & \text{(C.UNIV)} \\
\mid & \gamma @ \hat{\tau} & \text{(C.INST)} \\
 & & \\
\Gamma \mid & x \mapsto \tau \doteq \tau, \Gamma &
\end{array}
$$

Figure 5.6. Syntax of the target language

$$\boxed{\Gamma \vdash_e e : \tau \rightsquigarrow \hat{e}}$$

$$\frac{\Gamma \Vdash \tau_i^l \doteq \tau_i^r \rightsquigarrow \gamma \qquad \Gamma \vdash_e e : (\overline{\tau^l \doteq \tau^r}) \Rightarrow \tau \rightsquigarrow \hat{e}}{\Gamma \vdash_e e : \tau \rightsquigarrow \hat{e}\ \gamma}\ \text{E.APP.EQS}_T$$

$$\frac{\Gamma \vdash_e e : \overline{[v := \tau]}\tau' \rightsquigarrow \hat{e} \qquad \Gamma \Vdash \tau_i \doteq v_i \rightsquigarrow \gamma_i \qquad \gamma = \textbf{lift}\ \overline{\gamma}}{\Gamma \vdash_e e : \tau' \rightsquigarrow \hat{e} \rhd \gamma}\ \text{E.COERCE}_T$$

Figure 5.7. Expression type rules (T)

Rule E.APP.EQS states that if an proof of equality is expected for the source language, that this proof as coercion γ is passed explicitly as parameter in the target language.

With Rule E.COERCE, a type constants v deep inside τ' are converted. The small coercions $\overline{\gamma}$ for these type constants need to be combined into one coercion that operates on the entire τ', by adding the appropriate amount of coercion applications, universal abstractions and instantiations, and reflexive-coercions. For reasons of space, we hide this triviality behind the function **lift**.

$$\boxed{\Gamma \Vdash \tau^l \doteq \tau^r \rightsquigarrow \gamma}$$

$$\frac{\Gamma \Vdash \tau^r \doteq \tau^l \rightsquigarrow \gamma}{\Gamma \Vdash \tau^l \doteq \tau^r \rightsquigarrow \mathtt{sym}\ \gamma}\ \text{E.SYM}_T \qquad \frac{\Gamma \Vdash \tau^l \doteq \tau^a \rightsquigarrow \gamma^1 \quad \Gamma \Vdash \tau^a \doteq \tau^r \rightsquigarrow \gamma^2}{\Gamma \Vdash \tau^l \doteq \tau^r \rightsquigarrow \gamma^2 \circ \gamma^1}\ \text{E.TRANS}_T$$

$$\frac{\Gamma\,(x) = (\tau^l \doteq \tau^r)}{\Gamma \Vdash \tau^l \doteq \tau^r \rightsquigarrow x}\ \text{E.ASSUME}_T \qquad \frac{\Gamma \Vdash v_i \doteq \tau_i \rightsquigarrow \gamma_i \quad \gamma = \mathbf{lift}\ \overline{\gamma} \quad \Gamma \Vdash [\overline{v := \tau}]\ \tau^a \doteq \tau^b \rightsquigarrow \gamma'^l}{\Gamma \Vdash \tau^a \doteq \tau^b \rightsquigarrow \gamma'^l \circ \gamma}\ \text{E.CONGR}_T$$

$$\frac{\Gamma \Vdash \tau\,[\overline{\tau^a}] \doteq \tau\,[\overline{\tau^b}] \rightsquigarrow \gamma \quad \overline{\gamma} = \mathbf{decompose}\ \gamma}{\Gamma \Vdash \tau_i^a \doteq \tau_i^b \rightsquigarrow \gamma_i}\ \text{E.SUBSUME}_T$$

Figure 5.8. Entailment rules (T)

The actual construction of the coercion is a side effect of using the entailment rules of Figure 5.8 to proof an equality.

- The translation for pattern matches translates each assumption $\tau^l \doteq \tau^r$ of a constructor to an explicit match on a coercion of type $\tau^l \sim \tau^r$, binding this coercion to some unique identifier x and adding this binding to the environment. Then, entailment rule E.ASSUME lookups this binding in the environment and refers to it as the coercion x.

