Higher-Order Symb Comput (2010) 23:145–166
DOI 10.1007/s10990-011-9065-0

A lean specification for GADTs: system F with first-class equality proofs

Arie Middelkoop · Atze Dijkstra · S. Doaitse Swierstra

Published online: 23 July 2011

Abstract Generalized Algebraic Data Types are a generalization of Algebraic Data Types with additional type equality constraints. These found their use in many functional programs, including the development of embedded domain specific programming languages and generic programming.

Recently, several authors published novel inference algorithms and corresponding type system specifications. These approaches tend to be more algorithmic than declarative in nature, and tied to a given compiler infrastructure. This results in complex specifications. For a language implementor, adopting such a complex approach is hard, due to conflicting infrastructure and language features. Similarly, type inference is difficult to comprehend for a programmer when the specification is complex.

To make the integration of GADTs in languages easier, we thus need a more orthogonal specification. We present an orthogonal specification for GADTs: the language System $F_{\sim}$, consisting of System F augmented with first-class equality proofs. This specification exploits the Church encoding of data types to describe GADT matches in terms of conventional lambda abstractions.

Keywords Type system · GADT · System F · Equality proof

1 Introduction

Generalized Algebraic Data Types (GADTs) allow for additional equalities to hold between types of a data constructor. These equalities must hold when constructing a value with this data constructor. When a match against this constructor succeeds in a pattern match, these

A. Middelkoop · A. Dijkstra · S.D. Swierstra (✉)
Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, Netherlands
e-mail: doaitse@cs.uu.nl

A. Middelkoop
e-mail: ariem@cs.uu.nl

A. Dijkstra
e-mail: atze@cs.uu.nl

equalities hold between the types of the constructor, and may be used to coerce the type of an expression to an equivalent type.

This particular feature found many application areas. Many Haskell-related projects make use of GADTs, such as a diversity of generic programming approaches [6], and transformations on typed abstract syntax for the implementation of domain-specific languages [2]. Therefore, we added support for GADTs in UHC [5], the Haskell compiler that we develop at Universiteit Utrecht.

Dijkstra [4] describes the implementation of UHC as a combination of many specifications. Similarly, we wanted to add GADTs to UHC by taking a specification as guidance. However, it was not straightforward to match existing specifications to UHC's infrastructure, for the following reasons:

- GADT pattern matching is defined in terms of case-expressions. In a programming language such as Haskell, there are several other ways to pattern match, such as in let-bindings and list-comprehensions. Furthermore, Haskell has a concept of lazy pattern matching (irrefutable patterns). It is not directly clear if GADT matching is allowed in these situations, and what the side conditions are.
- For many approaches, it turns out that the effectiveness of inference for GADTs depends on what type information can be derived a priori from type annotations given by the programmer (Sect. 6). Specifications for inference of GADTs use a variety of techniques such as shape inference to describe that process. The implementation of such techniques crosscut the type inference implementation of the compiler, thus we are reluctant to incorporate them, unless absolutely necessary. Additionally, UHC already analyzes a program to derive type information from type signatures. Since the above techniques are intertwined with the specifications, it is not directly clear how to compare our infrastructure with the infrastructure used in the aforementioned specifications.

In this paper, we present a specification for GADT inference that addresses the above points, and has the following distinguishing features:

- We introduce a variation of System F, named System $F_\sim$, extended with equality assumptions. In contrast to other GADT specifications, System $F_\sim$ does not have syntax for data types or case expressions. These are redundant in our specification, because we exploit the folklore Church encoding of data types as conventional lambda expressions. The result is an abstract and concise description without the redundant syntax, while it is on the other hand more general, because it (necessarily) answers how to treat arbitrary pattern matching, including let-bindings and irrefutable patterns.
- Our specification is orthogonal to type inference infrastructure. Like System F, System $F_\sim$ is explicitly typed. These explicit types represent the type information that can be derived using analysis on type signatures and conventional type inference. Our specification thus focusses in isolation on how to infer type coercions.

We thus claim that our specification is *lean*: it is concise because it does not require the notion of data types and case expressions, and abstract because it is orthogonal to infrastructure (in particular regarding type annotation analysis). However, our specification only specifies when explicit type information is needed, not when type annotations may be omitted, which depends largely on the effectiveness of the algorithms we abstracted from.

In an earlier version of this paper [9], we presented this work in terms of an extension of System F_A [18], using explicit syntax for data types and case expressions. The encoding in System $F_\sim$ is significantly less complex.

Roadmap We introduce GADTs in Sect. 2, and motivate the design choices for our specification in Sect. 3. We formally define System $F_{\sim}$ and its type system in Sect. 4. In Sect. 5, we define the semantics of System $F_{\sim}$ via 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. [18]. The relation to various other approaches, we discuss in Sect. 6. Finally, in Sect. 7, we briefly mention our experiences with the integration of GADTs in the UHC.

2 Introduction to GADTs

We start this section with an introduction to GADTs. GADTs, combined with the Haskell class system, form an essential ingredient for many Haskell libraries. In this section, we picked two simplified examples as illustration.

2.1 Typed abstract syntax

One popular use of GADTs is related to the embedding of domain-specific languages in such a way that Haskell's type system can be used to type check the domain-specific language.

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:

```
data Expr
  = Num Int
  | Tup  Expr Expr
```

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:

```
eval e = case e of
           Num i   → i                -- expected: Int
           Tup p q → (eval p, eval q) -- expected: (a,b)
```

We bypass this restriction imposed by the Haskell type system with 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 *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*):

```
data Expr t
  =          Num (Equal t Int)     Int
  | ∀a b . Tup  (Equal t (a, b)) (Expr a) (Expr b)
```

Baars and Swierstra [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:

$$
\begin{array}{ll}
\mathit{coerce} & :: \mathit{Equal}\ a\ b \to a \to b \\
\mathit{sym} & :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ b\ a \\
\mathit{refl} & :: \mathit{Equal}\ a\ a \\
\mathit{trans} & :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ b\ c \to \mathit{Equal}\ a\ c \\
\mathit{congr} & :: \mathit{Equal}\ a\ b \to \mathit{Equal}\ (f\ a)\ (f\ b) \quad \text{-- has a more general signature} \\
\mathit{subsum} & :: \mathit{Equal}\ (f\ a)\ (f\ b) \to \mathit{Equal}\ a\ b \quad \text{-- than written here}
\end{array}
$$

A non-bottom value with the type $\mathit{Equal}\ \tau_1\ \tau_2$ is a proof that the types τ_1 and τ_2 are equal. The *coerce* function applied to such a proof is technically the identity function.

As a hypothetical example, consider the proof p_1 out of assumptions a_1 and a_2:

$$
\begin{array}{l}
a_1 :: \mathit{Equal}\ v_1\ (\mathit{Int}, v_2) \\
a_2 :: \mathit{Equal}\ v_1\ (v_3, \mathit{v4})
\end{array}
$$

$$
\begin{array}{l}
p_1 :: \mathit{Equal}\ (\mathit{Int}, v_2)\ (v_3, \mathit{v4}) \\
p_1 = \mathit{trans}\ (\mathit{sym}\ a_1)\ a_2
\end{array}
$$

$$
\begin{array}{l}
p_2 :: \mathit{Equal}\ (\mathit{Int}, \mathit{v4})\ (v_3, v_2) \\
p_2 = \ldots \mathit{congr} \ldots \mathit{subsum} \ldots
\end{array}
$$

We continue with the running example in this section and modify the *eval* function such that the case alternatives have the same type, namely the t in $\mathit{Expr}\ t$:

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

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

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

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.

