A Framework for Extended Algebraic Data Types

Martin Sulzmann[1], Jeremy Wazny[3] and Peter J. Stuckey[2,3]

[1] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg
[2] NICTA Victoria Laboratory
[3] Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{jermyrw,pjs}@cs.mu.oz.au

Abstract. There are a number of extended forms of algebraic data types such as type classes with existential types and generalized algebraic data types. Such extensions are highly useful but their interaction has not been studied formally so far. Here, we present a unifying framework for these extensions. We show that the combination of type classes and generalized algebraic data types allows us to express a number of interesting properties which are desired by programmers. We support type checking based on a novel constraint solver. Our results show that our system is practical and greatly extends the expressive power of languages such as Haskell and ML.

1 Introduction

Algebraic data types enable the programmer to write functions which pattern match over user-definable types. There exist several extensions of algebraic data types which increase the expressiveness of the language significantly. Läufer and Odersky [LO94] consider the addition of (boxed) existential types whereas Läufer [Läu96] was the first to introduce a combination of single-parameter type classes and existential types [LO94]. Xi, Chen and Chen [XCC03] introduced yet another extension of algebraic data types known as guarded recursive data types (GRDTs). GRDTs are largely equivalent to Cheney's and Hinze's first-class phantom types [CH03] and Peyton Jones's, Washburns' and Weirich's generalized algebraic data types (GADTs) [JWW04].[4] All these extensions are very interesting but have largely been studied independently.

Here, we present a system which unifies these seemingly unrelated extensions, something which has not been studied formally before.

Specifically, our contributions are:

– We formalize an extension of Hindley/Milner where the types of constructors of algebraic data types may be constrained by type equations *and* type classes (Section 4). Such a system of *extended* algebraic data types subsumes GADTs and type classes with extensions [JJM97,Jon00].

[4] For the purposes of this paper, we will use the term GADTs which appears to be gaining popularity.

– An important point of our system is that GADTs and type classes can interact freely with each other. Thus, we can express some interesting properties which are desired by programmers (Section 4.1).
– We support type checking based on a novel checking method for implication constraints (Section 5).
– We have implemented the type checker as part of the Chameleon system [SW] (experimental version of Haskell).

We continue in Section 2 where we introduce some basic notations and assumptions used throughout the paper. In Section 3 we review type classes with existential types and GADTs and summarize their differences and commonalities. Related work is discussed in Section 6. We conclude in Section 7. Note that we use Haskell-style syntax in example programs throughout the paper.

Additional details including a description of the semantic meaning of programs and its type soundness proof can be found in an accompanying technical report [SWP06].

2 Preliminaries

We write $\bar{o}$ to denote a sequence of objects o_1,...,o_n. As it is common, we write Γ to denote an environment consisting of a sequence of type assignments $x_1 : \sigma_1, ..., x_n : \sigma_n$. Types σ will be defined later. We commonly treat Γ as a set. We write "$-$" to denote set subtraction. We write $\mathit{fv}(o)$ to denote the set of free variables in some object o with the exception that $\mathit{fv}(\{x_1 : \sigma_1, ..., x_n : \sigma_n\})$ denotes $\mathit{fv}(\sigma_1, ..., \sigma_n)$. In case objects have binders, e.g. $\forall a$, we assume that $\mathit{fv}(\forall a.o) = \mathit{fv}([b/a]o) - \{b\}$ where b is a fresh variable and $[b/a]$ a renaming.

We generally assume that the reader is familiar with the concepts of substitutions, unifiers, most general unifiers (m.g.u.) etc [LMM87] and first-order logic [Sho67]. We write $[\overline{t/a}]$ to denote the simultaneous substitution of variables a_i by types t_i for $i = 1, .., n$. 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 use common notation for Boolean conjunction ($\wedge$), implication ($\supset$) and universal ($\forall$) and existential quantifiers ($\exists$). Often, we abbreviate $\wedge$ by "," and use set notation for conjunction of formulae. We sometimes use $\bar{\exists}_V.\mathit{Fml}$ as a short-hand for $\exists \mathit{fv}(\mathit{Fml}) - V.\mathit{Fml}$ where Fml is some first-order formula and V a set of variables, that is existential quantification of all variables in Fml apart from V. We write $\models$ to denote the model-theoretic entailment relation. When writing logical statements we often leave (outermost) quantifiers implicit. E.g., let Fml_1 and Fml_2 be two formulae where Fml_1 is closed (contains no free variables). Then, $\mathit{Fml}_1 \models \mathit{Fml}_2$ is a short-hand for $\mathit{Fml}_1 \models \forall \mathit{fv}(\mathit{Fml}_2).\mathit{Fml}_2$ stating that in any (first-order) model for Fml_1 formula $\forall \mathit{fv}(\mathit{Fml}_2).\mathit{Fml}_2$ is satisfied.

3 Background: Type classes with existential types and GADTs

In our first example program, we make use of existential types as introduced by Läufer and Odersky [LO94].

```
data KEY = forall a. Mk a (a->Int)
g (Mk x f) = f x
```

The algebraic data type KEY has one constructor `Mk` of type $\forall a.a \rightarrow (a \rightarrow Int) \rightarrow KEY$. Note that variable a does not appear in the result type. In the Haskell syntax we indicate this explicitly via `forall a`. When constructing a value of type KEY we are able to hide the actual values involved. Effectively, a refers to an existentially bound variable. [5] Thus, when pattern matching over KEY values we should not make any assumptions about the actual values involved. This is the case for the above program text. We find that the type of `g` is $KEY \rightarrow Int$.

The situation is different in case of

```
g1 (Mk x f) = (f x, x)
```

The type of `x` escapes as part of the result type. However, we have no knowledge about the actual type of `x`. Hence, function `g1` should be rejected.

In some subsequent work, Läufer [Läu96] considers a combination of single-parameter type classes [WB89] and existential types. Consider

```
class Key a where getKey::a->Int
data KEY2 = forall a. Key a => Mk2 a
g2 (Mk2 x) = getKey x
```

where the class declaration introduces a single-parameter type class Key with method declaration `getKey` : $\forall a.Key\ a \Rightarrow a \rightarrow Int$. We use Key to constrain the argument type of constructor `Mk2`, i.e. `Mk2` : $\forall a.Key\ a \Rightarrow a \rightarrow KEY2$. The pattern `Mk2 x` gives rise to $Key\ a$ and assigns the type a to `x`. In the function body of `g2`, expression `getKey x` has type Int and gives rise to $Key\ a$ which can be satisfied by the type class arising out of the pattern. Hence, function `g2` is of type $KEY2 \rightarrow Int$.

