Translating Generalized Algebraic Data Types to System F

Martin Sulzmann and Meng Wang
{sulzmann,wangmeng}@comp.nus.edu.sg

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543

Abstract. Generalized algebraic data types (GADTs) extend ordinary algebraic data types by refining the types of constructors with syntactic equality constraints. This is highly useful and allows for novel applications such as strongly-typed evaluators, typed LR parsing etc. To translate GADTs we need to enrich the System F style typed intermediate languages of modern language implementations to capture these equality constraints. We show that GADTs can be translated to a minor extension of System F where type equality proofs are compiled into System F typable proof terms. At run-time proof terms evaluate to the identity. Hence, they can be safely erased before execution of the program. We provide evidence that our approach scales to deal with extensions where equality is not anymore syntactic. The benefit of our method is that type checking of target programs remains as simple as type checking in System F. Thus, we can offer a light-weight approach to integrate GADTs and extensions of it into existing implementations.

1 Introduction

Generalized algebraic data types (GADTs) are an extension of (boxed) existential types [17]. The novelty of GADTs is that we may include syntactic type equality constraints, i.e. equality among Herbrand terms [16], to refine the types of constructors. Thus, we can type more programs. The following is a classic example and defines a strongly-typed evaluator for a simple language. Note that we make use of Haskell style syntax [11] in examples.

Example 1. We first introduce a GADT to ensure that well-formed expressions are well-typed.

```
data Exp  a = (a=Int) => Zero  | (a=Int) => Succ (Exp Int)
            | forall b c. (a=(b,c)) => Pair (Exp b) (Exp c)
```

In contrast to algebraic data types we may refine the type of a GADT depending on the particular constructor. Note that constructor `Pair` has type $\forall a, b, c.(a = (b, c)) \Rightarrow Exp\ b \rightarrow Exp\ c \rightarrow Exp\ a$. Type variables b and c appear not in the resulting type, hence, we consider these variables as "existentially" quantified. In the source syntax these existentially quantified variables are introduced by the `forall` keyword. For example, we find that `Zero 0` has type $Exp\ Int$ and `Pair (Succ (Zero 0) (Zero 0)` has type $Exp\ (Int, Int)$.

The real advantage of GADTs is that we may make use of type equality constraints when pattern matching over a GADT. Here is a evaluator for our expression language.

```
eval :: Exp a -> a
eval Zero = 0
eval (Succ e) = (eval e) + 1
eval (Pair x y) = (eval x, eval y)
```

At first look it may be surprising that `eval` has type $\forall a.Exp\ a \rightarrow a$. Consider the first clause. We match `Zero` against type $Exp\ a$ which gives rise to constraint $a = Int$. We type the right-hand side of the function definition under the constraint $a = Int$. Hence, we can convert `0`'s type which is Int to the type a. Note that the constraint $a = Int$ does not affect other parts of the program. Similar reasoning applies to the other clauses. Hence, `eval`'s annotation is correct.

The above is one of the many examples [20, 30, 24, 25] that show how to write more expressive programs and transformations with GADTs. Hence, it is desirable that GADTs become a standard feature supported by todays modern programming languages such as ML [19] and Haskell [11]. An important question is what impact GADTs have on existing language implementations such as GHC [7] and FLINT/ML [27].

The widely adopted translation scheme for typed *source* languages is to maintain types by means of a sufficiently rich typed *target* language. It is well-known how to translate Hindley/Milner to System F [10]. The advantage of System F [8, 26] is that type abstraction and application can be made explicit. Thus, the more "complicated" Hindley/Milner type inference process can be turned into some "simple" System F type checking. In case of GADTs we clearly need a richer variant of System F. There is a huge variety of System F variants we can choose from, e.g. consider [28] and the references therein. Previous work [14] suggests to extend System F with GADTs itself. We believe that a simpler extension is sufficient.

GADT type checking involves verifying statements such as $C \supset t_1 = t_2$ where $\supset$ refers to Boolean implication, C contains a set of type constraints and t_1 and t_2 are types. Our idea is to apply the "proofs are programs" principle (a.k.a. Curry-Howard isomorphism) and turn the proof for $C \supset t_1 = t_2$ into a System F typable proof term (f, g) such that $C \supset t_1 = t_2$ iff $\Gamma \vdash (f, g) : (t_1 \rightarrow t_2, t_2 \rightarrow t_1)$ where Γ is a type environment representing C. Thus, we can translate GADTs to System F_s, a simple variant of System F with (boxed) existential types [18] where we insert proof terms to represent type conversions.

Here is the translation of Example 1 to System F_s.

```
data Exp a = Zero <(a->Int,Int->a)> | Succ <(a->Int,Int->a)> (Exp Int)
           | Pair b c <(a->(b,c),(b,c)->a)> (Exp b) (Exp c)

eval = Λ a. λ x:Exp a. case x of
        Zero <(f,g)>      -> <g> 0
        Succ <(f,g)> e    -> <g> ((eval [Int] e) + 1)
        Pair_bc <(f,g)> x y -> <g> (eval [b] x, eval [c] y)
```

Note that proof terms are marked via <>. Our choice of notation is deliberate and resembles the staging operator found in the area of multi-stage programming [35]. "Staged" proof terms do not interact with the "real" program text. This is enforced by the System F_s typing rules and the only difference compared to System F. Proof terms are only used for typing purposes. E.g., consider the

first clause where proof term `g`, representing the GADT type conversion from Int to a, is applied to `0`. This guarantees that the resulting System F_s program is well-typed. At run-time proof terms turn out to be equivalent to the identity function. Hence, we can safely erase them before execution of the program (similar to the way we erase types). Here is the above program after erasing types and proof terms.

```
eval = λ x. case x of
         Zero      -> 0
         Succ e    -> (eval e) + 1
         Pair x y  -> (eval x, eval y)
```

Our translation scheme extends to systems with user-specifiable type constraints [29, 2]. Hence, we can identify a minimal extension of System F which is sufficiently rich to host a variety of GADT style source languages.

In summary, our contributions are:

- We define an extension of System F with staged typing to which we refer to as System F_s (Section 4.1).
- We introduce a proof system to represent type equations and their proofs. We show that all derivable proof terms are well-typed in System F and their evaluation cannot go wrong (Section 4.2).
- We give a type-directed translation from GADTs to System F_s where GADT type equation proofs are represented by staged System F expressions. We show that the resulting System F_s program preserves types and the semantic meaning of the original GADT program. We also guarantee that the System F_s program cannot go wrong. Thus, we obtain an alternative type soundness result for GADTs (Section 4.3).
- We establish sufficient conditions under which GADT programs can be directly translated to (boxed) existential types (Section 5).
- We explore how to extend our translation method to variants of GADTS where type equality is user-specifiable, i.e. not anymore syntactic (Section 6).

Section 2 spells out some basic assumptions and notations used throughout the paper. Section 3 provides some background information on GADTs. We conclude in Section 7. We discuss related work where appropriate.