GHC, the mainstream compiler for Haskell, has built-in support for GADTs. It offers two ways of writing GADTs. One can use qualified type notation, which resembles the example above. Instead of an additional field of the type *Equal*, we write an equality constraint:

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

Alternatively, one can use special notation for GADTs:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr}\ t\ \textbf{where} \\
\quad \mathit{Num} :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int} \\
\quad \mathit{Tup} :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b)
\end{array}
$$

These two forms of notation are interchangeable. To convert from the above qualified-type notation to GADT notation, turn each equality constraint into a substitution and apply it

exhaustively to the type signature of the constructor. The other way around, given a type signature of the constructor, take the result type as written (e.g. $Expr\ (a, b)$), and match it against the actual result type ($Expr\ t$).

Proofs do not have to be written manually. These are automatically constructed by GHC. The *eval* function can simply be written using either function alternatives or a case expression:

$$
\begin{aligned}
&eval :: Expr\ t \to t \\
&eval\ (Num\ i) \quad = i \\
&eval\ (Tup\ p\ q) = (eval\ p, eval\ q)
\end{aligned}
$$

GHC (version 6.12.1) requires the type signature to be given. We discuss in Sect. 3.2 if this type signature could be inferred and if it is desirable to do so.

2.2 Generic programming

Cheney and Hinze [3] show how GADTs, called Phantom Types in their work, can be used to implement generic functions that work for many data types. The idea is to have a representation of types as a first class value, then use this representation to navigate generically over a particular value:

$$
\begin{aligned}
&\textbf{data}\ Rep\ t\ \textbf{where} \\
&\quad RInt \quad :: Rep\ Int \\
&\quad RChar :: Rep\ Char \\
&\quad RList \quad :: Rep\ a \to Rep\ [a] \\
&\quad RTup \quad :: Rep\ a \to Rep\ b \to Rep\ (a, b) \\
&x :: [(Int, Char)] \qquad r :: Rep\ [(Int, Char)] \\
&x = [(3, \text{'4'})] \qquad r = RList\ (RTup\ RInt\ RChar)
\end{aligned}
$$

In this example, x is a value of some type, and r is a value of a representation of that type.

The following function, for example, traverses a value of any type for which we have a representation, and increments all integers with one:

$$
\begin{aligned}
&replace :: Rep\ t \to t \to t \\
&replace\ RInt \quad\quad x \quad\quad = x + 1 \\
&replace\ RChar \quad\quad c \quad\quad = c \\
&replace\ (RList\ r) \quad xs \quad\quad = map\ (replace\ r)\ xs \\
&replace\ (RTup\ a\ b)\ (p, q) = (replace\ a\ p, replace\ b\ q)
\end{aligned}
$$

Values of $Rep\ t$ can typically be obtained automatically at *replace*'s call site using Haskell's class system.

Cheney and Hinze [3] continue from here, and define a data type for the sum of products representation of data types, and use this representation to define combinators to express generic traversals over arbitrary data types for which there is a representation.

3 Design rationale

In this section, we discuss the design decisions of System $F_\sim$. We are liberal concerning syntax in this section, without loss of generality. In Sect. 4.1, we treat System $F_\sim$ formally.

3.1 Church encoding of data types

In other GADT specifications and algorithms, GADT matches are described in combination with case-expressions. In many type system descriptions, however, data types and case expressions are not part of the language, because these are implied by using a Church encoding or Mogensen-Scott encoding [10] of the data types. In our specification, we take a similar approach and consider a Church encoding of GADTs.

Church encoding of ADTs As a prelude, we informally sketch the Church encoding of algebraic data types.

The easiest example to start with are Church booleans. The constructors *True* and *False* can be represented as functions that take two continuation-expressions as parameters and return the appropriate one:

$$
\begin{aligned}
&\textbf{type } \mathit{Bool}' = \forall\alpha.\alpha \to \alpha \to \alpha\\
&\mathit{mkTrue}, \mathit{mkFalse} :: \mathit{Bool}'\\
&\mathit{mkTrue}\ \ t\, f = t\\
&\mathit{mkFalse}\, t\, f = f
\end{aligned}
$$

The functions *mkTrue* and *mkFalse* can be used instead of the original constructors. To pattern match against such a constructor, we give it the continuations. For example, the following function:

$$
\begin{aligned}
&\mathit{ifThenElse} :: \mathit{Bool} \to a \to a \to a\\
&\mathit{ifThenElse}\, b\, f\, g\\
&\quad = \textbf{case } b \textbf{ of}\\
&\qquad \mathit{True}\ \ \to f\\
&\qquad \mathit{False} \to g
\end{aligned}
$$

can be encoded as:

$$
\begin{aligned}
&\mathit{ifThenElse} :: \mathit{Bool}' \to a \to a \to a\\
&\mathit{ifThenElse}\, b\, f\, g = b\, f\, g
\end{aligned}
$$

In general, the Church encoding of a data constructor is a function that takes (Church encoded) values for each of its fields, and several continuation functions, one for each constructor. The values are passed on to the appropriate continuation function. For example, given the following data type:

$$
\begin{aligned}
&\textbf{data } D\\
&\quad = \mathit{C1}\ \mathit{Int}\\
&\quad \mid \mathit{C2}\ D
\end{aligned}
$$

We introduce a (recursive) type D' for Church-encoded Ds, and constructor functions *mkC1* and *mkC2*:

$$
\begin{aligned}
&\textbf{type } D' = \forall r.(\mathit{Int} \to r) \to (D' \to r) \to r\\
&\mathit{mkC1} :: \mathit{Int} \to D'\\
&\mathit{mkC1}\, x\, f1\ \_ = f1\, x
\end{aligned}
$$

$$mkC2 :: D' \to D'$$
$$mlC2\ r\ \_f2\ = f2\ r$$

In this example, we assume that built-in types are not encoded. As a technical detail, we require a number of built-in types in order to map recursive types to System F [19].

When we pattern match in a case-expression, we get a value of type D', and parameterize it with functions that deal with each of the cases. For example, given the following case expression:

$$\begin{array}{l}\textbf{case}\ x\ \textbf{of}\\ \quad C1\ y \to y + 1\\ \quad C2\ \_ \to \bot\end{array}$$

In the Church encoding, this corresponds to:

$$x\ (\lambda y.y + 1)\ (\backslash\_.\bot)$$

Likewise, we can look at encodings of other forms of pattern matching. A let-binding can be encoded by means of a lambda and application:

$$\textbf{let}\ (C1\ y) = x\ \textbf{in}\ y + 1$$

The pattern match takes place when y is evaluated. We can encode such a let-expression as a lambda, if we assume that patterns in a lambda match lazily:

$$(\lambda y.y + 1)\ ((\lambda(C1\ y).y)\ x)$$

As continuation functions we take $\bot$, except for the continuation corresponding to the data constructor matched on.

$$(\lambda y.y + 1)\ ((\lambda d.d\ (\lambda y.y)\ \bot)\ x)$$

If a let-binding involves multiple variables, we use the Church encoding of tuples.

Haskell has irrefutable (or, lazy) patterns. A match against such pattern always succeeds, and is actually only performed when values of variables that are bound by the pattern are needed. In the following example, the pattern match against $C1$ is only performed when the value of y is needed:

$$\begin{array}{l}\textbf{case}\ x\ \textbf{of}\\ \quad \sim(C1\ y) \to y + 1\end{array}$$

We can write the irrefutable pattern with a let-binding.

$$\begin{array}{l}\textbf{case}\ x\ \textbf{of}\\ \quad z \to \textbf{let}\ (C1\ y) = z\\ \qquad\quad \textbf{in}\ y + 1\end{array}$$

This expression can be encoded again in a similar way as above.

Church encoding of GADTs The above approach has as advantage that we only need to consider lambda applications in our specification, and from it, the behavior for case-expressions, let-bindings, etc. can be derived.