GADTs are one of the latest extensions of the concept of algebraic data types. They have attracted a lot of attention recently [SP04,PG04,Nil05]. The novelty of GADTs is that the (result) types of constructor may differ. Thus, we may make use of additional type equality assumptions while typing the body of a pattern clause.

Here, we give excerpts of a standard program which defines a strongly-typed evaluator for a simple term language. We use the GADT notation as implemented in GHC 6.4 [GHC].

```
data Term a where
     Zero : Term Int
     Pair : Term b->Term c->Term (b,c)
eval :: Term a -> a
eval Zero = 0
eval (Pair t1 t2) = (eval t1, eval t2)
```

The constructors of the above GADT `Term a` do *not* share the same (result) type (which is usually required by "standard" algebraic data types). In case of `Zero` the variable `a` in the GADT `Term a` is equal to `Int` whereas in case of `Pair` the variable `a` is equal to `(b,c)` for some `b` and `c`. Effectively, the constructors

[5] It may helpful to point out that $\forall a.a \rightarrow (a \rightarrow Int) \rightarrow KEY$ is equivalent to $(\exists a.a \rightarrow (a \rightarrow Int)) \rightarrow KEY$. Hence, "existential" variables are introduced with the "universal" `forall` keyword

mimic the typing rules of a simply-typed language. That is, we can guarantee that all constructed terms are well-typed. The actual novelty of GADTs is that when pattern matching over GADT constructors, i.e. deconstructing terms, we can make use of the type equality assumptions implied by constructors to type the body of pattern clauses. E.g. in case of the second function clause we find that `Pair t1 t2` has type $Term\ a$. Hence, `t1` has type $Term\ b$ and `t2` has type $Term\ c$ where $a = (b, c)$. Hence, `(eval t1, eval t2)` has type (b, c) which is equivalent to a under the constraint $a = (b, c)$. The constraint $a = (b, c)$ is only available while typing the body of this particular function clause. That is, this constraint does not float out and therefore does not interact with the rest of the program. Hence, the type annotation `eval::Term a->a` is correct.

The short summary of what we have seen so far is as follows. Typing-wise the differences between GADTs and type classes with existential types are marginal. Both systems are extensions of existential types and both make temporary use of primitive constraints (either type equalities or type class constraints) arising out of patterns while typing the body of a function clause. E.g, in function `g2` we temporarily make use of $Key\ a$ arising out of pattern `Mk2 x` whereas in the second clause of function `eval` we temporarily make use of $a = (b, c)$ arising out of pattern `Pair t1 t2`. A subtle difference is that the GADT type system has an additional typing rule to change the type of expressions under type equation assumptions (see upcoming rule (Eq) in Figure 2). Such a rule is lacking in the theory of qualified types [Jon92] which provides the basis for type classes with extensions. The consequence is that we cannot fully mimic GADTs via the combination of multi-parameter type classes [JJM97] with existential types and functional dependencies [Jon00]. We elaborate on this issue in more detail.

The following program is accepted in GHC.

```
class IsInt a b | ->a
instance IsInt Int b
class IsPair a b c | b c->a
instance IsPair (b,c) b c
data Term2 a = forall b. IsInt a b => Zero b
             | forall b c. IsPair a b c => Pair (Term b) (Term c)
```

The declaration `IsInt a b |->a` introduces a multi-parameter type class `IsInt`. The functional dependency `|->a` in combination with the instance declaration enforces that if we see `IsInt a b` the variable `a` will be improved by `Int`. The second parameter is somewhat redundant but necessary due to a GHC condition which demands that at least one type class parameter in a data definition must refer to an existential variable. Similarly, the declarations for type class `IsPair` enforce that in `IsPair a b c` the variable `a` will be improved by `(b,c)`. The up-shot of this encoding of type equations in terms of functional dependencies is that we can use the `Term2 a` type (and its constructors) instead of the GADT `Term a` to ensure that only well-typed terms will ever be constructed.

However, we cannot use the `Term2 a` type to deconstruct values. The following definition does not type check in GHC.

```
eval2 :: Term2 a -> a
eval2 (Zero _) = 0
eval2 (Pair t1 t2) = (eval t1, eval t2)
```

The pattern match in the first clause gives rise to the constraint `IsZero a b` which enforces that `a` is equal to `Int`. In the GHC implementation, the effect functional dependencies have on types in a program is "irreversible". Hence, the GHC type checker complains that variable `a` in the type annotation is unified with `Int`. In case of the GADT system, we can "undo" this effect by an extra (Eq) typing rule.

The immediate question is what kind of (type) behavior we can expect in a system which supports GADTs and type classes with extensions. The current GHC implementation treats both concepts separately and therefore function `eval` type checks but function `eval2` fails to type check. Given that types `Term a` and `Term2 a` effectively describe the same data structure, this may lead to confusion among programmers.

In the next section, we introduce a framework for extended algebraic types which unifies the concepts of type classes with existential types and GADTs. GADT and type class programs show a uniform behavior, e.g. functions `eval` and `eval2` are both accepted in our system.

4 Extended Algebraic Data Types

We start off by considering a few examples to show the benefit of extended algebraic data types (Section 4.1). Then, we describe the formal syntax of programs (Section 4.2) before we define the set of well-typed programs (Section 4.3).

4.1 Overview

Extended algebraic data types (EADTs) are introduced by declarations

```
 data T a1 ... am = forall b1,...,bn. D => K t1 ... tl | ...
```

where constraint `D` may consist of type class and type equality constraints. As we will see next, GHC-style GADTs are represented via EADTs where `D` contains type equations.

Here is a re-formulation of the `eval` example from the previous section in terms of our syntax. Additionally, we add a new case that deals with division among terms motivated by a similar example suggested on the GHC-users mailing list [Mor05].

```
data Term a = (a=Int) => Zero
            | forall b c.(a=(b,c)) => Pair (Term b) (Term c)
            | Fractional a => Div (Term a) (Term a)
eval :: Term a -> a
eval Zero = 0
eval (Pair t1 t2) = (eval t1, eval t2)
eval (Div t u) = (eval t) / (eval u)
```

Type equations `(a=Int)` and `(a=(b,c))` exactly correspond to the type equality assumption "implied" by GHC-style GADT constructors. The additional case makes use of the method `(/)` which is part of the `Fractional` type class. Hence, the program text `(eval t) / (eval u)` gives rise to `Fractional a`. This constraint is satisfied by `Fractional a` which arises out of the pattern `Div t u`. We conclude that the program type checks.