Further details such as proofs will appear in a forthcoming technical report. For the time being, we refer the interested reader to a preliminary technical report [34].

2 Preliminaries

For the purpose of this paper, we neglect the issue of type inference for GADTs which is known to be a hard problem [31, 32]. Hence, we assume that programs are sufficiently type annotated.

We make use of System F extended with Odersky and Läufer style "boxed existential types" [17, 18] (commonly referred to as "existential types" for short). We assume familiarity with the standard methods and techniques to establish syntactic type soundness [23, 38] which implies that "well-typed programs cannot go wrong".

$$\text{(Var)}\ \frac{(x : \forall \bar{a}.C' \Rightarrow t') \in \Gamma \quad C \supset [\overline{t/a}]C'}{C, \Gamma \vdash x : [\overline{t/a}]t'} \qquad \text{(App)}\ \frac{C, \Gamma \vdash e_2 : t_2 \quad C, \Gamma \vdash e_1 : t_2 \to t}{C, \Gamma \vdash e_1\ e_2 : t} \qquad \text{(Abs)}\ \frac{C, \Gamma.x : t_1 \vdash e : t_2}{C, \Gamma \vdash \lambda x.e : t_1 \to t_2}$$

$$\text{(}\forall\text{Intro)}\ \frac{C_1 \wedge C_2, \Gamma \vdash e : t \quad \bar{a} \cap fv(C_1, \Gamma) = \emptyset}{C_1 \wedge \exists \bar{a}.C_2, \Gamma \vdash e : \forall \bar{a}.C_2 \Rightarrow t} \qquad \text{(Let)}\ \frac{C, \Gamma \vdash e_1 : \sigma \quad C, \Gamma \cup \{g : \sigma\} \vdash e_2 : t_2}{C, \Gamma \vdash \mathsf{let}\ g = e_1\ \mathsf{in}\ e_2 : t_2}$$

$$\text{(Eq)}\ \frac{C, \Gamma \vdash e : t \quad C \supset t = t'}{C, \Gamma \vdash e : t'} \qquad \text{(Case)}\ \frac{C, \Gamma \vdash e : t_1 \quad C, \Gamma \vdash p_i \to e_i : t_1 \to t_2 \quad \text{for } i \in I}{C, \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I} : t_2}$$

$$\text{(Pat)}\ \frac{p : t_1 \vdash \forall \bar{b}.(C_p \blacktriangleright \Gamma_p) \quad \bar{b} \cap fv(C, \Gamma, t_2) = \emptyset \quad C \wedge C_p, \Gamma \cup \Gamma_p \vdash e : t_2}{C, \Gamma \vdash p \to e : t_1 \to t_2} \qquad \text{(P-Var)}\ x : t \vdash (True \blacktriangleright \{x : t\})$$

$$\text{(P-K)}\ \frac{K : \forall \bar{a}, \bar{b}.C \Rightarrow t_1 \to ... \to t_n \to F\ \bar{a} \quad \bar{b} \cap \bar{a} = \emptyset \quad p_i : [\overline{t'/a}]t_i \vdash \forall \overline{b'_i}.(C'_i \blacktriangleright \Gamma_{p_i}) \quad \text{for } i = 1, ..., n}{K\ p_1 ... p_n : F\ \bar{t} \vdash \forall \overline{b'_1, ..., b'_n}, \bar{b}.(C'_1 \wedge ... \wedge C'_n \wedge [\overline{t'/a}]C \blacktriangleright \Gamma_{p_1} \cup ... \cup \Gamma_{p_n})}$$

Fig. 1. GADT Typing Rules

We assume familiarity with the concepts of substitutions, unifiers etc [16]. We write $\bar{o}$ as a short-hand for a sequence of objects $o_1, ..., o_n$ (e.g. expressions, types etc). We write $[\overline{o/a}]o'$ to denote applying substitution $[\overline{o/a}]$ on o', i.e. simultanously replacing all occurrences of variables a_i by object o_i in o' for $i = 1, ..., n$. We write $o_1 = o_2$ to specify equality among object o_1 and o_2 (a.k.a. unification problem). Sometimes, we write $o_1 \equiv o_2$ to denote syntactic equivalence between two objects o_1 and o_2 in order to avoid confusion with $=$. We write $fv(o)$ to denote the free variables in some object o.

3 Background: Generalized Algebraic Data Types

GADTs first appeared under the name *guarded recursive data types* [39] and *first-class phantom types* [6]. Although, the idea of GADTs appears in some even earlier work [40]. Here, we use the more popular becoming name generalized algebraic data types [14]. Our formulation of GADTs as an extension of Hindley/Milner is closest to [31]. Note that all existing variations of GADTs are largely equivalent.

The language of expressions, types and constraints is as follows.

Expressions	e	$::= K \mid x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ g = e\ \mathsf{in}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I}$
Patterns	p	$::= x \mid K\ \bar{p}$
Types	t	$::= a \mid t \to t \mid F\ \bar{t}$
Constraints	C	$::= t = t \mid C \wedge C$
Type Schemes	σ	$::= t \mid \forall \bar{\alpha}.C \Rightarrow t$

For simplicity, we leave out type annotations and recursive function definitions but may make use of them in examples. Pattern matching syntax used in examples can be straightforwardly expressed in terms of case expressions. We assume that K refers to constructors of user-defined data types $F\ \bar{a}$. Basic types such as booleans, integers, tuples and lists are predefined and their constructors are recorded in some initial environment Γ_{init}. As usual patterns are assumed to be linear, i.e., each variable occurs at most once. We use $True$ as a short-hand for some always satisfiable constraint, e.g. $Int = Int$.

We assume that GADT definitions are pre-processed and the types of their constructors recorded in some initial environment Γ_{init}. E.g., the GADT from Example 1 implies constructors $Zero : \forall a.(a = Int) \Rightarrow Exp\ a$, $Succ : \forall a.(a = Int) \Rightarrow Exp\ Int \rightarrow Exp\ a$ and $Pair : \forall a, b, c.(a = (b, c)) \Rightarrow Exp\ b \rightarrow Exp\ c \rightarrow Exp\ a$. This explains the need for "constrained" type schemes of the form $\forall \bar{a}.C \Rightarrow t$.

The typing rules describing well-typing of expressions are in Figure 1. We introduce judgments $C, \Gamma \vdash e : t$ to denote that expression e has type t under constraint C and environment Γ which holds a set of type bindings of the form $x : \sigma$. A judgment is valid if we find a derivation w.r.t. the typing rules.

In rule (Var), we build a type instance of a type scheme by demanding that the instantiated constraint is entailed by the given constraint. This is formally expressed by $C \supset \overline{[t/a]}C'$. Note that $C \supset t_1 = t_2$ holds iff (1) C does not have a unifier, or (2) for any unifier ϕ of C we have that $\phi(t_1) = \phi(t_2)$ holds.

Rule (∀Intro) is the familiar HM(X) [21, 33] quantifier introduction rule. Note that our constraint language does not support existential quantification explicitly. However, w.l.o.g. the constraint $\exists \bar{a}.C_2$ is equivalent to some constraint C_3 consisting of type equations only.