In the previous section, we showed that the mechanism underlying GADTs is the construction and application of equality proofs. These proofs are constructed and stored as fields in the constructor, such that they can be taken out when the match succeeds. In the Church encoding, a field has become a lambda. We thus introduce two new forms of expressions. An expression $\tau \sim \sigma$ that builds a proof of the equality between τ and σ, and an expression $\lambda(\tau \sim \sigma).e$ that matches against such a proof and allows it to be used for coercions in e.

For example, for a GADT (written using qualified-type notation):

$$\begin{aligned}&\textbf{data}\ \mathit{Rep}\ a\\ &\quad = (a \sim \mathit{Int}) \Rightarrow \mathit{RInt}\end{aligned}$$

The encoding for *RInt* takes an equality proof, and passes it on to the continuation:

$$\begin{aligned}&\textbf{type}\ \mathit{Rep}'\ a = \forall r.((a \sim \mathit{Int}) \to r) \to r\\ &\mathit{mkRInt} :: (a \sim \mathit{Int}) \to \mathit{Rep}'\ a\\ &\mathit{mkRInt} = \lambda(a \sim \mathit{Int}).\lambda f.f\ (a \sim \mathit{Int})\end{aligned}$$

In order to construct a value of $\mathit{Rep}'\ a$ using *mkRInt*, we first need to pass a proof that $a \sim \mathit{Int}$.

Lazy equalities We choose the equality-lambda to bind lazily. The following example gives an explanation. The pattern occurs in a let-expression and does not define any variables, which essentially means that it will never be evaluated.

$$\begin{aligned}&f :: \mathit{Rep}\ a \to a \to \mathit{Int}\\ &f\ x\ y = \textbf{let}\ \mathit{RInt} = x\\ &\qquad\quad \textbf{in}\ 3\end{aligned}$$

In the encoding, x is only evaluated if its equality proof is needed when equalities bind lazily:

$$\lambda(x :: \mathit{Rep}\ a).\lambda(y :: a).(\lambda(a \sim \mathit{Int}).3 :: \mathit{Int})\ (x\ (\lambda(a \sim \mathit{Int}).(a \sim \mathit{Int})))$$

However, if we replace the 3 by y, the equality proof is needed to coerce the type a to *Int*. This subsequently leads to evaluation of x.

It is, however, undesirable that the construction of an equality proof influences the evaluation of the program. With our encoding, we have a model to reason about these effects. For example, it is straightforward to verify that the encoding of GADT matches in a case expression, and the encoding of GADT matches in a lambda, do not influence the evaluation of the program.

3.2 Type inference

Above, we gave type signatures to all expressions involving GADTs. Several authors showed that there are situations where coercions can be inferred without an accompanying type annotation. So far, such approaches are not predictable: it is not intuitive when an annotation may be omitted or when it is obligatory. Several examples in this section illustrate the difficulties regarding inference. We therefore decided to allow coercions only when directed to via a type annotation, and describe how this shows up in the specification.

The main problem regarding GADT inference is that without a type annotation, it is not clear whether or not to coerce a type. In the following example, the pattern match on *Num* brings an equality on *Int* in scope, which may be applied to coerce the type of 3:

$$\begin{array}{l}\textbf{data}\ \mathit{Expr}\ t\ \textbf{where}\\ \quad \mathit{Num}\ \ :: \mathit{Int} \to \mathit{Expr}\ \mathit{Int}\\ \quad \mathit{Tuple} :: \mathit{Expr}\ a \to \mathit{Expr}\ b \to \mathit{Expr}\ (a, b)\\ \mathit{eval}\ (\mathit{Num}\ x) = x\end{array}$$

There are several types possible for *eval*:

$$\begin{array}{l}(1)\ \mathit{eval} :: \forall t\ .\ \mathit{Expr}\ t \to t\\ (2)\ \mathit{eval} :: \mathit{Expr}\ \mathit{Int} \to \mathit{Int}\\ (3)\ \mathit{eval} :: \forall t\ .\ \mathit{Expr}\ t \to \mathit{Int}\end{array}$$

The second type seems appropriate in this case. However, the first type is more general than the second, and the first and third are incomparable. We cannot choose between the first and third type, unless we know precisely how *eval* is to be used. Many inference approaches analyze the usages of such a function to make a choice. When usage of such a function changes due to modifications of the programmer, this may affect the choice, and cause type errors in other parts of the program. This leads to an unpredictable type inference. We choose only to apply equalities when directed to via a type annotation. In the above case, no type annotation is present, we do not apply the equality, which results in type (3).

In the above example, the usage of x in the body seems to suggest a relation to type t in *Expr t*. However, the following example demonstrates that this is not a good criterium, because there are also expressions that have a coercible type, aside from variables occurring in the pattern:

$$\begin{array}{l}\mathit{flex}\ \mathit{True}\ (\mathit{Num}\ x)\ \ = x\\ \mathit{flex}\ \mathit{False}\ (\mathit{Num}\ x) = 3\end{array}$$

Multiple pattern matches pose additional challenges. In the following example, there are two possible equalities on *Int* to choose from:

$$\mathit{choose}\ (\mathit{Num}\ x)\ (\mathit{Num}\ y) = 3$$

There are now many alternative types possible, including:

$$\begin{array}{l}\mathit{choose} :: \forall t\ s\ .\ \mathit{Expr}\ t \to \mathit{Expr}\ s \to t\\ \mathit{choose} :: \forall t\ s\ .\ \mathit{Expr}\ t \to \mathit{Expr}\ s \to s\\ \mathit{choose} :: \forall t\ s\ .\ \mathit{Expr}\ t \to \mathit{Expr}\ s \to \mathit{Int}\end{array}$$

During inference of the body of *choose*, the problem boils down to making a choice between either coercion to a Skolem constant such as s or t, or not coerce types at all. Again, we refrain from applying an equality unless directed to by a type annotation, and thus end up with the last type.

When there are multiple function or case alternatives, there is often no choice. The equalities have to be applied to have branches with equal types.

$$\begin{array}{l}\mathit{eval}\ (\mathit{Num}\ x)\ \ = x\\ \mathit{eval}\ (\mathit{Tup}\ a\ b) = (a, b)\end{array}$$

$$
\begin{array}{lll}
e, f, g, a \in \mathit{Expr} & & \\
\quad ::= x & \text{(E.VAR)} & \\
\quad \mid \tau \sim \sigma & \text{(E.EQ)} & \\
\quad \mid \lambda x.e & \text{(E.LAM.ABS)} & \\
\quad \mid \lambda \alpha.e & \text{(E.UNIV.ABS)} & \\
\quad \mid \lambda(\tau \sim \sigma).e & \text{(E.EQ.ABS)} & \\
\quad \mid f\ e & \text{(E.APP.EXPR)} & \\
\quad \mid f\ \tau & \text{(E.APP.UNIV)} &
\end{array}
$$

$$
\begin{array}{lll}
\tau, \rho, \varrho, \sigma \in \mathit{Type} & & \\
\quad ::= \alpha & \text{(T.VAR)} & \\
\quad \mid \sigma \to \tau & \text{(T.ARR)} & \\
\quad \mid \forall \alpha.\tau & \text{(T.FORALL)} & \\
\quad \mid \tau \sim \sigma & \text{(T.EQS)} &
\end{array}
$$

$$
\begin{array}{lll}
\Gamma \in \mathit{Env} & & \\
\quad ::= \Gamma, x :: \tau, & \text{(E.TY)} & \\
\quad \mid \Gamma, \alpha & \text{(E.VAR)} &
\end{array}
$$

$$
\begin{array}{l}
x, y \in \mathit{Ident} \\
\alpha, \beta \in \mathit{TyIdent}
\end{array}
$$

Fig. 1 Syntax of System $F_\sim$

We still require a type annotation. The design rationale is here that adding additional cases to a function without changing the existing cases, should only make a type less specific. In the above case, adding the *Tup* alternative causes the type of *eval* to change to an incomparable type, which again makes type inference hard to predict from the perspective of the programmer. Also, we point out that in the situation that multiple function or case alternatives are written, the overhead of the annotation (expressed for example in terms of lines of code) decreases.