The above shows that we can specify EADTs which combine GADTs and type classes with existential types. In our next example, we require the combination of GADTs with type classes with extensions. Our goal is to refine the type of the `append` function to state that appending two lists yields a lists whose length is the sum of the two input lists.

First, we introduce a EADT where the extra parameter keeps track of the length of the list. Type constructors `Zero` and `Succ` are used to represent numbers on the level of types.

```
data Zero
data Succ n
data List a n = (n=Zero) => Nil | forall m. (n=Succ m) => Cons a m
```

Then, we define a type class and instances to define addition among our (type) number representation.

```
class Add l m n | l m -> n
instance Add Zero m m                          -- (1)
instance Add l m n => Add (Succ l) m (Succ n)
```

Note that the functional dependency [Jon00] `l m->n` states that the first two parameters uniquely determine the third parameter. Hence, the `Add` type class behaves like a function. E.g., in case we encounter `Add Zero m n` the type `n` will be improved [Jon95] to `m`. We make use of the `Add` type class to refine the type of the `append` function. Thus, we can state the desired property that the length of the output list equals the sum of the length of the two input lists.

```
append ::  Add l m n => List a l -> List a m -> List a n
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

The above program type checks in our system. E.g, consider the first clause. When pattern matching over `Nil` and `ys` we encounter the constraint `l=Zero` and find that `ys` has type `List a m`. From the annotation, we obtain `Add l m n`. In combination with the functional dependency imposed on `Add` and instance (1) both constraints imply that `m=n`. Hence, the function body `ys` satisfies the annotation. A similar observation applies to the second clause.

The above examples show that EADTs are a natural generalization of GADTs and types classes with extensions. By employing some type programming we can even mimic type properties which previously required special-purpose systems [Zen99]. Next, we take a look at the formal underpinnings of EADTs.

4.2 Expressions, Types and Constraints

The syntax of programs can be found in Figure 1. We assume that K refers to constructors of user-defined data types. As usual patterns are assumed to be linear, i.e., each variable occurs at most once. In examples we will use pattern matching notation for convenience.

For simplicity, we assume that type annotations are *closed*, i.e. in `f::C=>t` we will quantify over all variables in $fv(C, t)$ when building `f`'s type. Though, the straightforward extension to lexically scoped annotations may be necessary

Expressions	e	$::= K \mid x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ g = e\ \mathsf{in}\ e \mid$ $\mathsf{let}\ \begin{array}{l} g :: C \Rightarrow t \\ g = e \end{array}\ \mathsf{in}\ e \mid \begin{array}{l} \mathsf{case}\ e\ \mathsf{of} \\ [p_i \rightarrow e_i]_{i \in I} \end{array}$
Patterns	p	$::= x \mid K\ p...p$
Types	t	$::= a \mid t \rightarrow t \mid T\ \bar{t}$
Primitive Constraints	at	$::= t = t \mid TC\ \bar{t}$
Constraints	C	$::= at \mid C \wedge C$
Type Schemes	σ	$::= t \mid \forall \bar{a}.C \Rightarrow t$
Data Decls	$ddec$	$::= \mathtt{data}\ T\ a_1...a_m = \mathtt{forall}\ b_1, ..., b_n.D \Rightarrow K\ t_1...t_l$
Type Class Decls	$tcdec$	$::= \mathsf{class}\ TC\ \bar{a}\ \mathsf{where}\ m :: C \Rightarrow t \mid \mathsf{instance}\ C \Rightarrow TC\ \bar{t}$
CHRs	R	$::= \mathtt{rule}\ TC_1\ \overline{t_1}, ..., TC_n\ \overline{t_n} \iff C \mid$ $\mathtt{rule}\ TC_1\ \overline{t_1}, ..., TC_n\ \overline{t_n} \implies C$

Fig. 1. Syntax of Programs

to sufficiently annotate programs [SW05]. We also omit un-annotated recursive function definitions, another straightforward extension.

Our type language is standard. We assume that $T\ \bar{t}$ refer to user-definable data types. We use common Haskell notation for writing function, pair, list types etc. We assume that constructor and destructor functions are recorded in some initial environment Γ_{init}, e.g. $(\cdot, \cdot) : \forall a, b.a \rightarrow b \rightarrow (a, b)$, $\mathit{fst} : \forall a, b.(a, b) \rightarrow a$, $\mathit{snd} : \forall a, b.(a, b) \rightarrow b \in \Gamma_{init}$ etc.

A *primitive* constraint (a.k.a. *atom*) is either an equation $t = t'$ (a.k.a. type equality) or an n-ary *type class* constraint $TC\ \bar{t}$. We assume a special (always satisfiable) constraint $True$ representing the empty conjunction of constraints, and a special (never satisfiable) constraint $False$. Often, we treat conjunctions of constraints as sets and abbreviate Boolean conjunction $\wedge$ by ",". We generally use symbols C and D to refer to sets of constraints.

Type schemes have an additional constraint component which allows us to restrict the set of type instances. We often refer to a type scheme as a type for short. Note that we consider $\forall \bar{a}.t$ as a short-hand for $\forall \bar{a}.True \Rightarrow t$. The presence of equations makes our system slightly more general compared to standard Hindley/Milner. E.g., types $\forall a, b.a = b \Rightarrow a \rightarrow b$ and $\forall a.a \rightarrow a$ are equivalent. Equations will become interesting once we allow them to appear in type assumptions of constructors.

We assume that data type declarations

```
data T a1 ... am = forall b1,...,bn. D => K t1 ... tl | ...
```

are preprocessed and the types of constructors $K : \forall \bar{a}, \bar{b}.D \Rightarrow t_1 \rightarrow ... \rightarrow t_l \rightarrow T\ \bar{a}$ are recorded in the initial environment Γ_{init}. We assume that $\bar{a} \cap \bar{b} = \emptyset$. Note that $\bar{a}$ and $\bar{b}$ can be empty.

Similarly, for each class declaration $\mathsf{class}\ TC\ \bar{a}\ \mathsf{where}\ m :: C \Rightarrow t$ we find $m : \forall \mathit{fv}(C, t, \bar{a}).(TC\ \bar{a}, C) \Rightarrow t \in \Gamma_{init}$. Super-classes do not impose any challenges for typing and translation programs. Hence, we ignore them for brevity.

We also ignore the bodies of instance declarations. Details of how to type and translate instance bodies can be found elsewhere [SWP05].

In our source syntax, there is no explicit support for type improvement mechanisms such as functional dependencies [Jon00] or associated types [CKJ05]. Improvement conditions are encoded via CHRs as we will see shortly. Though, we may make use of the functional dependency notation in example programs.

Following our earlier work [SS05,DJSS04] we employ Constraint Handling Rules (CHRs) [Frü95] as the internal formalism to describe the valid type class relations. E.g., each declaration instance $C' \Rightarrow TC\ \bar{t}$ is described by the *simplification* `rule` $TC\ \bar{t} \Longleftrightarrow C'$ whereas type improvement conditions are specified by the second kind of CHRs which are referred to as *propagation* rules.

Here are the CHRs describing the instance and functional dependency relations of the `Add` type class from the previous section.