Rules (Abs), (App), (Case) and (Let) are standard. The only worth mentioning point is that we consider $p \rightarrow e$ as a (special purpose) expression only appearing in intermediate steps.

Next, we consider the GADT specific rules. In rule (Eq) we are able to change the type of an expression. [1] In rule (Pat) we make use of an auxiliary judgment $p : t \vdash \forall \bar{b}.(C_p \mid \Gamma_p)$ which establishes a relation among pattern p of type t, the constraint C_p arising out of p and the binding Γ_p of variables in p. Variables $\bar{b}$ refer to all "existential" variables. Logically, these variables must be considered as universally quantified. Hence, we write $\forall \bar{b}$. The side condition $\bar{b} \cap fv(C, \Gamma, t_2) = \emptyset$ prevents existential variables from escaping.

A common property for typing derivations involving constraints is that if the constraint in the final judgment is satisfiable all constraints arising in intermediate judgments are satisfiable too. This property is lost here as the following variation of Example 1 shows.

```
eval :: Exp Bool -> (Int->Int)
eval Zero = 0
```

Matching the pattern `Zero` against the type $Exp\ Bool$ leads to the temporary type constraint $Bool = Int$. This constraint is equivalent to $False$. Thus, we can

[1] Some formulations allow to change the type of (sub)patterns [31]. This may matter if patterns are nested. For brevity, we neglect such an extension. Note that in case patterns are evaluated in a certain order, say from left-to-right, we can simply translate a nested pattern into a sequence of shallow patterns. This is done in GHC.

give any type to `0` and therefore `eval`'s rather strange looking type is correct. Note that $False$ does not appear in the final judgment.

Cheney and Hinze [6] observed that such programs are "meaningless" because there is no value of type $Exp\ Bool$. Hence, we can never apply `eval` (under its new type) to a concrete value. We can rule out such meaningless programs by demanding that all constraints in a typing derivation must be satisfiable, i.e. have a unifier.

Assumption 1 *W.l.o.g. we assume that for any GADT typing derivation we have that all constraints in typing judgments have a unifier.*

The important consequence is that the entailment test between constraints is now constructive. We exploit this fact in the following section where we represent type equation proofs via well-typed programs.

4 Translating Generalized Algebraic Data Types

We proceed in three steps. First, we introduce the syntax and static semantics of System F_s. Typing in System F_s is staged whereas the dynamic semantics remains essentially unchanged compared to System F. Then, we define a proof system to represent GADT type equations by proof terms typable in System F. Finally, we give a type-directed translation scheme from GADTs to System F_s where we insert proof terms to mimic GADT type conversions. Proof terms are staged and evaluate to the identity at run-time. Hence, we can safely remove them without affecting the semantic meaning of the original GADT program. Note that we could achieve our technical results using System F directly. However, we feel that System F_s helps to understand the separation between proof terms representing type conversions and the real program text.

4.1 An Extension of System F with Staged Typing

System F_s inherits the syntax of expressions and types from System F.

$$\begin{array}{lll}
\text{Target} & E ::= x \mid E\ E \mid \lambda x : T.E \mid E\ [\overline{T}] \mid \Lambda \bar{a}.E \mid < E > \\
& \quad \mathsf{let}\ g : T = E\ \mathsf{in}\ E \mid \mathsf{case}\ (T)\ E\ \mathsf{of}\ [P_i \to E_i]_{i \in I} \\
\text{Patterns} & P ::= (x : T) \mid K_{\bar{b}}\ \bar{P} \mid < P > \\
\text{Types} & T ::= a \mid T \to T \mid < T > \mid \forall \bar{a}.T
\end{array}$$

The only extension is the staging operator $<>$ which allows us to form staged expressions, patterns and types.

We assume that type bindings $(x :^l T)$ and judgments $\Gamma \vdash^l_{F_s} E : T$ carry some stage information; a natural number l. The typing rules in Figure 2 guarantee that lower staged expressions/patterns cannot affect the type of higher staged expressions/patterns and vice versa. See rules (Var), ($<>\uparrow$) and (P-$<>$). The only exception is rule ($<>\downarrow$) where a staged expression has a unstaged type. As we will see, via this rule we will mimic the GADT (Eq) rule. Note that constructors are explicitly annotated with the set of existential variables. See rule (P-K). The remaining rules contain no surprises as they leave the stage l unchanged.

Commonly, we write $\{x_1 : T_1, ..., x_n : T_n\} \vdash_{F_s} E : T$ as a short-hand for $\{x_1 :^1 T_1, ..., x_n :^1 T_n\} \vdash^1_{F_s} E : T$. We write $\Gamma \vdash_F E : T$ to denote a System F

$$\text{(Abs)}\ \frac{\Gamma \cup \{x :^l T_1\} \vdash^l_{F_s} E : T_2}{\Gamma \vdash^l_{F_s} \lambda x : T_1.E : T_1 \to T_2} \quad \text{(App)}\ \frac{\Gamma \vdash^l_{F_s} E_1 : T_1 \to T_2 \quad \Gamma \vdash^l_{F_s} E_2 : T_1}{\Gamma \vdash^l_{F_s} E_1\ E_2 : T_2}$$

$$\text{(T-Abs)}\ \frac{\Gamma \vdash^l_{F_s} E : T \quad \bar{a} \cap fv(\Gamma) = \emptyset}{\Gamma \vdash^l_{F_s} \Lambda \bar{a}.E : \forall \bar{a}.T} \quad \text{(T-App)}\ \frac{\Gamma \vdash^l_{F_s} E : \forall \bar{a}.T'}{\Gamma \vdash^l_{F_s} E\ [\overline{T}] : [\overline{T/a}]T'}$$

$$\text{(Var)}\ \frac{(x :^l \forall \bar{a}.T) \in \Gamma}{\Gamma \vdash^l_{F_s} x : \forall \bar{a}.T} \quad \text{(Let)}\ \frac{\Gamma \vdash^l_{F_s} E : \forall \bar{a}.T \quad \Gamma \cup \{g :^l \forall \bar{a}.T\} \vdash^l_{F_s} E' : T'}{\Gamma \vdash^l_{F_s} \mathsf{let}\ g : \forall \bar{a}.T = E\ \mathsf{in}\ E' : T'}$$

$$\text{(Case)}\ \frac{\Gamma \vdash^l_{F_s} E : T' \quad \Gamma \vdash^l_{F_s} P_i \to E_i : T' \to T \quad \text{for } i \in I}{\Gamma \vdash^l_{F_s} \mathsf{case}\ (T')\ E\ \mathsf{of}\ [P_i \to E_i]_{i \in I} : T} \quad \text{(Pat)}\ \frac{P : T_1 \vdash^l_{F_s} \forall \bar{b}.\Gamma_P \quad \bar{b} \cap fv(\Gamma, T_2) = \emptyset \quad \Gamma_P \cup \Gamma \vdash^l_{F_s} E : T_2}{\Gamma \vdash^l_{F_s} P \to E : T_1 \to T_2}$$