- The coercion for the congruence rule may only replace some type constants somewhere deep inside the type structure. Again, **lift** is used for the construction of the full coercion.

- The subsumption rule decomposes a coercion in small coercions, by adding the appropriate amount of `left` and `right` coercions in front of them. We

omit rules for this decomposition and hide this triviality behind the **decompose** function.

5.6 CONCLUSION

We described a translation of GADTs to an explicitly typed system with qualified types (Section 5.5) with equality constraints as qualifiers. The type system comes basically for free, since dealing with assumptions and insertion of evidence is already done by the system for qualified types. Adding GADTs to such a language boils down to implementing an entailment relation on equality constraints. To prevent having entailment rules for each construct in the type language, we used an auxiliary relation to construct coercions based on the structure of types.

To validate our work, we added support for GADTs to the Essential Haskell compiler [3], using Constraint Handling Rules [5] to implement the entailment rules. The EH language has a rich type system (higher ranked types, impredicative types, existential types, many type class extensions, scoped instances, polymorphic kinds, extensible records). The implementation is orthogonal to all these extensions.

5.6.1 Future work

The obvious future work is to describe our implementation in terms of this specification. Second to that, we can now describe implementation issues separately, such as the propagation of type information, and the implementation of entailment. For example, the following issues can now be described without having to give an implementation for other aspects of a GADT implementation:

The construction of equality proofs can be expensive, especially when big types with many variables are involved. There are two potential directions for optimization. The solver uses a trie-structure to select candidate CHRs and is optimized to deal with a great number of very specific CHR rules. By generating many but specialized versions of the CHR rules based on the GADT declarations, we hope to reduce the time it takes to select the next applicable rule. Furthermore, there can be several applicable rules to choose from during a solve step. At the moment, the choice is non-deterministic, but more advanced heuristics are definable in the CHR framework (for example, to make the symmetry rule the least attractive choice). These optimizations can speed up the time it takes to construct an equality proof considerably.

A succeeding pattern match witnesses that the type equalities hold. However, in the presence of irrefutable patterns, the pattern match may not have taken place. One way to deal with this is to only assume the additional type equalities when the pattern match is not inside an irrefutable pattern. This approach is taken by the Haskell compiler GHC for example. However, since we formulated GADTs in terms of qualified types and have the facilities for evidence generation at our disposal, we can do better: generate coercion functions that force evaluation of the irrefutable pattern when a coerced value is evaluated which needed the corre-

sponding assumption. Proper heuristics need to be defined to minimize the forcing of evaluations.

Acknowledgements We thank the anonymous referees for their comments; the student paper feedback report in particular. Special thanks to Lucília Camarao from Universidade Federal de Ouro Preto, Brazil, for her support just before the first submission deadline.

This work was supported in part by Microsoft Research through its European PhD Scholarship Programme.

REFERENCES

[1] A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37, pages 157–166. ACM Press, September 2002.

[2] A. I. Baars and S. D. Swierstra. Typed transformations of typed abstract syntax. http://www.cs.uu.nl/wiki/Center/TTTAS, 4 2008.

[3] A. Dijkstra. EHC Web. `http://www.cs.uu.nl/wiki/Ehc/WebHome`, 2004.

[4] A. Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

[5] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[6] S. L. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *ICFP*, pages 50–61, 2006.

[7] S. L. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[8] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *POPL*, pages 232–244, 2006.

[9] P. J. Stuckey and M. Sulzmann. Type inference for guarded recursive data types. *CoRR*, abs/cs/0507037, 2005.

[10] M. Sulzmann, M. M. T. Chakravarty, S. L. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI*, pages 53–66, 2007.

[11] M. Sulzmann, J. Wazny, and P. J. Stuckey. A framework for extended algebraic data types. In *FLOPS*, pages 47–64, 2006.

[12] D. Vytiniotis, S. Weirich, and S. L. Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In J. H. Reppy and J. L. Lawall, editors, *ICFP*, pages 251–262. ACM, 2006.

[13] J. R. Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, The university of Melbourne, January 2006.