```
rule Add Zero m m <==> True                (A1)
rule Add (Succ l) m (Succ n) <==> Add l m n (A2)
rule Add l m n1, Add l m n2 ==> n1=n2       (A3)
rule Add Zero m n ==> m=n                   (A4)
rule Add (Succ l) m n' ==> n'=Succ n        (A5)
```

How to systematically derive such CHRs from source programs can be found here [SS05,DJSS04]. For a given program we assume a fixed set P of CHRs to which we refer to as the *program logic*.

For the description of the set of well-typed expressions in the next section we apply the logic interpretation of CHRs which is as follows: Symbol `==>` denotes Boolean implication and `<==>` denotes Boolean equivalence. Variables in the rule *head* (left-hand side) are universally quantified whereas all remaining variables on the right-hand side are existentially quantified. E.g., the CHR (A5) is interpreted as the formula $\forall l, m, n'.(Add\ (Succ\ l)\ m\ n' \supset \exists n.n' = Succ\ n)$.

4.3 Type System

To describe well-typing of expressions we make use of judgments of the form $C, \Gamma \vdash e : t$ where C is a constraint, Γ refers to the set of lambda-bound variables, predefined and user-defined functions, e is an expression and t is a type. Note that we leave the program logic P implicit. None of the typing rules affect P, hence, we can assume that P is fixed for a given expression. We say a judgment is *valid* iff there is a derivation w.r.t. the rules found in Figure 2. Each valid judgment implies that the expression is well-typed.

Let us take a look at the typing rules in detail. In rule (Var-∀E), we build a type instance if we can verify that the instantiated constraint is logically contained by the given constraint under the given program logic. This is formally expressed by the side condition $P \models C \supset [\overline{t/a}]D$.

In rule (Eq) we can change the type of expressions.[6] Note that the set C of constraints may not necessarily be the same in all parts of the program (see upcoming rule (Pat)). Therefore, this rule plays a crucial role. Recall function append from Section 4.1 where in case of the second clause we find $Add\ l\ m\ n, l =$

[6] Some formulations allow us to change the type of (sub)patterns [SP05]. 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 and our Chameleon implementation.

$$\text{(Var-}\forall\text{E)}\quad \frac{\begin{array}{c}(x : \forall \bar{a}.D \Rightarrow t') \in \Gamma \\ P \models C \supset [\overline{t/a}]D\end{array}}{C, \Gamma \vdash x : [\overline{t/a}]t'} \qquad \text{(Eq)}\quad \frac{\begin{array}{c}C, \Gamma \vdash e : t_1 \\ P \models C \supset t_1 = t_2\end{array}}{C, \Gamma \vdash e : t_2}$$

$$\text{(Abs)}\quad \frac{C, \Gamma \cup \{x : t_1\} \vdash e : t_2}{C, \Gamma \vdash \lambda x.e : t_1 \rightarrow t_2} \qquad \text{(App)}\quad \frac{\begin{array}{c}C, \Gamma \vdash e_1 : t_1 \rightarrow t_2 \\ C, \Gamma \vdash e_2 : t_1\end{array}}{C, \Gamma \vdash e_1 e_2 : t_2}$$

$$\text{(Let)}\quad \frac{\begin{array}{c}C_1, \Gamma \vdash e_1 : t_1 \quad \bar{a} = fv(C_1, t_1) - fv(C_2, \Gamma) \\ C_2, \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\} \vdash e_2 : t_2\end{array}}{C_2, \Gamma \vdash \mathsf{let}\ g = e_1\ \mathsf{in}\ e_2 : t_2}$$

$$\text{(LetA)}\quad \frac{\begin{array}{c}\bar{a} = fv(C_1, t_1) \quad C_2 \wedge C_1, \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\} \vdash e_1 : t_1 \\ C_2, \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\} \vdash e_2 : t_2\end{array}}{C_2, \Gamma \vdash \mathsf{let}\ \begin{array}{c}g :: C_1 \Rightarrow t_1 \\ g = e_1\end{array}\ \mathsf{in}\ e_2 : t_2}$$

$$\text{(Case)}\quad \frac{\begin{array}{c}C, \Gamma \vdash e : t_1 \\ C, \Gamma \vdash p_i \rightarrow e_i : t_1 \rightarrow t_2 \quad \text{for } i \in I\end{array}}{C, \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} : t_2} \qquad \text{(Pat)}\quad \frac{\begin{array}{c}p : t_1 \vdash \forall \bar{b}.(D \mid \Gamma_p) \\ fv(C, \Gamma, t_2) \cap \bar{b} = \emptyset \\ C \wedge D, \Gamma \cup \Gamma_p \vdash e : t_2\end{array}}{C, \Gamma \vdash p \rightarrow e : t_1 \rightarrow t_2}$$

$$\text{(Pat-Var)}\quad x : t \vdash (\mathit{True} \mid \{x : t\})$$

$$\text{(Pat-K)}\quad \frac{\begin{array}{c}K : \forall \bar{a}, \bar{b}.D \Rightarrow t'_1 \rightarrow \ldots \rightarrow t'_l \rightarrow T\ \bar{a} \quad \bar{b} \cap \bar{a} = \emptyset \\ p_k : [\overline{t/a}]t'_k \vdash \forall \bar{b}'_k.(D'_k \mid \Gamma_{p_k}) \quad \text{for } k = 1, \ldots, l\end{array}}{K\ p_1 \ldots p_l : T\ \bar{t} \vdash \forall \bar{b}'_1, \ldots, \bar{b}'_l, \bar{b}.(D'_1 \wedge \ldots D'_l \wedge [\overline{t/a}]D \mid \Gamma_{p_1} \cup \ldots \Gamma_{p_l})}$$

Fig. 2. Typing Rules

Zero in the constraint component and variable `ys` has type *List a m*. Here, P consists of rules (A1-5) from the previous section. We find that $P \models \mathit{Add}\, l\, m\, n, l = \mathit{Zero} \supset m = n$. Hence, we can change the type of `ys` to *List a n*. Thus, we can verify that `append`'s annotation is correct.

Rules (Abs) and (App) are straightforward. In rule (Let), we include the rule for quantifier introduction. Note that we could be more efficient by only quantifying over the "affected" constraints. Further note that satisfiability of the "final" constraint C_2 does not imply that C_1 is satisfiable. E.g., consider the situation where `g` is not used. Hence, our formulation is more "lazy" compared to other schemes. We refer to [OSW99,Sul00] for a detailed discussion of different formulations of quantifier introduction.

Rule (LetA) deals with a closed annotation. Variables in $C_1 \Rightarrow t_1$ are assumed to be universally quantified. Note that via this rule we can support polymorphic recursive functions (for simplicity, we omit the rule to deal with monomorphic recursive functions).

Rules (Case) and (Pat) deal with pattern matching. Rule (Pat) is in particular interesting. For convenience, we consider $p \to e$ as a (special purpose) expression only appearing in intermediate steps. We make use of an auxiliary judgment $p : t \vdash \forall \bar{b}.(D \mid \Gamma_p)$ to establish a relation among pattern p of type t and the binding Γ_p of variables in p. Constraint D arises from constructor uses in p. Variables $\bar{b}$ are not allowed to escape which is captured by the side condition $fv(C, \Gamma, t_2) \cap \bar{b} = \emptyset$. Note that we type the body of the pattern clause under the "temporary" type assumption D and environment Γ_p arising out of p. Consider again function `append` from Section 4.1 where we temporarily make use of $l = Zero$ in the first clause and $l = Succ\ l'$ in the second clause.

The rules for the auxiliary judgment are as follows. In rule (Pat-K) we assume that variables $\bar{a}$ and $\bar{b}$ are fresh. Hence, w.l.o.g. there are no name clashes between variables $\bar{b'_1}$,...,$\bar{b'_l}$. Rule (Pat-Var) is standard.

The description of the semantic meaning of programs and its type soundness proof had to be sacrificed due to space restrictions. Details can be found in an accompanying technical report [SWP06]. Here, we only consider type checking which we discuss next.

5 Type Checking

The problem we face is as follows. Given a constraint C, an environment Γ, a type t and an expression e where all let-defined functions are type annotated, we want to verify that $C, \Gamma \vdash e : t$ holds. This is known as type checking as opposed to inference where we compute C and t given Γ and e.

It is folklore knowledge that type checking can be turned into a entailment test among constraints. The path we choose is to translate in an intermediate step the type checking problem to a set of implication constraints [SP05]. A program type checks if the implication constraint holds. We then show how to reduce the decision problem for checking implication constraints to standard CHR solving. Thus, we obtain a computationally tractable type checking method.

5.1 Type Checking via Implication Constraints

The syntax of implication constraints is as follows.