$$(<>\uparrow)\ \frac{\Gamma \vdash^{l+1}_{F_s} E : T}{\Gamma \vdash^l_{F_s} < E >:< T >} \quad (<>\downarrow)\ \frac{\Gamma \vdash^{l+1}_{F_s} E : T_1 \to T_2}{\Gamma \vdash^l_{F_s} < E >: T_1 \to T_2}$$

$$\text{(P-Var)}\ (x : T) : T \vdash^l_{F_s} \{x :^l T\} \qquad \text{(P-K)}\ \frac{K_{\bar{b}} : \forall \bar{a}, \bar{b}.T_1 \to ... \to T_n \to F\ \bar{a} \quad \bar{b} \cap \bar{a} = \emptyset \quad P_i : [\overline{T'/a}]T_i \vdash^l_{F_s} \forall \overline{b'_i}.\Gamma_{P_i} \quad \text{for } i = 1, ..., n \quad \bar{c} = \overline{b'_1, ..., b'_n}, \bar{b} \quad \Gamma = \Gamma_{P_1} \cup ... \cup \Gamma_{P_n}}{K_{\bar{b}}\ \bar{b}\ P_1 ... P_n : F\ [\overline{T'/a}] \vdash^l_{F_s} \forall \bar{c}.\Gamma}$$

$$\text{(P-<>)}\ \frac{P : T \vdash^{l+1}_{F_s} \forall \bar{b}.\Gamma}{< P >:< T > \vdash^l_{F_s} \forall \bar{b}.\Gamma}$$

Fig. 2. System F_s Typing Rules

derivation where we do not make use of the "staged" rules (<>↑), (<>↓) and (P-<>).

We explain the dynamic semantics of System F_s in terms of a standard single-step rewriting semantics [38] written $E \rightarrowtail E'$. This gives rise to a system of *reductions* where each expression E either *reduces* (or *evaluates*) to a value v, written $E \rightarrowtail^* v$, or we encounter failure (this includes the case that the reduction gets stuck). We assume that staged expressions and patterns follow the same evaluation rules. That is, we simply replace $< E >$ by E and $< P >$ by P before evaluation. The entire development is standard and can be found elsewhere [38, 23]. Hence, we omit the details.

4.2 Proof Terms for Type Equations

We define a proof system for representing (syntactic) equality among types where type equations carry proof terms. We write $(f : g) : t_1 = t_2$ to denote that (f, g) is the proof for $t_1 = t_2$. Depending on the context, we may silently drop proof terms if they do not matter.

The proof rules are formulated in terms of judgments $C \vdash_= (f, g) : t_1 = t_2$. They are directly derived from a constructive formulation of type equality. Details are in Figure 3.

Rule (Var) looks up an equality constraint. For convenience, we interpret a constraint as a set of type equations. Rules (Ref) and (Sym) describe reflexivity and symmetry of equality. In rule (Trans), we write $f \cdot g$ as a short-hand for

$$\text{(Var)}\ \frac{(f,g) : t_1 = t_2 \in C}{C \vdash_= (f,g) : t_1 = t_2}\quad \text{(Ref)}\ C \vdash_= (\lambda x.x, \lambda x.x) : t = t\quad \text{(Sym)}\ \frac{C \vdash_= (f,g) : t_1 = t_2}{C \vdash_= (g,f) : t_2 = t_1}$$

$$\text{(Trans)}\ \frac{C \vdash_= (f_1,g_1) : t_1 = t_2 \quad C \vdash_= (f_2,g_2) : t_2 = t_3}{C \vdash_= (f_2 \cdot f_1, g_1 \cdot g_2) : t_1 = t_3}$$

$$(F \downarrow_i)\ \frac{C \vdash_= (f,g) : F\ t_1...t_n = F\ t'_1...t'_n}{C \vdash_= (\mathit{decompFf}_i\ [\bar{t}\ \overline{t'}]\ (f,g), \mathit{decompFg}_i\ [\bar{t}\ \overline{t'}]\ (f,g)) : t_i = t'_i}$$

$$(F \uparrow)\ \frac{C \vdash_= (f_i,g_i) : t_i = t'_i \quad \text{for } i = 1,...,n}{C \vdash_= (\mathit{compFf}\ [\bar{t}\ \overline{t'}]\ (f_1,g_1)...(f_n,g_n), \mathit{compFg}\ [\bar{t}\ \overline{t'}]\ (f_1,g_1)...(f_n,g_n)) : F\ \bar{t} = F\ \overline{t'}}$$

Fig. 3. Type Equation Proof System

$\lambda x.f\ (g\ x)$ to define a proof term representing transitivity. Rules $(F \uparrow)$ and $(F \downarrow_i)$ deal with user-defined types $F\ \bar{a}$ (including the function type). Recall that $\bar{t}$ and $\overline{t'}$ are short-hands for $t_1, ..., t_n$ and $t'_1, ..., t'_n$. We have that $F\ t_1...t_n = F\ t'_1...t'_n$ iff $t_i = t'_i$ for $i = 1, ..., n$. Our task is to associate appropriate proof terms to each type equation. For this purpose, we assume the following set of composition and decomposition functions whose types are recorded in some initial proof environment $\Gamma_{proof} \supseteq$

$$\left\{\begin{array}{ll} \mathit{compFf} & : \forall\overline{a,b}.(a_1 \to b_1, b_1 \to a_1) \to ... \to (a_n \to b_n, b_n \to a_n) \to (F\ \bar{a} \to F\ \bar{b}) \\ \mathit{compFg} & : \forall\overline{a,b}.(a_1 \to b_1, b_1 \to a_1) \to ... \to (a_n \to b_n, b_n \to a_n) \to (F\ \bar{b} \to F\ \bar{a}) \\ \mathit{decompFf}_i & : \forall\overline{a,b}.(F\ \bar{a} \to F\ \bar{b}, F\ \bar{b} \to F\ \bar{a}) \to (a_i \to b_i) \\ \mathit{decompFg}_i & : \forall\overline{a,b}.(F\ \bar{a} \to F\ \bar{b}, F\ \bar{b} \to F\ \bar{a}) \to (b_i \to a_i) \end{array}\right\}$$

It is straightforward to verify that proof terms are typable in System F. Let $C \equiv (f_1,g_1) : t_1 = t'_1 \wedge ... \wedge (f_n,g_n) : t_n = t'_n$. We write $C \rightsquigarrow \Gamma$ to denote the conversion of constraint C into a type environment $\Gamma = \{f_1 : t_1 \to t'_1, g_1 : t'_1 \to t_1, ..., f_n : t_n \to t'_n, g_n : t'_n \to t_n\}$.