In our specification, we abstract from the type inference algorithm. Moreover, we require type inference to be completed before the inference of coercions. The language System $F_\sim$, based on System F, is explicitly typed. These explicit types represent the types derived from type annotations and those inferred. This makes it unambiguous where to find the locations where a coercion is needed: at those locations where the actual type does not match the expected type (Sect. 4.4). What remains is to specify how coercions are inferred, and that we make explicit in the specification.

4 Specification

In this section, we present our specification for GADT inference. We start with the language System $F_\sim$ in Sect. 4.1, present System $F_\sim$'s typing rules in Sect. 4.2, and show how to deal with equality proofs in Sect. 4.3.

4.1 System $F_\sim$

Figure 1 lists the syntax of System $F_\sim$, which consists of System F with the addition of equality proofs. These additions consist of:

- The abstraction $\lambda(\tau \sim \sigma).e$ matches against an equality proof for $\tau \sim \sigma$, and brings it in scope by adding it to the environment. The equalities in scope are called equality assumptions, and can be used in equality proofs. The process of constructing a proof is fully automatic.
- The proof expression $(\tau \sim \sigma)$ constructs an equality $\tau \sim \sigma$.

– The type language contains a type for equality proofs. Furthermore, we assume a number of primitive types to be present in the type language, such as *Int*.

Environments record Skolem constants introduced by type abstraction, and types for identifiers. Since we present equality proofs as conventional types, these can be stored in the environment as well, when we give them a name.

As illustration, we encode the following fragment of a program that generically increments integers in data types:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Rep}\ \alpha\ \textbf{where} \\
\quad \mathit{RInt} :: \mathit{Rep}\ \mathit{Int} \\
\mathit{inc} :: \mathit{Rep}\ \alpha \to \alpha \to \mathit{Int} \\
\mathit{inc} = \lambda r.\lambda x.\textbf{case}\ r\ \textbf{of} \\
\qquad\qquad \mathit{RInt} \to x + 1 \\
z :: \mathit{Int} \\
z = \mathit{inc}\ \mathit{RInt}\ 3
\end{array}
$$

As an intermediate step, we write:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Rep}'\ \alpha = \forall \beta.((\alpha \sim \mathit{Int}) \to \beta) \to \beta \\
\mathit{mkRInt} :: \alpha \sim \mathit{Int} \to \mathit{Rep}'\ \alpha \\
\mathit{mkRInt} = \lambda \mathit{eq}.\lambda f.f\ \mathit{eq} \\
\mathit{inc} :: \mathit{Rep}'\ \alpha \to \alpha \to \mathit{Int} \\
\mathit{inc} = \lambda r.\lambda x.r\ (\lambda(\alpha \sim \mathit{Int}).x + 1)) \\
z :: \mathit{Int} \\
z = \mathit{inc}\ (\mathit{mkRInt}\ (\mathit{Int} \sim \mathit{Int}))\ 3
\end{array}
$$

Finally, expressed in System $F_\sim$:

$$
\begin{array}{l}
\quad \text{-- } \mathit{mkRInt} \\
\Lambda\alpha.\lambda(\mathit{eq} :: \alpha \sim \mathit{Int}) \\
\quad .\lambda(f :: \forall \beta.((\alpha \sim \mathit{Int}) \to \beta) \to \beta) \\
\quad .f\ \alpha\ \mathit{eq} \\
\quad \text{-- } \mathit{inc} \\
\Lambda\alpha.\lambda(r :: \forall \beta.((\alpha \sim \mathit{Int}) \to \beta) \to \beta) \\
\quad .\lambda(x :: \alpha) \\
\quad .r\ \mathit{Int}\ (\lambda(\alpha \sim \mathit{Int}).(+)\ x\ 1) \\
\quad \text{-- } z \\
\mathit{inc}\ \mathit{Int}\ (\mathit{mkRInt}\ \mathit{Int}\ (\mathit{Int} \sim \mathit{Int}))\ 3
\end{array}
$$

Note that the type of x in *inc* is α, and is required to have type *Int*. In System F, this expression is not correctly typed. It has a valid type in System $F_\sim$'s type system, which we discuss below.

4.2 Type rules

The typing relation (rules in Fig. 2) states that in environment Γ, the expression e has type τ. These rules are syntax directed, except for rule COERCE. The idea is that the shape of the expression determines which rules to apply, and we resort to rule COERCE when there is a mismatch between the types. We illustrate this process briefly with inference for the *inc* example of the previous section, then explain the rules in more detail.

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

$$\frac{x :: \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{VAR} \qquad \frac{\Gamma \vdash e : \sigma \quad \Gamma \Vdash \sigma \sim \tau \quad \mathtt{ftv}\,\tau \subseteq \mathtt{ftv}\,\Gamma}{\Gamma \vdash e : \tau}\ \text{COERCE} \qquad \frac{\Gamma \Vdash \tau \sim \sigma \quad \mathtt{ftv}\,\tau \subseteq \mathtt{ftv}\,\Gamma \quad \mathtt{ftv}\,\sigma \subseteq \mathtt{ftv}\,\Gamma}{\Gamma \vdash \tau \sim \sigma : \tau \sim \sigma}\ \text{EQ}$$

$$\frac{\Gamma \vdash f : \sigma \to \tau \quad \Gamma \vdash e : \sigma}{\Gamma \vdash f\, e : \tau}\ \text{APP.EXPR} \qquad \frac{\Gamma \vdash f : \sigma \to \tau \quad \mathtt{ftv}\,\sigma \subseteq \mathtt{ftv}\,\Gamma}{\Gamma \vdash f\, \sigma : \tau}\ \text{APP.TY}$$

$$\frac{\Gamma, x :: \sigma \vdash e : \tau}{\Gamma \vdash \lambda(x :: \sigma).e : \sigma \to \tau}\ \text{LAM.EXPR} \qquad \frac{\Gamma, \alpha \vdash e : \tau \quad \alpha \notin \mathtt{ftv}\,\Gamma}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \text{LAM.TY}$$

$$\frac{\Gamma, x :: \rho \sim \sigma \vdash e : \tau \quad x \notin \Gamma}{\Gamma \vdash \lambda(\rho \sim \sigma).e : (\rho \sim \sigma) \to \tau}\ \text{LAM.EQ}$$

Fig. 2 Expression type rules

Example To type the *inc* expression (see Fig. 3), we apply first rule LAM.TY, then rule LAM.EXPR twice. The former administers the Skolem constant α in the environment, the second two the types of r and x. The type of r, we extract again using the rule VAR, then use rule APP.TY to instantiate the universally quantified β of the type of r to the result type *Int*.

To type the subexpression $(\lambda(\alpha \sim \mathit{Int}).(+)\ x\ 1)$, we use rule LAM.EQ to introduce the equality into the environment, bound to a fresh name (not important for now). The subexpression $(+)\ x\ 1$ must have type *Int*. Assuming that $(+) :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ is in the environment, this means that x must have type *Int*. However, the type we get for x by applying the VAR-rule is α. So, we apply the COERCE-rule to convert the type α to *Int*. This requires us to prove that $\Gamma \Vdash \alpha \sim \mathit{Int}$, for which we give the rules later. This is in this case not difficult, because exactly this equality we added just before into the environment.

Type rules overview Most of the rules are vanilla System F rules.