$$\begin{array}{ll} \text{Constraints} & C ::= t = t \mid TC\ \bar{t} \mid C \wedge C \\ \text{ImpConstraints} & F ::= C \mid \forall \bar{b}.(C \supset \exists \bar{a}.F) \mid F \wedge F \end{array}$$

The actual translation to implication constraints follows the description of [SP05]. We employ a deduction system in style of algorithm $\mathcal{W}$ where we use judgments of the form $\Gamma, e \vdash_W (F \mid t)$ to denote that under input environment Γ and expression e we obtain the output implication constraint F and type t. The rules can be found in Figure 3. Recall that let-defined functions are type annotated.

We briefly review the individual rules. We write $\equiv$ to denote syntactic equality. In rules (Var) and (Pat-K) we assume that the bound variables $\bar{a}$ and $\bar{b}$ are fresh. We assume that $dom(\phi)$ computes the variables in the domain of ϕ. In rule (Pat) we make use of the implication constraint $\forall \bar{b}.(D \supset \bar{\exists}_{fv(\Gamma, \bar{b}, t_e)}.F_e)$ which states that under the temporary assumptions D arising out of the pattern p we can satisfy the implication constraint F_e arising out of e. The $\forall$ quantifier ensures that no existential variables $\bar{b}$ escape. Formula $\bar{\exists}_{fv(\Gamma, \bar{b}, t_e)}.F_e$ is a short-hand

$$\text{(LetA)}\ \frac{\begin{array}{c}\bar{a} = fv(C_1, t_1) \quad b_1, b_1' \text{ fresh} \\ \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\}, e_1 \vdash_W (F_1 \mid t_1') \quad \Gamma \cup \{g : \forall \bar{a}.C_1 \Rightarrow t_1\}, e_2 \vdash_W (F_2 \mid t_2) \\ F \equiv F_2 \wedge \forall \bar{a}.((C_1 \wedge t_1 = b_1 \wedge b_1 = b_1') \supset \bar{\exists}_{fv(\Gamma, b_1')}.F_1 \wedge b_1' = t_1')\end{array}}{\Gamma, \mathsf{let}\ \begin{array}{c} g :: C_1 \Rightarrow t_1 \\ g = e_1 \end{array}\ \mathsf{in}\ e_2 \vdash_W (F \mid t_2)}$$

$$\text{(App)}\ \frac{\begin{array}{c}\Gamma, e_1 \vdash_W (F_1 \mid t_1) \\ tenv, e_2 \vdash_W (F_2 \mid t_2) \\ t \text{ fresh} \\ F \equiv F_1 \wedge F_2 \wedge t_1 = t_2 \rightarrow t\end{array}}{\Gamma, e_1\ e_2 \vdash_W (F \mid t)} \qquad \text{(Var)}\ \frac{(x : \forall \bar{a}.C \Rightarrow t) \in \Gamma}{\Gamma, x \vdash_W (C \mid t)} \qquad \text{(Abs)}\ \frac{\begin{array}{c} a \text{ fresh} \\ \Gamma \cup \{x : a\}, e \vdash_W (F \mid t)\end{array}}{\Gamma, \lambda x.e \vdash_W (F \mid a \rightarrow t)}$$

$$\text{(Case)}\ \frac{\begin{array}{c}\Gamma, p_i \rightarrow e_i \vdash_W (F_i \mid t_i') \quad \text{for } i \in I \quad \Gamma, e \vdash_W (F_e \mid t_e) \quad t_1, t_2 \text{ fresh} \\ F \equiv F_e \wedge t_1 = t_e \rightarrow t_2 \wedge \bigwedge_{i \in I}(F_i \wedge t_1 = t_i')\end{array}}{\Gamma, \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} \vdash_W (F \mid t_2)}$$

$$\text{(Pat)}\ \frac{\begin{array}{c} p \vdash \forall \bar{b}.(D \mid \Gamma_p \mid t_1) \quad \Gamma \cup \Gamma_p, e \vdash_W (F_e \mid t_e) \quad t \text{ fresh} \\ F \equiv \forall \bar{b}.(D \supset \bar{\exists}_{fv(\Gamma, \bar{b}, t_e)}.F_e) \wedge t = t_1 \rightarrow t_e\end{array}}{\Gamma, p \rightarrow e \vdash_W (F \mid t)}$$

$$\text{(Pat-Var)}\ \frac{t \text{ fresh}}{x \vdash (True \mid \{x : t\} \mid t)}$$

$$\text{(Pat-K)}\ \frac{\begin{array}{c} K : \forall \bar{a}, \bar{b}.D \Rightarrow t_1 \rightarrow ... \rightarrow t_l \rightarrow T\ \bar{a} \quad \bar{b} \cap \bar{a} = \emptyset \\ p_k \vdash \forall \bar{b_k'}.(D_k' \mid \Gamma_{p_k} \mid t_{p_k}) \quad \phi \text{ m.g.u. of } t_{p_k} = t_k \quad \text{for } k = 1, ..., l \quad dom(\phi) \cap \bar{b} = \emptyset\end{array}}{K\ p_1 ... p_l \vdash \forall \bar{b_1'}, ..., \bar{b_l'}, \bar{b}.(\phi(D_1') \wedge ... \phi(D_l') \wedge \phi(D) \mid \phi(\Gamma_{p_1}) \cup ... \cup \phi(\Gamma_{p_l}) \mid T\ \phi(\bar{a}))}$$

Fig. 3. Translation to Implication Constraints

for $\exists \bar{a}.F_e$ where $\bar{a} = fv(F_e) - fv(\Gamma, \bar{b}, t_e)$. That is, we existentially quantify over all variables which are strictly local in F_e. In the special case of (existential) algebraic data types the constraint D equals $True$ and F_e is a constraint. In rule (LetA), we generate a formula to check that the type of the body e_1 subsumes the annotated type of function f. In logical terms, the subsumption condition is expressed by the formula $\forall \bar{a}(C_1 \supset \bar{\exists}_{fv(\Gamma, t_1)}.F_1 \wedge t_1 = t_1')$. However, this form is not suitable for the upcoming checking procedure because we would need to guess the possible assignments under which $t_1 = t_1'$ holds. Therefore, we push the constraint $t_1 = t_1'$ into the assumption part (left-hand side of $\supset$). For technical reasons, we need to ensure that type schemes $\forall \bar{a}.C \Rightarrow t$ are in the (equivalent) normalized form $\forall \bar{a}, b.C \wedge b = t \Rightarrow b$ where b is fresh. Details are in [SS05]. Note that there is no (Let) rule because we assume that all let-defined functions must be annotated with a type.

The type checking problem says that for a given constraint C, environment Γ, expression e and type t we need to verify that $C, \Gamma \vdash e : t$ holds. We can reduce this problem to testing whether constraint C implies the implication constraint generated from Γ and e

Theorem 1 (Type Checking via Implication Checking). *Let P be a program logic. Let Γ be an environment, e an expression, C a constraint and t a type. Let $\Gamma, e \vdash_W (F \mid t')$ such that $P \models (C, t = t') \supset \bar{\exists}_{fv(\Gamma,t')}.F$ holds where w.l.o.g. t and t' are variables. Then, $C, \Gamma \vdash e : t$.*

Note that $(C, t = t') \supset \bar{\exists}_{fv(\Gamma,t')}.F$ is itself again a implication constraint (we leave the outermost universal quantifier implicit). Hence, the type checking problem boils down to testing whether a implication constraint holds w.r.t. the program logic P. We neglect here the "opposite" task of finding C which corresponds to type inference.

5.2 Checking Implication Constraints

First, we review some background material on CHR solving. The operational reading of constraint rules (CHRs) is simple. In case of propagation rules we add the right-hand side if we find a matching copy of the lhs in the constraint store. In case of simplifications rules we remove the matching copy and replace it by the right-hand side. The formal definition is as follows [Frü95].

Definition 1 (CHR Semantics). *Let P be a set of CHRs.*

Propagation: *Let $(R)\ \ c_1, ..., c_n \Longrightarrow d_1, ..., d_m \in P$ and C be a constraint. Let ϕ be the m.g.u. of all equations in C. Let $c'_1, ..., c'_n \in C$ such that there exists a substitution θ on variables in rule (R) such that $\theta(c_i) = \phi(c'_i)$ for $i = 1...n$, that is user-defined constraints c'_1,...,c'_n* match *the left-hand side of rule (R). Then, $C \rightarrowtail_R C, \theta(d_1), ..., \theta(d_m)$.*

Simplification: *Let $(R)\ \ c_1, ..., c_n \Longleftrightarrow d_1, ..., d_m \in P$ and C be a constraint. Let ϕ be the m.g.u. of all equations in C. Let $c'_1, ..., c'_n \in C$ such that there exists a substitution θ on variables in rule (R) such that $\theta(c_i) = \phi(c'_i)$ for $i = 1, ..., n.$, Then, $C \rightarrowtail_R C - \{c'_1, ..., c'_n\}, \theta(d_1), ..., \theta(d_m)$.*

Often, we perform some equivalence transformations (e.g. normalize equations by building the m.g.u. etc) which are either implicit or explicitly denoted by $\longleftrightarrow$. A *derivation*, denoted $C \rightarrowtail_P^* C'$ is a sequence of derivation steps using rules in P such that no further derivation step is applicable to C'. CHRs are applied exhaustively, being careful not to apply propagation rules twice on the same constraints (to avoid infinite propagation). For more details on avoiding re-propagation see e.g. [Abd97]. We say a set P of CHRs is *terminating* if for each C there exists C' such that $C \rightarrowtail_P^* C'$.

We repeat the CHR soundness result [Frü95] which states that CHR rule applications perform equivalence transformations.

Lemma 1 (CHR Soundness [Frü95]). *Let $C \rightarrowtail_P^* C'$. Then $P \models C \leftrightarrow \bar{\exists}_{fv(C)}.C'$.*

Recall the CHRs for the `Add` type class.