Lemma 1 (Typability). *Let $C \vdash_= (E_1, E_2) : t = t'$ such that $C \rightsquigarrow \Gamma$. Then, $\Gamma \cup \Gamma_{proof} \vdash_F E_1 : t \to t'$ and $\Gamma \cup \Gamma_{proof} \vdash_F E_2 : t' \to t$.*

Next, we show that well-typed proof terms are sound. We follow [38] and give meaning to primitive functions such as $\mathit{decompFf}_i$,$\mathit{decompFg}_i$,*compFf* and *compFg* in terms of a function $\delta : (\mathit{Primitive} \mid \overline{\mathit{ClosedVal}}) \to \mathit{ClosedVal}$ that interprets the application of primitive functions to closed input values and yields closed output values. Thus, we define the rewrite rule

$$\mathit{primitive}\ v_1...v_n \rightarrowtail \delta(c \mid v_1...v_n) \quad \text{if } \delta(\mathit{primitive} \mid v_1...v_n) \text{ is defined}$$

For each composition and decomposition primitive for type $F\ a_1...a_n$ we define

$$\begin{array}{ll} \delta(\mathit{compFf} \mid [t_1...t_n\ t'_1...t'_n]\ (v_1,v'_1)...(v_n,v'_n)) & = \lambda x.x \\ \delta(\mathit{compFf} \mid [t_1...t_n\ t'_1...t'_n]\ (v_1,v'_1)...(v_n,v'_n)) & = \lambda x.x \\ \delta(\mathit{decompFf}_i \mid [t_1...t_n\ t'_1...t'_n]\ (v_1,v_2)) & = \lambda x.x \\ \delta(\mathit{decompFg}_i \mid [t_1...t_n\ t'_1...t'_n]\ (v_1,v_2)) & = \lambda x.x \end{array}$$

if $t_j \equiv t'_j$ (i.e. t_j and t'_j are syntactically equal) for $j = 1, ..., n$.

Note that the δ function is undefined in case t_j and t'_j are syntactically different. Hence, there is the danger that the reduction of well-typed proof terms gets stuck. The important observation is that $\vdash_= (E_1, E_2) : t_1 = t_2$ implies that t_1 and t_2 must be syntactically equal. Hence, during the reduction of proof terms E_1 and E_2 we can guarantee that whenever one of the above primitives is applied to a sequence of closed values $[t_1...t_n\ t'_1...t'_n]\ v_1...v_n$ the condition $t_j \equiv t'_j$ for $j = 1, ..., n$ is always satisfied. This observation is formalized in the following result. For clarity, we enumerate the individual statements.

Lemma 2 (Subject Reduction). *Let $C \vdash_= (E_1, E_2) : t_1 = t_2$ such that (1) for each $(f, g) : t = t' \in C$ we have that $t \equiv t'$, (2) $E_1 \to E'_1$, (3) $E_2 \to E'_2$ and (4) $C \rightsquigarrow \Gamma$ for some E'_1, E'_2 and Γ. Then, (5) $t_1 \equiv t_2$, (6) $\Gamma \cup \Gamma_{proof} \vdash_F E'_1 : t_1 \to t_2$ and (7) $\Gamma \cup \Gamma_{proof} \vdash_F E'_2 : t_2 \to t_1$.*

In order to obtain type soundness, we yet need to verify that well-typed proof terms do not get stuck. We omit the straightforward details. We summarize the results of this section. Proof terms are well-typed and evaluate to the identity function. [2]

Theorem 1. *Let $\emptyset \vdash_= (E_1, E_2) : t_1 = t_2$. Then, (1) $t_1 \equiv t_2$, (2) $\Gamma_{proof} \vdash_F E_1 : t_1 \to t_2$, (3) $\Gamma_{proof} \vdash_F E_2 : t_2 \to t_1$, (4) $E_1 \rightarrowtail^* \lambda x.x$ and (5) $E_2 \rightarrowtail^* \lambda x.x$.*

Note that proof term construction is decidable. That is, given C and t_1, t_2 where C has a unifier, there is a decidable algorithm which either finds E_1 and E_2 such that $C \vdash_= (E_1, E_2) : t_1 = t_2$ or $C \supset t_1 = t_2$ does not hold. Due to space limitations, we refer the interested reader to [34].

4.3 Type-Directed Translation Scheme

We are in the position to define the translation of GADTs to System F_s. For translation purposes, we assume that type constraints $t_1 = t_2$ in GADT constructors are annotated with proof terms (f, g) where f and g are distinct variables. In a preprocessing step, we translate each GADT constructor

$$K : \forall \bar{a}, \bar{b}.(f_1, g_1) : t'_1 = t'''_1 \wedge ... \wedge (f_n, g_n) : t''_n = t'''_n) \Rightarrow t_1 \to ... \to t_n \to F\ \bar{a}$$

to a System F_s constructor

$$K_{\bar{b}} : \forall \bar{a}, \bar{b}. < (t''_1 \to t'''_1, t'''_1 \to t''_1, ..., t''_n \to t'''_n, t'''_n \to t''_n) > \to t_1 \to ... \to t_n \to F\ \bar{a}$$

where type constraints are turned into additional (staged) arguments.

The actual translation scheme is formulated in terms of judgments $C, \Gamma \vdash e : t \rightsquigarrow E$. For simplicity, we only consider the most interesting rules. Details are in Figure 4.

In rule (Var), we must provide proof terms to satisfy the instantiated constraint $[\overline{t/a}]C'$. For convenience, we assume that there is a fixed order among

[2] Strictly speaking, when applied to a value proof terms behave like the identity function, see rule (Trans) in Figure 3. Though, we could introduce a transitivity primitive which always evaluates to the identity similar to the composition and decomposition primitives.

$$\text{(Var)}\quad \frac{(x : \forall \bar{a}.C' \Rightarrow t') \in \Gamma \quad C \equiv (f_1, g_1) : t_1 = t_1' \wedge ... \wedge (f_n, g_n) : t_n = t_n' \quad \text{for each } t_i'' = t_i''' \in [\overline{t/a}]C' \text{ we have that } C \vdash_= (E_i, E_i') : t_i'' = t_i''''}{C, \Gamma \vdash x : [\overline{t/a}]t' \rightsquigarrow x\ [\bar{t}]\ <(E_1, E_1', ..., E_n, E_n')>}$$

$$\text{(Eq)}\quad \frac{C, \Gamma \vdash e : t \rightsquigarrow E \quad C \vdash_= (E', E'') : t = t'}{C, \Gamma \vdash e : t' \rightsquigarrow <E'>\ E}$$

$$\text{(}\forall\text{Intro)}\quad \frac{\begin{array}{c} C_1 \wedge C_2, \Gamma \vdash e : t \rightsquigarrow E \quad \bar{a} \cap fv(C_1, \Gamma) = \emptyset \\ C_2 \equiv (f_1, g_1) : t_1 = t_1' \wedge ... \wedge (f_n, g_n) : t_n = t_n' \end{array}}{\begin{array}{c} C_1 \wedge \exists \bar{a}.C_2, \Gamma \vdash e : \forall \bar{a}.C_2 \Rightarrow t \\ \rightsquigarrow \\ \Lambda \bar{a}.\lambda <(f_1, g_1, ..., f_n, g_n)> :< (t_1 \to t_1', t_1' \to t_1, ..., t_n \to t_n', t_n' \to t_n)> .E \end{array}}$$