Via the `ftv` relation, we state that the types chosen in rules COERCE, EQ, and APP.TY are closed (and well-formed). The `ftv` relation is defined as:

$$\begin{aligned}
\mathtt{ftv}\,\alpha &= \{\alpha\} \\
\mathtt{ftv}\,(\tau \to \sigma) &= \mathtt{ftv}\,\tau \cup \mathtt{ftv}\,\sigma \\
\mathtt{ftv}\,(\forall\alpha.\tau) &= \mathtt{ftv}\,\tau - \{\alpha\} \\
\mathtt{ftv}\,(\tau \sim \sigma) &= \mathtt{ftv}\,\tau \cup \mathtt{ftv}\,\sigma
\end{aligned}$$

We overload `ftv` to work on environments as well:

$$\begin{aligned}
\mathtt{ftv}\,\emptyset &= \emptyset \\
\mathtt{ftv}\,(\Gamma, \alpha) &= \mathtt{ftv}\,\Gamma \cup \{\alpha\} \\
\mathtt{ftv}\,(\Gamma, x :: \tau) &= \mathtt{ftv}\,\Gamma \cup \mathtt{ftv}\,\tau
\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash r : \forall\beta.(((\alpha \sim \beta) \to \beta) \to \beta)}{\Gamma_1 \vdash r\ Int : ((\alpha \sim Int) \to Int) \to Int} \qquad \Gamma_1 \vdash \lambda(\alpha \sim Int).(+)\ x\ 1 : \ldots}{\Gamma_1 \vdash r\ Int\ (\lambda(\alpha \sim Int).(+)\ x\ 1) : Int}}{\ldots \vdash \lambda x.r\ Int\ (\lambda(\alpha \sim Int).(+)\ x\ 1) : \ldots \to Int}}{\ldots \vdash \lambda r.\lambda x.r\ Int\ (\lambda(\alpha \sim Int).(+)\ x\ 1) : \ldots \to \ldots \to Int}}{\emptyset \vdash \Lambda\alpha.\lambda r.\lambda x.r\ Int\ (\lambda(\alpha \sim Int).(+)\ x\ 1) : \forall\alpha.\ldots \to \ldots \to Int}$$

$$\dfrac{\dfrac{\dfrac{\Gamma_2 \vdash (+) : \ldots \qquad \dfrac{\Gamma_2 \vdash x : \alpha \qquad \Gamma_2 \Vdash \alpha \sim Int}{\Gamma_2 \vdash x : Int}}{\Gamma_2 \vdash (+)\ x : Int \to Int} \qquad \Gamma_2 \vdash 1 : Int}{\Gamma_2 \vdash (+)\ x\ 1 : Int}}{\Gamma_1 \vdash \lambda(\alpha \sim Int).(+)\ x\ 1 : (\alpha \sim Int) \to Int}$$

$\Gamma_0 = \emptyset, 1 :: Int, (+) :: Int \to Int \to Int$
$\Gamma_1 = \Gamma_0, \alpha, r :: \forall\beta.(((\alpha \sim \beta) \to \beta) \to \beta, x :: \alpha$
$\Gamma_2 = \Gamma_1, e :: \alpha \sim \beta$

Fig. 3 Typing Derivation of *inc*

Rules COERCE and EQ specify the construction of equality proofs. In the former, the equality proof is used to coerce a type, in the latter to pass it on as a first-class value.

In rules LAM.EXPR, LAM.TY, and LAM.EQ, we extend the environment. Extension of an environment with a binding shadows a possible previous binding with the same name. In rule LAM.EQ, we introduce an equality in the environment, using a fresh name x. This name is of consequence for the next section, but has no meaning in this section.

4.3 Type conversions

Figure 4 lists the inference rules for equality proofs. The rules REFL, SYM and TRANS correspond to the conventional rules of an equality theory. An equality assumption can be applied through rule ASSUM.

In the example of the beginning of this section, we need to prove $\Gamma \Vdash Int \sim Int$ (by means of the REFL-rule), and $\Gamma \Vdash a \sim Int$ (with the ASSUM-rule). In Sect. 2.1 we showed some examples using more involved equality proofs. Simply put, proving an equality is a matter of exhaustively applying all the rules.

To be able to apply coercions deeper into types, we have congruence and subsumption rules for each member of the type language that has a substructure. In our case, for arrows and universal quantification. We do not need nor allow coercions on equality proofs.

With congruence rules, if two types share a common structure, we only need to prove the equality between the components that differ. The other way around, with subsumption rules, we lift a proof on smaller type into a proof on bigger types.

$$\boxed{\Gamma \Vdash \tau_1 \sim \tau_2}$$

$$\Gamma \Vdash \tau \sim \tau \ \text{REFL} \qquad \frac{\Gamma \Vdash \sigma \sim \tau}{\Gamma \Vdash \tau \sim \sigma}\ \text{SYM} \qquad \frac{\Gamma \Vdash \tau \sim \rho \quad \Gamma \Vdash \rho \sim \sigma}{\Gamma \Vdash \tau \sim \sigma}\ \text{TRANS} \qquad \frac{x :: \tau \sim \sigma \in \Gamma}{\Gamma \Vdash \tau \sim \sigma}\ \text{ASSUM}$$

$$\frac{\Gamma \Vdash \sigma \sim \tau \quad \Gamma \Vdash \rho \sim \varrho}{\Gamma \Vdash \tau \to \rho \sim \sigma \to \varrho}\ \text{CON.ARR} \qquad \frac{\Gamma, \beta \Vdash \tau\,[\alpha := \beta] \sim \sigma \quad \beta \notin \mathtt{ftv}\,\tau \cup \mathtt{ftv}\,\Gamma}{\Gamma \Vdash \forall \alpha.\tau \sim \sigma}\ \text{CON.UNIV.LEFT}$$

$$\frac{\Gamma \Vdash \tau \sim \sigma\,[\beta := \alpha] \quad \beta \in \mathtt{ftv}\,\Gamma \quad \alpha \notin \mathtt{ftv}\,\sigma}{\Gamma \Vdash \tau \sim \forall \alpha.\sigma}\ \text{CON.UNIV.RIGHT} \qquad \frac{\begin{array}{c} x :: (\tau_3 \sim (\forall \alpha.\tau_4)) \in \Gamma \quad y :: (\tau_5 \sim (\forall \alpha.\tau_6)) \in \Gamma \\ \Gamma \Vdash \tau_3 \sim \tau_5 \quad x' :: \tau_4 \sim \tau_6 \Vdash \tau_1 \sim \tau_2 \quad x' \notin \Gamma \end{array}}{\Gamma \Vdash \tau_1 \sim \tau_2}\ \text{SUB.UNIV}$$

$$\frac{\begin{array}{c} x :: (\tau_3 \sim (\tau_4 \to \tau_5)) \in \Gamma \quad y :: (\tau_6 \sim (\tau_7 \to \tau_8)) \in \Gamma \\ \Gamma \Vdash \tau_3 \sim \tau_6 \quad x' :: \tau_4 \sim \tau_7, y' :: \tau_5 \sim \tau_8 \Vdash \tau_1 \sim \tau_2 \quad x', y' \notin \Gamma \end{array}}{\Gamma \Vdash \tau_1 \sim \tau_2}\ \text{SUB.ARR}$$

Fig. 4 Equality proof inference rules

In other specifications Schrijvers et al. [14], Sulzmann et al. [18], the subsumption rules e.g. on arrows are typically specified as:

$$\frac{\Gamma \Vdash \tau \to \rho \sim \sigma \to \varrho}{\Gamma \Vdash \tau \sim \sigma}\ \text{SUB.L} \qquad \frac{\Gamma \Vdash \tau \to \rho \sim \sigma \to \varrho}{\Gamma \Vdash \tau \sim \sigma}\ \text{SUB.R}$$

These rules appear simpler than the rule in our specification. However, we have objections against these rules:

- For any environment Γ, the other rules have a finite number of unique instantiations. However, when we include the above subsumption rule, arbitrary big types can be constructed. Thus the search space can be infinite. In practice, we only need to consider a finite portion of the search space to prove or disprove the equality of two types, but this rule does not force us to.
- The subsumption rule does not give us an intuition when to actually apply this rule. To construct an equality proof, one first decomposes the equality to prove using the congruence rules, then apply subsumption rules to work up to assumptions in the environment. We thus restrict the subsumption rule to assumptions in the environment.

$$
\begin{array}{lll}
\gamma \in \mathit{Coercion} & & \hat{e} \in \mathit{Expr} \\
\quad ::= x & \text{(C.VAR)} & \quad \mid e \triangleright \gamma \quad \text{(E.APP.COE)} \\
\quad \mid \gamma_1\ \gamma_2 & \text{(C.APP)} & \quad \mid \textbf{case}\ \hat{e}\ \textbf{of}\ \overline{p \to \hat{e}} \quad \text{(E.CASE)} \\
\quad \mid \tau & \text{(C.REFL)} & \\
\quad \mid \mathtt{sym}\ \gamma & \text{(C.SYM)} & p \in \mathit{Pat} \\
\quad \mid \gamma_1 \circ \gamma_2 & \text{(C.TRANS)} & \quad \mid C\ \overline{\gamma : \tau \sim \sigma}\ \overline{\alpha : \star}\ \overline{x : \rho} \quad \text{(P.CON)} \\
\quad \mid \mathtt{right}\ \gamma & \text{(C.RIGHT)} & \\
\quad \mid \mathtt{left}\ \gamma & \text{(C.LEFT)} & d \in \mathit{Decl} \\
\quad \mid \forall \alpha\ .\ \gamma & \text{(C.UNIV)} & \quad \mid \textbf{data}\ D\ \overline{a} \mid \overline{C\ \overline{\tau \sim \sigma}\ \overline{\alpha : \star}\ \overline{\rho}} \\
\quad \mid \gamma\ @\ \rho & \text{(C.INST)} &
\end{array}
$$

Fig. 5 Subset of syntax of System F_C

Our rule simply expresses that if there are two equalities in the environment, and these equalities can be shown equal on one side, then we may assume that each pairwise subtype on the other side is equal as well.

4.4 Algorithm

The specification of this section has a straightforward implementation, even combined with type inference. We first infer types of a program in the conventional way, but for each type conflict, we generate a coercion constraint. At the end of type inference for e.g. a binding group, we try to solve these coercion constraints, using an exhaustive application of the above equality rules. For those constraints that cannot be solved we generate a type error. For those we can, we generate a coercion. In the next section, we show how to generate these coercions.

5 Semantics

We define the semantics of System $F_\sim$ in terms of System F_C [18]. We introduction to System F_C, then show the important parts of the translation.

5.1 System F_C

System F_C extends System F with equality coercions. It has a built-in syntax to represent values of the *Equal* data type mentioned in the Sect. 2.1.

We limit our explanation to a simplified fragment of System F_C, which contains what we need: System F (lambda abstraction, etc.), data types, case expressions, and coercions. Figure 5 lists the extensions to System F. A data constructor C consists of three components: equality coercions, existentials, and fields, respectively, hence it is a representation of the qualified-type style of GADT constructors.

A coercion γ can be applied to an (System F_C) expression $\hat{e}$, denoted as $\hat{e} \triangleright \gamma$, which changes the type of $\hat{e}$ according to coercion. A coercion γ of type $\tau_1 \sim \tau_2$ represents a proof of the equality of τ_1 and τ_2. Figure 6 lists the typing rules of coercion terms. See [18] for the full listing.

$$\boxed{\Gamma \vdash \gamma : \tau_1 \sim \tau_2}$$

$$\frac{x :: \tau_1 \sim \tau_2 \in \Gamma}{\Gamma \vdash x : \tau_1 \sim \tau_2}\ \text{VAR} \qquad \frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \tau_3 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim \tau_4}{\Gamma \vdash \gamma_1\ \gamma_2 : \tau_1\ \tau_2 \sim \tau_3\ \tau_4}\ \text{APP} \qquad \Gamma \vdash \tau : \tau \sim \tau\ \text{REFL}$$

$$\frac{\Gamma \vdash \gamma : \tau_2 \sim \tau_1}{\Gamma \vdash \mathtt{sym}\,\gamma : \tau_1 \sim \tau_2}\ \text{SYM} \qquad \frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \tau_2 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim \tau_3}{\Gamma \vdash \gamma_1 \circ \gamma_2 : \tau_1 \sim \tau_3}\ \text{TRANS}$$

$$\frac{\Gamma \vdash \gamma : \tau_1\ \tau_2 \sim \tau_3\ \tau_4}{\Gamma \vdash \mathtt{left}\,\gamma : \tau_1 \sim \tau_3}\ \text{LEFT} \qquad \frac{\Gamma \vdash \gamma : \tau_1\ \tau_2 \sim \tau_3\ \tau_4}{\Gamma \vdash \mathtt{right}\,\gamma : \tau_2 \sim \tau_4}\ \text{RIGHT}$$

$$\frac{\Gamma \vdash \gamma : \tau_1 \sim \tau_2 \qquad \alpha \notin \mathtt{ftv}\,\Gamma}{\Gamma \vdash \forall \alpha\,.\,\gamma : \forall \alpha\,.\,\tau_1 \sim \forall \alpha\,.\,\tau_2}\ \text{FORALL} \qquad \frac{\Gamma \vdash \gamma : \forall \alpha\,.\,\tau_1 \sim \forall \beta\,.\,\tau_2}{\Gamma \vdash \gamma\,@\rho : \tau_1\,[\alpha := \rho] \sim \tau_2\,[\beta := \rho]}\ \text{INST}$$

Fig. 6 Coercion type rules

Coercion application $\gamma_1\ \gamma_2$ builds a coercion that applies γ_1 to the function part f and γ_2 to the argument part a of a type application $f\ a$. With transitivity $\gamma_2 \circ \gamma_1$, the second coercion is applied to the result of applying the first coercion. The `right`-coercion extracts the coercion of the arguments from a coercion on a type application $f\ a$.

5.2 Translation overview

We use a type-directed translation. The typing relations have the form:

$\Gamma \vdash e : \tau \rightsquigarrow \hat{e}$ -- System F_C-expr $\hat{e}$ is the translation of System $F_\sim$-expr e.
$\Gamma \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma$ -- γ is a System F_C-coercion term of type $\tau_1 \sim \tau_2$.

The translation consists of two challenges: we need to represent System $F_\sim$'s equality proofs in System F_C, and need to associate with each equality-proof rule of the previous section a well-typed System F_C-coercion term.

5.3 System $F_\sim$ expressions

The first-class equality proofs in System $F_\sim$ have no direct counterpart in System F_C. However, in System F_C, equality proofs may be stored in a data constructor. For example, for an equality proof of type $\alpha \sim \beta \to \mathit{Int}$, we can associate a data type:

$$\textbf{data}\ D\ \alpha\ \beta = C\ (\alpha \sim \beta \to \mathit{Int})$$

In general, we associate a data type $D_{key\,(\tau,\sigma)}\overline{\alpha}$ with a System $F_\sim$ proof of type $\tau \sim \sigma$ ($\overline{\alpha} = \mathtt{ftv}\ \tau \cup \mathtt{ftv}\ \sigma$), as well as a constructor $C_{key\,(\tau,\sigma)}$ storing the System F_C-coercion. We denote with $[\![\tau \sim \sigma]\!]$ the conversion from a System $F_\sim$-proof type to a System F_C data type, or data constructor according to the above procedure. Similarly, $[\![\sigma]\!]$ denotes the translation of a System $F_\sim$ type to a System F_C type, by recursively mapping each coercion type $\tau \sim \sigma$ to $D_{key\,(\tau,\sigma)}\overline{\alpha}$.