```
rule Add Zero m m <==>                       (A1)
rule Add (Succ l) m (Succ n) <==> Add l m n (A2)
rule Add l m n1, Add l m n2 ==> n1=n2       (A3)
rule Add Zero m n ==> m=n                   (A4)
rule Add (Succ l) m n' ==> n'=Succ n        (A5)
```

We have that

$$\begin{array}{ll} & \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ m\ n, \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ m\ n', m = \mathit{Zero} \\ \rightarrowtail_{A3} & \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ m\ n, m = \mathit{Zero}, n = n' \\ \longleftrightarrow & \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ \mathit{Zero}\ n, m = \mathit{Zero}, n = n' \\ \rightarrowtail_{A5} & \mathit{Add}\ (\mathit{Succ}\ \mathit{Zero})\ \mathit{Zero}\ (\mathit{Succ}\ n''), n = \mathit{Succ}\ n'', m = \mathit{Zero}, n = n' \\ \rightarrowtail_{A1} & \mathit{Add}\ \mathit{Zero}\ \mathit{Zero}\ n'', n = \mathit{Succ}\ n'', m = \mathit{Zero}, n = n' \\ \rightarrowtail_{A4} & \mathit{Add}\ \mathit{Zero}\ \mathit{Zero}\ \mathit{Zero}, n = \mathit{Succ}\ \mathit{Zero}, n'' = \mathit{Zero}, m = \mathit{Zero}, n = n' \\ \rightarrowtail_{A2} & n = \mathit{Succ}\ \mathit{Zero}, n'' = \mathit{Zero}, m = \mathit{Zero}, n = n' \end{array}$$

We show how to lift the (primitive) constraint solver $\rightarrowtail_P^*$ to the domain of implication constraints. We write $F \gg_P^* C$ to denote that checking of implication constraint F yields (after some n number of steps) solution C. Our idea is to turn the implication checking problem into an equivalence checking problem by making use of the fact that $C_1 \supset C_2$ iff $C_1 \leftrightarrow C_1, C_2$. Then, we can use the primitive constraint solver and execute $C_1 \rightarrowtail_P^* D_1$ and $C_1, C_2 \rightarrowtail_P^* D_2$. Next, we check for logical equivalence by testing whether D_1 and D_2 share the same m.g.u. and their user-defined constraints are renamings of each other. If equivalence holds, then $C_1 \supset C_2$ is solved and $True$ is the solution. The exact checking details are as follows.

Definition 2 (CHR-Based Implication Checker). *Let P be a set of CHRs and F an implication constraint.*

Primitive: *We define $F \gg_P C'$ where $C \rightarrowtail_P^* C'$ if $F \equiv \exists \bar{a}.C$.*

General: *Otherwise $F \equiv C_0, (\forall \bar{a}.D \supset \exists \bar{b}.F_1), F_2$ where C_0 is a conjunction of primitive constraints, D is a set of assumptions and F_1 and F_2 are implication constraints.*

We compute (1) $C_0, D \rightarrowtail_P^* D'$ and (2) $C_0, D, F_1 \gg_P^* C'$ for some D' and C'. We distinguish among the following cases.

Solved: *We define $F \gg_P C_0, F_2$ if $\models (\bar{\exists}_V.D') \leftrightarrow (\bar{\exists}_V.C')$ where $V = fv(C_0, D, \bar{a})$.*

Failure: *We define $F \gg_P False$ in all other cases.*

We assume that $\gg_P^*$ denotes the exhaustive application of CHR implication solving steps.

In the **Primitive** step we apply standard CHR solving. No surprises here. In the **General** step, we split the constraint store into C_0 containing sets of primitive constraints, a single implication constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ and F_2 containing the remaining implication constraints. Strictly speaking, F_2 itself could be a set of primitive constraints. Silently, we assume that all sets of primitive constraints are collected in C_0. Also note that we inductively solve nested implication constraints, see (2).

The **Solved** step applies if the equivalence check succeeds. Hence, the constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ is removed from the store. Note that w.l.o.g. we assume that $\bar{b} = fv(F_2) - fv(\bar{a}, C_0)$. Any variable not bound by a universal quantifier is (implicitly) existentially bound. The CHR Soundness result immediately yields that this step is sound, hence, the overall procedure is correct (see upcoming Lemma 2).

For example, the `append` function from Section 4.1 gives rise to the following (simplified) implication constraint.

$$\begin{array}{l} \forall a, l, m, n.\ (Add\ l\ m\ n, t = List\ a\ l \to List\ a\ m \to List\ a\ n) \supset \\ \qquad \left(\begin{array}{l} (l = Zero \supset t = List\ a\ l \to List\ a\ m \to List\ a\ m), \quad (1) \\ \exists l'.(l = Succ\ l' \supset (Add\ l'\ m\ n', t = List\ a\ l \to List\ a\ m \to List\ a\ (Succ\ n')) \end{array} \right) \end{array}$$

Based on our implication checking procedure, we can verify that the above formula F holds, i.e. $F \gg_P^* True$. E.g., in an intermediate step, we find that

$$\begin{array}{rl} & Add\ l\ m\ n, t = List\ a\ l \to List\ a\ m \to List\ a\ n, l = Zero \\ \rightarrowtail^* & t = List\ a\ Zero \to List\ a\ m \to List\ a\ n, l = Zero, m = n \end{array}$$

and thus we can verify the inner implication (1). A similar reasoning applies to the remaining part. Hence, from Theorem 1 and Lemma 2 we can conclude that the `append` function type checks.

Lemma 2 (Implication Checking Soundness). *Let P be a set of CHRs, F be an implication constraint and C be a set of primitive constraints such that $F \gg_P^* C$. Then, $P \models C \leftrightarrow F$.*

Our implication checking procedure is terminating if the underlying primitive constraint solver is terminating. Thus, we obtain decidable type checking. There are plenty of criteria (imposed on source EADT programs) such as the Haskell type class [Pey03] and Jones's functional dependency restrictions [Jon00] which ensure termination of the resulting CHRs. We refer to [DJSS04] for more details.

Though, termination does not ensure that type checking is complete. The problem is that type constraints arising out of the program text may be "ambiguous" which requires us to guess types. The solution is to reject such programs or demand further user assistance in form of type annotations.

Yet another source of incompleteness is our primitive constraint solver which we use for equivalence checking. We will elaborate on such issues and how to obtain complete type checking in an extended version of this paper.

6 Discussion and Related Work

Peyton Jones, Washburn and Weirich [JWW04] have added GADTs to GHC. However, GADTs and type classes do not seem to interact well in GHC. E.g., the examples from Section 4.1 are not typable.

Sheard [She05] and Chen/Xi [CX05] have extended GADTs to allow users to specify their own program properties which previously required external proof systems such as [PS99,BBC+96]. An interesting question is to what extent "extended GADTs" can be expressed in terms of EADTs. E.g., consider the `append` function from Section 4.1 which appears in very similar form in [CX05]. One difference is that the works in [She05,CX05] use type functions to specify type