$$\text{(Pat)}\quad \frac{\begin{array}{c} p : t_1 \vdash \forall \bar{b}.(C_p \blacktriangleright \Gamma_p) \rightsquigarrow P \\ \bar{b} \cap fv(C, \Gamma, t_2) = \emptyset \quad C \wedge C_p, \Gamma \cup \Gamma_p \vdash e : t_2 \rightsquigarrow E \end{array}}{C, \Gamma \vdash p \to e : t_1 \to t_2 \rightsquigarrow P \to E}$$

$$\text{(P-Var)}\quad x : t \vdash (True \blacktriangleright \{x : t\}) \rightsquigarrow x$$

$$\text{(P-K)}\quad \frac{\begin{array}{c} K : \forall \bar{a}, \bar{b}.C \Rightarrow t_1 \to ... \to t_n \to F\ \bar{a} \\ C \equiv (f_1, g_1) : t_1'' = t_1''' \wedge ... \wedge (f_n, g_n) : t_n'' = t_n''') \quad \bar{b} \cap \bar{a} = \emptyset \\ p_i : [\overline{t'/a}]t_i \vdash \forall \overline{b_i'}.(C_i' \blacktriangleright \Gamma_{p_i}) \rightsquigarrow P_i \quad \text{for } i = 1, ..., n \end{array}}{\begin{array}{l} K\ p_1 ... p_n : F\ \bar{t} \vdash \\ \end{array} \begin{array}{c} \forall \overline{b_1'}, ..., \overline{b_n'}, \bar{b}.C_1' \wedge (.. \wedge C_n' \wedge [\overline{t'/a}]C \blacktriangleright \Gamma_{p_1} \cup ... \cup \Gamma_{p_n}) \rightsquigarrow \\ K_{\bar{b}}\ <(f_1, g_1, ..., f_n, g_n)>\ P_1 ... P_n \end{array}}$$

Fig. 4. Type-Directed Translation (interesting cases)

type equations $t_i'' = t_i'''$ in $[\overline{t/a}]C'$. We are a bit sloppy here and simply drop the proof terms associated to $t_i' = t_i''$. Based on Assumption 1 we know that $C \vdash_= E_i : t_i' = t_i''$ holds iff $C \supset t_i' = t_i''$. The same observations applies in case of rule (Eq) where we apply proof term E' to maintain well-typing of expressions. In rule (∀Intro), we turn constraints in type schemes into additional proof term parameters. For convenience, we use pattern syntax in λ-abstraction. The main tasks of rules (Pat), (Pat-Var) and (Pat-K) are to translate patterns which now may contain proof terms. The left out rules are the familiar ones for translating Hindley/Milner to System F [10].

We can verify that resulting expressions are typable in System F_s. E.g., consider rule (Eq). Let $C \rightsquigarrow \Gamma_C$. We write Γ_C^2 and Γ_{proof}^2 to denote that the type bindings are at stage 2. Then, $\Gamma_C^2 \cup \Gamma_{proof}^2 \vdash_{F_s}^2 E' : t \to t'$ (follows from Lemma 1). Via the System F_s rules (<>↓) and (App), we can conclude that $\Gamma_C^2 \cup \Gamma_{proof}^2 \cup \Gamma \vdash_{F_s}^1 <E'>\ E : t'$. Similar reasoning steps apply to the other rules.

We can also verify that the evaluation of resulting expressions cannot go wrong. System F_s inherits the System F type soundness result. By construction,

the evaluation of proof terms is independent of the surrounding program text (this is explicitly enforced by the System F_s typing rules). Hence, Theorem 1 (proof terms cannot go wrong) is applicable and therefore the translated program cannot go wrong.

We summarize.

Theorem 2 (Type Preservation and Soundness). *Let* $True, \emptyset \vdash e : t \rightsquigarrow E$. *Then,* $\Gamma^2_{proof} \vdash^1_{F_s} E : t$ *and* $E \rightarrowtail^* v$ *for some value* v. [3]

Theorem 1 (proof terms evaluate to the identity) also guarantees that proof terms do not affect the semantic meaning of the GADT program. Hence, we can safely erase them. Hence, the original GADT program is equivalent to the resulting System F_s program after erasing proof terms and types.

Corollary 1. *Let* $\emptyset \vdash e : t \rightsquigarrow E$. *Then,* $\mathcal{E}(E) = e$.

The erasure function $\mathcal{E}()$ is defined following the structure of possible target and pattern terms:

$$\begin{array}{lll} \mathcal{E}(E\ [\overline{T}]) = \mathcal{E}(E) & \mid \mathcal{E}(\Lambda\bar{a}.E) = \mathcal{E}(E) & \mid \mathcal{E}(E_1\ E_2) = \mathcal{E}(E_1)\ \mathcal{E}(E_2) \\ \mathcal{E}(\lambda x : T.E) = \lambda x.\mathcal{E}(E) & \mid \mathcal{E}(< E_1 >\ E_2) = \mathcal{E}(E_2) & \mid \mathcal{E}(x\ [\overline{T}]\ < E >) = x \end{array}$$

$$\begin{array}{l} \mathcal{E}(\mathsf{let}\ g : T = E\ \mathsf{in}\ E') = \mathsf{let}\ g = \mathcal{E}(E)\ \mathsf{in}\ \mathcal{E}(E') \quad \mid \quad \mathcal{E}(x) = x \\ \mathcal{E}(\mathsf{case}\ (T)\ E\ \mathsf{of}\ [P_i \rightarrow E_i]_{i\in I}) = \mathsf{case}\ \mathcal{E}(E)\ \mathsf{of}\ [\mathcal{E}(P_i) \rightarrow \mathcal{E}(E_i)]_{i\in I} \\ \mathcal{E}(K\ < E >\ P_1 ... P_n) = K\ \mathcal{E}(P_1) ... \mathcal{E}(P_n) \quad \mid \quad \mathcal{E}(x : T) = x \end{array}$$

Another immediate consequence is the following result.

Corollary 2. *Well-typed GADT programs cannot go wrong.*

5 Direct Translation of GADTs to Existential Types

A number of authors [1, 4, 5, 22, 37] have shown by example that GADT style behavior can *often* be directly encoded in terms of existing languages such as Haskell [11]. The following (non-sensical) example shows that a source-level encoding is *not always* possible.

Example 2. Consider

```
data Erk a = (Int->Int=a->Int) => I a
f :: Erk a->a
f (I x) = 0
```

which is clearly type correct. Note that $Int \rightarrow Int = a \rightarrow Int$ iff $Int = a$.