Figure 7 shows the translation rules, which are the type rules of the previous section extended with the resulting $\hat{e}$ expression.

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

$$\frac{x :: \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow x}\ \text{VAR} \qquad \frac{\Gamma \vdash e : \sigma \rightsquigarrow \hat{e} \quad \Gamma \Vdash \sigma \sim \tau \rightsquigarrow \gamma \quad \mathtt{ftv}\,\tau \subseteq \mathtt{ftv}\,\Gamma}{\Gamma \vdash e : \tau \rightsquigarrow \hat{e} \rhd \gamma}\ \text{COERCE}$$

$$\frac{\Gamma \Vdash \tau \sim \sigma \rightsquigarrow \gamma \quad \mathtt{ftv}\,\tau \subseteq \mathtt{ftv}\,\Gamma \quad \mathtt{ftv}\,\sigma \subseteq \mathtt{ftv}\,\Gamma}{\Gamma \vdash \tau \sim \sigma : \tau \sim \sigma \rightsquigarrow [\![\tau \sim \sigma]\!]\ \gamma}\ \text{EQ} \qquad \frac{\Gamma \vdash f : \sigma \to \tau \rightsquigarrow \hat{f} \quad \Gamma \vdash e : \sigma \rightsquigarrow \hat{e}}{\Gamma \vdash f\ e : \tau \rightsquigarrow \hat{f}\ \hat{e}}\ \text{APP.EXPR}$$

$$\frac{\Gamma \vdash f : \sigma \to \tau \rightsquigarrow \hat{f} \quad \mathtt{ftv}\,\sigma \subseteq \mathtt{ftv}\,\Gamma}{\Gamma \vdash f\ \sigma : \tau \rightsquigarrow \hat{f}\ [\![\sigma]\!]}\ \text{APP.TY} \qquad \frac{\Gamma, x :: \sigma \vdash e : \tau \rightsquigarrow \hat{e}}{\Gamma \vdash \lambda(x :: \sigma).e : \sigma \to \tau \rightsquigarrow \lambda(x :: [\![\sigma]\!]).\hat{e}}\ \text{LAM.EXPR}$$

$$\frac{\Gamma, \alpha \vdash e : \tau \rightsquigarrow \hat{e} \quad \alpha \notin \mathtt{ftv}\,\Gamma}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau \rightsquigarrow \Lambda\alpha.\hat{e}}\ \text{LAM.TY}$$

$$\frac{\Gamma, x :: \rho \sim \sigma \vdash e : \tau \rightsquigarrow \hat{e} \qquad x, y \notin \Gamma}{\Gamma \vdash \lambda(\rho \sim \sigma).e : (\rho \sim \sigma) \to \tau \rightsquigarrow \lambda y.\mathbf{case}\ y\ \mathbf{of}[\![\rho \sim \sigma]\!]\ (x : [\![\rho]\!] \sim [\![\sigma]\!]) \to \hat{e}}\ \text{LAM.EQ}$$

Fig. 7 System $F_{\sim}$-expr translation rules

5.4 Translation of proofs

Figure 8 shows how to produce coercion terms from the equality proof. We omitted the rules for universal quantification: these are analogous. The rules REFL, SYM, TRANS, and ASSUM have a direct mapping to a coercion-term.

Each individual rule is a solution to a small puzzle. For a proof of $\tau \sim \sigma$, we combine coercions γ until they have exactly the same type according to coercion type rules in Figure 6. For example, we can verify that rule SUB.ARR constructs a coercion with the right type as follows:

$$\begin{array}{ll}
\gamma_1 & :: \tau_3 \sim (\tau_4 \to \tau_5) \\
\gamma_2 & :: \tau_6 \sim (\tau_7 \to \tau_8) \\
\gamma_3 & :: \tau_3 \sim \tau_6 \\
\mathtt{sym}\,\gamma_1 \circ \gamma_3 \circ \gamma_2 & :: (\tau_4 \to \tau_5) \sim (\tau_7 \to \tau_8) \\
\mathtt{right}\,(\mathtt{sym}\,\gamma_1 \circ \gamma_3 \circ \gamma_2) & :: \tau_4 \sim \tau_7 \\
\mathtt{right}\,(\mathtt{right}\,(\mathtt{sym}\,\gamma_1 \circ \gamma_3 \circ \gamma_2)) & :: \tau_5 \sim \tau_8
\end{array}$$

$$\boxed{\Gamma \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma}$$

$$\Gamma \Vdash \tau \sim \tau \rightsquigarrow [\![\tau]\!] \ \text{REFL} \qquad \frac{\Gamma \Vdash \tau^r \sim \tau^l \rightsquigarrow \gamma}{\Gamma \Vdash \tau^l \sim \tau^r \rightsquigarrow \mathtt{sym}\,\gamma} \ \text{SYM} \qquad \frac{\Gamma \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma_1 \quad \Gamma \Vdash \tau_2 \sim \tau_3 \rightsquigarrow \gamma_2}{\Gamma \Vdash \tau_1 \sim \tau_3 \rightsquigarrow \gamma_2 \circ \gamma_1} \ \text{TRANS}$$

$$\frac{x : \tau_1 \sim \tau_2 \in \Gamma}{\Gamma \Vdash \tau_1 \sim \tau \rightsquigarrow x} \ \text{ASSUM} \qquad \frac{\Gamma \Vdash \sigma \sim \tau \rightsquigarrow \gamma_1 \quad \Gamma \Vdash \rho \sim \varrho \rightsquigarrow \gamma_2}{\Gamma \Vdash \tau \to \rho \sim \sigma \to \varrho \rightsquigarrow \gamma_1 \to \gamma_2} \ \text{CON.ARR}$$

$$\frac{\begin{array}{c}\gamma_1 :: (\tau_3 \sim (\tau_4 \to \tau_5)) \in \Gamma \qquad \gamma_2 :: (\tau_6 \sim (\tau_7 \to \tau_8)) \in \Gamma \\ \Gamma \Vdash \tau_3 \sim (\tau_6 \rightsquigarrow \gamma_3) \\ \gamma_5 = \mathtt{right}\,(\mathtt{sym}\,\gamma_1 \circ \gamma_3 \circ \gamma_2) \\ \gamma_6 = \mathtt{right}\,(\mathtt{right}\,(\mathtt{sym}\,\gamma_1 \circ \gamma_3 \circ \gamma_2)) \\ \gamma_5 :: \tau_4 \sim \tau_7, \gamma_6 :: \tau_5 \sim \tau_8 \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma_4 \\ \gamma_1, \gamma_2 \notin \Gamma\end{array}}{\Gamma \Vdash \tau_1 \sim \tau_2 \rightsquigarrow \gamma_4} \ \text{SUB.ARR}$$

Fig. 8 Equality proof translation rules

5.5 Correctness

It is easy to show that a well-typed equality-proof is translated to a well-typed coercion term (see above). It is also easy to show that a System $F_\sim$ expression is translated to a well-typed System F_C expression. Because System F_C has type soundness, we thus obtain that System $F_\sim$ also has type soundness.

6 Related work

Several approaches propose limitations to make inference for GADTs tractable. Many specifications and algorithms for GADTs consist 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.

6.1 Restricting expressiveness for tractable inference

Sulzmann et al. [16] show that inference for GADTs is undecidable without a helping hand from the programmer in the form of type annotations, and show that principal typing is lost. They present an inference algorithm that generates implication constraints, and apply a technique called Herbrand abduction [8] to solve these constraints. A solution to these constraints is a normalized constraint that implies the generated constraints. The authors define a notion of a sensible solution; under these conditions the resulting type is principal. These conditions are specified in terms of the algorithm.