```
data Exp = ...                        -- resource automaton --
-- resource EADT                      data S0 -- states
data Cmd p q =                        data S1
  forall r. Seq (Cmd p r) (Cmd r q)   data Open -- alphabet
   | ITE Exp (Cmd p q) (Cmd p q)      data Close
   | (p=q) => While Exp (Cmd p q)     data Write
  -- while state is invariant         -- valid transition
   | Delta p Open q => OpenF          rule Delta S0 Open x <==> x=S1
   | Delta p Close q => CloseF        rule Delta S1 Close x <==> x=S0
   | Delta p Write q => WriteF        rule Delta S1 Write x <==> x=S1

-- improvement
rule Delta a b c, Delta a b d ==> c=d
rule Delta x Open y ==> x=S0, y=S1
rule Delta x Close y ==> x=S1, y=S0 -- (Imp)
rule Delta S1 Open x ==> False
-- failure
rule Delta S0 Close x ==> False
rule Delta S0 Write x ==> False -- (Fail)
```

Fig. 4. Resource EADT

properties whereas we use CHRs. We claim that CHRs allow us to specify more complex type properties than specifiable via type functions as shown by the EADT in Figure 4.

We introduce a EADT to represent a while language which satisfies a resource usage analysis specified in terms of a DFA. The DFA relations are specified via CHRs. Type parameters `p` and `q` represent the input and output state, before and after execution of the command. Notice how we constrain states `p` and `q` by the constraint `Delta` which represents the DFA state transition function `delta` (see the last three cases).

The CHRs encode a specific automaton for a resource usage policy where we may open a file, write an arbitrary number of times to the file and close the file. The improvement and failure rules are particularly interesting here. They allow us to aggressively enforce the resource automaton. E.g., after closing a file we must return to the start state `S0` (see (Imp)). We are not allowed to write if we are in the start state (see (Fail)) etc. We cannot see how to model this behavior via type functions in the systems described in [She05,CX05].

Simonet and Pottier [SP05] have also employed implication constraints for type inference in an extension of Hindley/Milner with GADTs but did not give any checking method in the general form as stated here. In some subsequent work, Pottier and Régis-Gianas [PRG06] showed how to perform complete type checking for GADTs without implication constraints on the expense of demanding an excessive amount of type annotations. These annotations are inferred by an elaboration phase. E.g., they show how to successfully elaborate the following program based on a heuristic algorithm Ibis.