Our translation method yields

```
data Erk a = I <((Int->Int)->(a->Int),(a->Int)->(Int->Int))> a
f = Λ y:Erk a.case y of
        I <(f,g)> x -> <decomp [a Int Int Int] (f,g)> 0
```

[3] In case we extend the source language with recursive functions we need the additional case that reduction of E may not terminate.

where $decomp : \forall a_1, a_2, a_3, a_4.((a_1 \to a_2) \to (a_3 \to a_4), (a_3 \to a_4) \to (a_1 \to a_2)) \to (a_1 \to a_3)$ and $\delta(decomp \blacksquare [t_1 t_2 t_3 t_4](v_1, v_2)) = \lambda x.x$ if $t_1 \equiv t_3$ and $t_2 \equiv t_4$.

Assume we want to represent the System F_s program in Haskell. Then, we need to find a "closed" definition of *decomp*. However, there seems to be no Haskell expression of type $\forall a_1, a_2, a_3, a_4.((a_1 \to a_2) \to (a_3 \to a_4), (a_3 \to a_4) \to (a_1 \to a_2)) \to (a_1 \to a_3)$.

The "decomposition" issue has already been pointed out by Chen, Zhu and Xi [4]. [4] The methods developed here allow us to establish sufficient conditions under which a translation from GADT to existential types is possible.

Definition 1 (Decomposition). *We say a GADT program satisfies the* decomposition *condition if all proof terms arising in a GADT typing derivation are well-typed in a system with existential types.*

Immediately, we can derive the following result from Theorem 2.

Corollary 3. *GADTs can be translated to existential types if the decomposition condition is satisfied.*

We have conducted a survey of all (sensible) GADT examples we found in the literature. The surprising observation is that all examples satisfy the decomposition condition. Hence, a translation to existential types is possible in theory. A selection of GADT programs and their translation to existential types can be found here [34].

Obviously, the direct encoding of GADTs in terms of existential types (if possible at all) is not practical due to a high run-time overhead of proof terms. We know that at run-time all proof terms evaluate to the identity. However, in order to ensure that composition/decomposition functions are well-typed with existential types, we have to traverse structured data (such as lists, trees etc) and repeatedly apply (proof term) functions to each element to get the types "right". E.g., here is the closed definition of the composition function for lists.

$$(\text{List}\uparrow)\ \frac{C \vdash_= (f, g) : a = b}{C \vdash_= (\lambda x.map\ f\ x, \lambda x.map\ g\ x) : [a] = [b]}$$

On the other hand, the composition/decomposition primitives in System F_s never inspect the structure of the input arguments. They immediately evaluate to the identity.

In this context, it is interesting to point out that the type-preserving defunctionalization transformation of polymorphic programs to System F extended with GADTs by Pottier and Gauthier [24] *always* satisfies the decomposition condition. This follows straightforwardly from their formal results (proofs of Lemmas 4.1 and 4.2 in [24]). More details can be found here [34].

[4] This is hardly surprising given that similar situations arise when translating type class programs [9]. E.g., we cannot decompose the equality type class $Eq\ [a]$ into $Eq\ a$ for any a.

6 Extensions with User-Specifiable Type Equality

We provide evidence that our translation method scales to extensions where type equality is user-specifiable. For concreteness, we use the associated types (AT) formalism [2] to represent type equality definitions. Our task is to represent AT type definitions by extending the proof system in Section 4.2.

As a simple example, we define a *extended* GADT to represent a while language which satisfies a resource usage policy specified in terms of a DFA. The DFA transitions are specified via AT type functions. In the extended GADT, we make use of these type functions.

Example 3. Here is an excerpt of a possible specification.

```
-- states
type S0; type S1
-- alphabet
type Open; type Write; type Close
-- transition function
type Delta S0 Open = S1     -- (1)
type Delta S1 Close = S0    -- (2)
type Delta S1 Write = S1    -- (3)

-- Resource GADT
data Cmd p q =
    forall r. Seq (Cmd p r) (Cmd r q)
  | ITE Exp (Cmd p q) (Cmd p q)
  | (p=q) => While Exp (Cmd p q)
    -- while state is invariant
  | (Delta p Open = q) => OpenF
  | (Delta p Close = q) => CloseF
  | (Delta p Write = q) => WriteF
```

We introduce type function `Delta` and some type constants to specify the resource DFA in terms of (type) function definitions (1-3). We make use of these assumptions when defining the GADT `Cmd p q` where type parameters `p` and `q` represent the input and output state, before and after execution of the command.

We can translate programs making use of such extended GADTs by appropriately extending our proof system in Section 4.2. For the above example, we define

$$\text{(D1)}\ C \vdash_{=} (f_1, g_1) : \mathit{Delta}\ S0\ \mathit{Open} = S1$$

$$\text{(D2)}\ C \vdash_{=} (f_2, g_2) : \mathit{Delta}\ S1\ \mathit{Close} = S0$$

$$\text{(D3)}\ C \vdash_{=} (f_3, g_3) : \mathit{Delta}\ S1\ \mathit{Write} = S1$$

where $f_1 : \mathit{Delta}\ S0\ \mathit{Open} \to S1$, $g_1 : S1 \to \mathit{Delta}\ S0\ \mathit{Open}$ etc are in Γ_{proof}, and $\delta(f_1 \blacksquare v) = \lambda x.x$ if $\mathit{typeof}(v) = S1$, $\delta(g_1 \blacksquare v) = \lambda x.x$ if $\mathit{typeof}(v) = \mathit{Delta}\ S0\ \mathit{Open}$ etc. Function $\mathit{typeof}()$ retrieves the type of a value.

Let's consider another (contrived) example. We define `type F [a] = [F a]`, i.e. for any a we have that $F\ [a]$ and $[F\ a]$ are equal. The corresponding proof rule is as follows.

$$\text{(F)}\ C \vdash_{=} (\mathit{upF}\ [t], \mathit{downF}\ [t]) : F\ [t] = [F\ t]$$

where $upF : \forall a.F\ [a] \rightarrow [F\ a]$, $downF : \forall a.[F\ a] \rightarrow F\ [a] \in \Gamma_{proof}$ and $\delta(upF \blacksquare [t]\ v) = \lambda x.x$, $\delta(downF \blacksquare [t]\ v) = \lambda x.x$ if $F\ [t] = [F\ t]$.

Note that the additional primitives complicate the semantics of System F_s. E.g., when executing primitive upF in context $[t]\ v$ we need to test that $F\ [t] = [F\ t]$ holds. Depending on the set of AT definitions deciding type equality may become undecidable (e.g. consider `type F [a] = [F [a]]`). The important point to note is that the test for type equality is only necessary to verify soundness of our (extended) translation scheme. At run-time proof terms evaluate to the identity. They can be safely erased and therefore they do not incur any additional run-time cost. A second important point is that our target language can host source programs with a in general undecidable equational theory. We only require that when translating a specific source program we must provide a (type equality) proof which can be translated into a proof term.