We are reluctant to adopt such an approach, because it requires programmers to think in terms of an algorithm. To discover why their program is (not) accepted by the type inferencer, the steps of the algorithm have to be performed by hand.

Schrijvers et al. [14] recently presented two type systems for GADTs, and the inference algorithm OutsideIn. The first type system is relatively simple, but inference for it is undecidable. The latter is more restricted, but has OutsideIn as a sound and complete inference algorithm. Both specification and inference algorithm are given in terms of implication constraints.

When type inference is combined with coercion inference, additionally a coercion $\tau_1 \sim \tau_2$ may be constructed when τ_1 unifies with τ_2 (the result then corresponds to the REFL rule). The main idea of OutsideIn is that binding of variables during such a unification is not allowed on variables that are free in the environment (of the enclosing let-expression or case-branch). These are called the "untouchable" unification variables. Type checking proceeds as normal, except that a unification of a Skolem constant or with an untouchable requires a coercion to be constructed. The actual construction of these coercions is done after type checking is complete.

The approach is presented as a general solution to inference for GADTs. Although this approach is sound and complete with respect to their specification, their specification still requires type annotations for most functions with GADT patterns. For example, with OutsideIn, a type signature is needed for the following code that uses GADTs in the typical way:

$$
\begin{array}{l}
\textbf{data}\ D\ a\ \textbf{where} \\
\quad C1 :: Int \to D\ Int \\
\quad C2 :: Bool \to D\ Bool \\
\quad \text{-- needs: } f :: D\ \alpha \to \alpha \\
f\ x = \textbf{case}\ x\ \textbf{of} \\
\quad C1\ y \to y \\
\quad C2\ z \to z
\end{array}
$$

The reason is that α is free in the environment, and thus not unified.

This approach still lacks a good specification: it introduces "suspended judgments" to encode a two-phase typing of an expression, which is actually harder to comprehend than the algorithm itself. A problem here is that the untouchable variables do not appear explicitly in conventional type system specifications. They might be eliminated by unification during conventional type inference, and are notationally equal to the type the variable is unified with. Thus, such a specification requires a programmer to know how the algorithm is distributing its type variables over the program.

Lin and Sheard [7] propose the Pointwise GADT type system. The notion pointwise comes from the correspondence between the range-type of a constructor and the type of the scrutinee in a GADT case expression. During inference in case branches, type information flows from and to such types. The authors identify two typical usage-patterns for GADTs, and make a case to restrict the use of GADTs to these patterns. *Parametric instantiation* entails that each case branch assumes the same specific instance of a polymorphic data type. *Type indexing* entails that each case branch assumes a different instance of a polymorphic data type. The authors propose a unification technique to restrict this bidirectional information flow. From an implementation point-of-view, this approach is promising: the authors performed several case studies of typical uses of GADTs and presented a mechanism that types many typical usages of GADT programs without explicit type annotations. However,

it is not clear how this approach relates to the challenges we pointed out in Sect. 3.2. Furthermore, the approach lacks a specification to serve as an explanation to the programmer.

6.2 Type annotation propagation strategies

Pottier and Régis-Gianas [13] use shape inference. 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, iteratively. More iterations give rise to a better spread of annotations, at a cost of performance. 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. [11, 12] define a notion of wobbly types to combine type checking and type inference, which is based on earlier work on boxy types [20]. 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 a 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 piggybacked 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].

6.3 Type conversion strategies

Peyton Jones et al. [11, 12] 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.

Wazny [21] uses a constraint-based strategy. The GADT aspect of this strategy is covered separately by Sulzmann et al. [17], Stuckey and Sulzmann [15]. 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. 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 often use them. In contrast to Peyton Jones et al. [12], restrictions on type conversions are not mentioned.

7 Conclusion

We presented a specification for GADT inference in terms of the language System $F_{\sim}$. This language deviates from System F in three ways: it has syntax to request an equality proof, a lambda to take an equality proof as argument and bring it in scope, and automatic coercions based on equality proofs in scope. In this language, we can express GADTs using the folklore Church encoding for data types.

Compared to other approaches, our specification is a small extension of a bare System F, which allows us to treat data types and case expressions as syntactic sugar. Furthermore, our specification describes what to do with other forms of pattern matching and binding, exploiting encodings as conventional System F terms.

As future work, it may be worthwhile to investigate if algorithms such as OutsideIn [14] can be specified in a simpler way by taking an implicitly typed variation on System $F_{\sim}$ as a basis.

Acknowledgements This work was supported by Microsoft Research through its European PhD Scholarship Programme. We thank the anonymous reviewers of TFP 08 and TFP-HOSC for extensive comments on earlier versions of this paper.

References

1. Baars, A.I., Swierstra, S.D.: Typing dynamic typing. In: ICFP, pp. 157–166 (2002)
2. Baars, A.I., Swierstra, S.D., Viera, M.: Typed transformations of typed abstract syntax. In: Kennedy, A., Ahmed, A. (eds.) TLDI, pp. 15–26. ACM, New York (2009)
3. Cheney, J., Hinze, R.: First-class phantom types. Tech. rep., Cornell University (2003)
4. Dijkstra, A.: Stepping through Haskell. PhD thesis, Utrecht University, Department of Information and Computing Sciences (2005). http://www.cs.uu.nl/wiki/Ehc/WebHome, papers/dijkstra05phd.pdf
5. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Weirich, S. (ed.) Haskell, pp. 93–104. ACM, New York (2009)
6. Jeuring, J., Leather, S., Magalhães, J.P., Yakushev, AR: Libraries for generic programming in Haskell. In: AFP, pp. 165–229 (2008)
7. Lin, C., Sheard, T.: Pointwise generalized algebraic data types. In: Kennedy, A., Benton, N. (eds.) TLDI, pp. 51–62. ACM, New York (2010)
8. Maher, M.J.: Herbrand constraint abduction. In: LICS, pp. 397–406. IEEE Computer Society, Washington (2005)
9. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: A leaner specification for generalized algebraic data types. In: TFP, vol. 9, pp. 65–80 (2008)
10. Mogensen, T.Æ.: Efficient self-interpretations in lambda calculus. J. Funct. Program. **2**(3), 345–363 (1992)
11. Peyton Jones, S.L., Washburn, G., Weirich, S.: Wobbly types: type inference for generalised algebraic data types. Tech. Rep. MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389 (2004)
12. Peyton Jones, S.L., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP, pp. 50–61 (2006)
13. Pottier, F., Régis-Gianas, Y.: Stratified type inference for generalized algebraic data types. In: POPL, pp. 232–244 (2006)
14. Schrijvers, T., Jones, S.L.P., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: ICFP, pp. 341–352 (2009)
15. Stuckey, P.J., Sulzmann, M.: Type inference for guarded recursive data types. CoRR abs/cs/0507037 (2005)
16. Sulzmann, M., Schrijvers, T., Stuckey, P.J.: Type inference for GADTs via Herbrand constraint abduction. Manuscript (2006)
17. Sulzmann, M., Wazny, J., Stuckey, P.J.: A framework for extended algebraic data types. In: FLOPS, pp. 47–64 (2006)

18. Sulzmann, M., Chakravarty, M.M.T., Peyton Jones, S.L., Donnelly, K.: System F with Type Equality Coercions. In: TLDI, pp. 53–66 (2007)
19. Urzyczyn, P.: Positive recursive type assignment. Fundam. Inform. **28**(1–2), 197–209 (1996)
20. Vytiniotis, D., Weirich, S., Peyton Jones, S.L.: Boxy types: inference for higher-rank types and impredicativity. In: Reppy, J.H., Lawall, J.L. (eds.) ICFP, pp. 251–262. ACM, New York (2006)
21. Wazny, J.R.: Type inference and type error diagnosis for Hindley/Milner with extensions. PhD thesis, The university of Melbourne (2006)