```
data T = (a=Int) => I
double::T a->[a]->[a]
double t l = map (\x-> case t of I -> x+x) l
```

We also successfully accept the above program based on our type checking method. We consider this as an indication that implication constraints are an interesting approach to describe the elaboration of programs to achieve complete type checking.

7 Conclusion

We have formalized the concept of extended algebraic data types which unifies type classes with extensions and GADTs. We could provide evidence that the extension is useful and extends the expressiveness of languages such as Haskell significantly. We have introduced a novel method to support type checking for all instances of our framework.

For practical reasons, we also want to look into type inference to relieve the user from the burden of providing annotations. In this context, it is also important to consider how to give feedback in case of type errors. We have already started work in this direction which we plan to report in the near future.

Acknowledgments

We thank the reviewers for their constructive comments.

References

[Abd97] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.

[BBC$^+$96] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, H. Herbelin, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 6.1.* INRIA-Rocquencourt-CNRS-ENS Lyon, December 1996.

[CH03] J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[CKJ05] M. Chakravarty, G. Keller, and S. Peyton Jones. Associated types synonyms. In *Proc. of ICFP'05*, pages 241–253. ACM Press, 2005.

[CX05] C. Chen and H. Xi. Combining programming with theorem proving. In *Proc. of ICFP'05*, pages 66–77. ACM Press, 2005.

[DJSS04] G. J. Duck, S. Peyton Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

[Frü95] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[GHC] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[JJM97] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[Jon92] M. P. Jones. *Qualified Types: Theory and Practice.* D.phil. thesis, Oxford University, September 1992.

[Jon95] M. P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture.* ACM Press, 1995.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[JWW04] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types, 2004. Submitted to POPL'05.

[Läu96] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, 1996.

[LMM87] J. Lassez, M. Maher, and K. Marriott. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan Kauffman, 1987.

[LO94] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.

[Mor05] J. Garrett Morris. GADT question. http://www.haskell.org//pipermail/glasgow-haskell-users/2005-October/009076.html, 2005.

[Nil05] H. Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proc. of ICFP'05*, pages 54–65. ACM Press, 2005.

[OSW99] M. Odersky, M. Sulzmann, and M Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[Pey03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PG04] F. Pottier and N. Gauthier. Polymorphic typed defunctionalization. In *Proc. of POPL'04*, pages 89–98. ACM Press, January 2004.

[PRG06] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Proc. of POPL'06*, pages 232–244. ACM Press, 2006.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *CADE*, volume 1632 of *LNCS*, pages 202–206. Springer-Verlag, 1999.

[She05] T. Sheard. Putting curry-howard to work. In *Proc. of Haskell'05*, pages 74–85. ACM Press, 2005.

[Sho67] J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[SP04] T. Sheard and E. Pasalic. Meta-programming with built-in type equality. In *Fourth International Workshop on Logical Frameworks and Meta-Languages*, 2004.

[SP05] V. Simonet and F. Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, January 2005.

[SS05] P.J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems*, 2005. To appear.

[Sul00] M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

[SW] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/˜ sulzmann/chameleon.

[SW05] M. Sulzmann and J. Wazny. Lexically scoped type annotations. http://www.comp.nus.edu.sg/˜ sulzmann, 2005.

[SWP05] M. Sulzmann, J. Wazny, and P.J.Stuckey. Co-induction and type improvement in type class proofs. http://www.comp.nus.edu.sg/˜ sulzmann, 2005.

[SWP06] M. Sulzmann, J. Wazny, and P.J.Stuckey. A framework for extended algebraic data types. Technical report, The National University of Singapore, 2006.

[WB89] 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.

[XCC03] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. of POPL'03*, pages 224–235. ACM Press, 2003.

[Zen99] C. Zenger. *Indizierte Typen*. PhD thesis, Universität Karlsruhe, 1999.