7 Conclusion

We have shown how to translate GADTs to System F_s, a minor extension of System F with existential types and staged proof terms. The advantage of our method is that type equality is only necessary to establish soundness of the translation scheme. Type inference on the source program has already done all the hard work for us, e.g. verifying type equality. There seems no point in repeating such tests in the target language. Instead, we apply the "proofs are programs" principle and compile type equation proofs into System F typable proof terms. We could provide evidence that our methods extends to systems where type equality is user-specifiable as long as we find a suitable System F representation. Thus, type checking of target programs remains as simple as type checking in System F. At run-time, proof terms evaluate to the identity, therefore, they can be safely erased. Hence, our method does not impose any additional run-time overhead.

In case of user-specifiable type equality we need to ensure that the additional proof rules still satisfy Theorem 1 which is crucial to guarantee soundness of our translation method. In this context, it may be worthwhile to consider extending System F itself with some form of type equality. These are interesting topics which deserve further studies in the future.

We also established sufficient conditions under which GADTs can be translated to existential types as found in Haskell (Section 5). This result is not of high practical relevance, but confirms the observation made by a number of authors [1, 4, 5, 22, 37] that the concepts of GADTs and existential types are fairly close and often equivalent in terms of expressive power.

We note that System F_s is more general than necessary. To host GADT programs, we only need "two stage" expressions, patterns and types (see Section 4). However, we envision extensions of GADTs where types are classified by kinds (in the same way values are classified by types). Then, type equation proof terms may contain (staged) kind proof terms. We believe that System F_s is already equipped to deal with the translation of such cases.

References

1. A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *Proc. of ICF'02*, pages 157–166. ACM Press, 2002.

2. M. Chakravarty, G. Keller, and S. Peyton Jones. Associated types synonyms. In *Proc. of ICFP'05*, pages 241–253. ACM Press, 2005.
3. M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proc. of POPL'05*, pages 1–13. ACM Press, 2005.
4. C. Chen, D. Zhu, and H. Xi. Implementing cut elimination: A case study of simulating dependent types in Haskell. In *Proc. of PADL'04*, volume 3057 of *LNCS*, pages 239–254. Springer-Verlag, 2004.
5. J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. of Haskell Workshop'02*, pages 90–104. ACM Press, 2002.
6. J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.
7. Glasgow haskell compiler home page. http://www.haskell.org/ghc/.
8. J. Girard. *Interpretation fonctionnelle et elimination des coupures de l'arithmetique d'ordre superieur*. Thèse d'état, Université Paris 7, June 1972.
9. C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. In *ESOP'94*, volume 788 of *LNCS*, pages 241–256. Springer-Verlag, April 1994.
10. R. Harper and J. C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.
11. Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.
12. M. P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 1995.
13. M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.
14. S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types, 2004. Submitted to POPL'05.
15. S. Kaes. Parametric overloading in polymorphic programming languages. In *In Proc. of ESOP'88*, volume 300 of *LNCS*, pages 131–141. Springer-Verlag, 1988.
16. J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.
17. K. Läufer and M. Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, 1992.
18. K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.
19. Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.
20. H. Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proc. of ICFP'05*, pages 54–65. ACM Press, 2005.
21. M. Odersky, M. Sulzmann, and M Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
22. E. Pasalic. *The Role of Type Equality in Meta-Programming*. PhD thesis, Oregon Health & Science University, OGI School of Science & Engineering, September 2004.
23. B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
24. F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Proc. of POPL'04*, pages 89–98. ACM Press, January 2004.
25. F. Pottier and Y. Rgis-Gianas. Towards efficient, typed LR parsers. Draft paper, September 2004.
26. J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.
27. Z. Shao. An overview of the FLINT/ML compiler. In *Proc. of ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, 1997.
28. Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proc. of POPL'02*, pages 217–232. ACM Press, 2002.

29. T. Sheard. Omega. http://www.cs.pdx.edu/ sheard/Omega/index.html.
30. T. Sheard. Languages of the future. *SIGPLAN Not.*, 39(10):116–119, 2004.
31. V. Simonet and F. Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, January 2005.
32. P. J. Stuckey and M. Sulzmann. Type inference for guarded recursive data types. http://www.comp.nus.edu.sg/˜ sulzmann, 2005. Draft.
33. M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.
34. M. Sulzmann and M. Wang. A systematic translation of guarded recursive data types to existential types. Technical Report TR22/04, The National University of Singapore, 2004.
35. W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
36. P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76. ACM Press, 1989.
37. S. Weirich. Type-safe cast: (functional pearl). In *Proc. of ICFP'00*, pages 58–67. ACM Press, 2000.
38. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
39. H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. of POPL'03*, pages 224–235. ACM Press, 2003.
40. C. Zenger. *Indizierte Typen*. PhD thesis, Universität Karlsruhe, 1999.

A Translating Type Classes in the Presence of Type Improvement

User-specifiable type equations also arise in the context of type classes [36, 15] with type improvement [12]. For concreteness, we consider the associated types (ATs) [3, 2] (improvement) formalism. Similar observations apply to other formalisms such as functional dependencies [13].

Example 4. Consider the following AT program

```
class C a where
  type F a            -- (1)
class D a where
  d :: C a => F a->F a
instance C Int where
  type F Int = Bool   -- (2)
instance D Int where
  d x = x && True     -- (3)
```

Type class `C` introduces a type function `F` (see (1)) whose concrete definitions are provided by the instance declarations (see (2)). Effectively, we state that the type `F Int` can be *improved* to `Int` and vice versa. Type class `D` introduces a method which is locally constrained by the type class `C`. Note that the second input and result type are constrained by the type `F a`.

We verify type correctness of instance definition (3). Let us assume that `x` has type `F Int`. Based on (2) we can improve `x`'s type to `Bool`. Hence, expression `x && True` has type `Bool`. which can be improved to `F Int`. Hence, the definition is correct.

Existing translation schemes [3, 2] use System F as their target language. For the above program, the only choice we have is to replace type `F a` by a universally quantified variable `b` and add `b` as a parameter to type class `C`. Then, the translation of definition (3) is as follows.

```
data C a b = ...
d = Λ b λ d:C b. λ x:b. x && True
```

However, the above translation is not type correct. Variable `x` has the universal type `b` but is conjoined with `True`. Thus, `x` must have type `Bool` which leads to a contradiction.

The above program is contrived, but similar examples crop up all the time on the Haskell mailing list. The bottom line is this. Existing implementations [7] rely on System F as their target language. Hence, some sensible programs are rejected.

On the other hand, in a translation scheme based on System F_s we can represent the type equation `F Int=Bool` via primitives `f1:F Int->Bool` and `g1:Bool->F Int`. Thus, definition (3) can be translated as follows while preserving well-typing.

```
d = λ d:C Int. λ x:F Int. <g1>((<f1> x) && Bool)
```

Here is another AT example.

Example 5. Consider

```
type F a
data G a = forall b. (F a=b) => K (b->b)
f (K h1) (K h2) = K (h1.h2)
```

translates to

```
data G a = forall b. (F a=b) => K (F a->b,b->F a) (b->b)
f (K <(f1,g1)> h1) (K <(f2,g2>) h2) =
  K <(f2,g2)> (\x->(<f2> (<g1> (h1 (<f1> (<g2> (h2 x)))))))
```