# Complete and Decidable Type Inference for GADTs

Tom Schrijvers *
Katholieke Universiteit Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Simon Peyton Jones
Microsoft Research Cambridge, UK
simonpj@microsoft.com

Martin Sulzmann
Intaris Software GmbH, Germany
martin.sulzmann@gmail.com

Dimitrios Vytiniotis
Microsoft Research Cambridge, UK
dimitris@microsoft.com

## Abstract

GADTs have proven to be an invaluable language extension, for ensuring data invariants and program correctness among others. Unfortunately, they pose a tough problem for type inference: we lose the principal-type property, which is necessary for modular type inference.

We present a novel and simplified type inference approach for local type assumptions from GADT pattern matches. Our approach is complete and decidable, while more liberal than previous such approaches.

***Categories and Subject Descriptors*** D.3.2 [*Programming Languages*]: Language Classifications—Functional Languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type Structure

***General Terms*** Algorithms, Languages

***Keywords*** Haskell, type inference, GADTs

## 1. Introduction

Generalized Algebraic Data Types (GADTs) pose a particularly tough problem for type inference: we lose the principal-type property, which is necessary for modular type inference (CH03), and it is even undecidable whether or not a term *has* a principal type (Section 4.3).

A variety of papers have tackled this problem, by a combination of user-supplied type annotations and/or constraint-based inference (PVWW06; PRG06; SP07; SSS08). Unfortunately, none of these approaches is satisfying, even to their originators, for a variety of reasons (Section 9). Simonet and Pottier give an excellent executive summary in the closing sentences of their recent paper (SP07):

> *We believe that one should, instead, strive to produce simpler constraints, whose satisfiability can be efficiently determined by a (correct and complete) solver. Inspired by Peyton Jones et al.'s wobbly types (PVWW06), recent work by Pottier and Régis-Gianas (PRG06) proposes one way of doing so, by relying on explicit, user-provided type annotations and on an ad hoc local shape inference phase. It would be interesting to know whether it is possible to do better, that is, not to rely on an ad hoc preprocessing phase.*

This is the challenge we meet in this paper. In particular, our contributions are:

- We present **OutsideIn**, a new inference algorithm for GADT programs (Sections 4 and 5). It deals with the tricky problem of solving so-called implication constraints (Section 4.2) by allowing information to propagate from outside a GADT pattern match to the inside, but not vice versa.
- The declarative specification of the type system is given in Section 6. While still not entirely satisfactory, is arguably significantly simpler than earlier attempts. It is also rather expressive: it types strictly more programs than GHC's existing algorithm (PVWW06), and very nearly all those typed by (PRG06) — Section 9 elaborates.
- Our type system has the crucial principal-types property; and any program accepted by our type system is also typed by the simpler "natural" type system for GADTs, which lacks principal types. Furthermore, all programs typeable in our approach *do have principal types* in the natural type system for GADTs and hence our approach is not "adhoc" (Section 7.1). That is not the case for either (PVWW06) or (PRG06).
- The type inference algorithm is sound and complete with respect to the specification; and it is decidable (Section 7.2).
- The type inference algorithm is easy to implement (Section 8). The inference engine gathers and solves constraints, but that is something Haskell compilers already do for type classes; all we do here is add some new forms of constraints. Nevertheless inference remains efficient, because almost all equalities can be solved with on-the-fly unification in the conventional Hindley-Milner way, with constraints gathered only when necessary. Our prototype implementation is available for download. *Note: we have not yet implemented it in GHC, but expect to have done so before ICFP.*

Our approach builds directly on the work of others. We urge the reader to consult Section 9 for a summary of these foundations.

[*] Post-doctoral researcher of the Fund for Scientific Research - Flanders.

*ICFP'09,* August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08...$5.00

## 2. The challenge we address

Generalized Algebraic Data Types (GADTs) have proved extremely popular with programmers, but they present the type inference engine with tricky choices. Consider the following GADT program:

```
data T :: *->* where
  T1 :: Int -> T Bool
  T2 :: [a] -> T a

f1 (T1 n) = n>0
```

What type should be inferred for function `f1`? Alas there are two possible most-general types, neither of which is an instance of the other:

$$\begin{array}{l} \texttt{f1} :: \forall a.\, \texttt{T}\ a \rightarrow \texttt{Bool} \\ \texttt{f1} :: \forall a.\, \texttt{T}\ a \rightarrow a \end{array}$$

The loss of principal types is both well-known and unavoidable (CH03). Since `f1` has no principal type (one that is more general than all others) the right thing must be to reject the program, and ask the programmer to say which type is required by means of an explicit type signature, like this, for example:

```
f1 :: T a -> a
f1 (T1 n) = n>0
```

But exactly *which* programs should be rejected in this way? For example, consider `f2`:

```
f2 (T1 n)  = n>0
f2 (T2 xs) = null xs
```

Since `null :: [a] -> Bool` returns a `Bool`, and `T2` is an ordinary (non-GADT) data constructor, the *only* way to type `f2` is with result `Bool`, so the programmer might be confused at being required to say so. After all, there is only one solution: why can't the compiler find it?

An exactly similar issue arises in relation to variables in the environment. Consider

```
h1 x (T1 n)  = x && n>0
h1 x (T2 xs) = null xs
```

Which of these two incomparable types should we infer?

$$\begin{array}{l} \texttt{h1} :: \forall a.\, a \rightarrow \texttt{T}\ a \rightarrow \texttt{Bool} \\ \texttt{h1} :: \forall a.\, \texttt{Bool} \rightarrow \texttt{T}\ a \rightarrow \texttt{Bool} \end{array}$$

Again, since neither is more general than the other, we should reject the program. But if we somehow know from elsewhere that `x` is a `Bool`, then there is no ambiguity, and we might prefer to accept the definition. Here is an example

```
h2 x (T1 n)  = x && n>0
h2 x (T2 xs) = not x
```

The key difficulty is that *a GADT pattern match brings local type constraints into scope*. For example in the `T1` branch of the definition of `f1`, we know that the constraint $a \sim$ `Bool` holds, where the second argument of `f1` has type `T` $a$.[1] Indeed, while the declaration for the GADT `T` above is very convenient for the programmer, it is quite helpful to re-express it with an explicit equality constraint, like this[2]:

```
data T :: *->* where
  T1 :: (a˜Bool) => Int -> T a
      -- Was: T1 :: Int -> T Bool
  T2 :: [a] -> T a
```

$$\begin{array}{rlrl} \text{Term variables} & x, y, z & \text{Type constructors} & S, T \\ \text{Type variables} & a, b, c & \text{Data constructors} & K \\ \text{Unification variables} & \alpha, \beta, \gamma & & \end{array}$$

Programs and data type declarations

$$\begin{array}{rcl} prog & ::= & dd_1 \ldots dd_n;\ e \\ dd & ::= & \texttt{data}\, T\ a_1 ... a_m\ \texttt{where} \\ \hline & & \texttt{K} ::\ \forall\, a_1 ... a_m, b_1 ... b_n . C \Rightarrow \\ & & \qquad \tau_1 \rightarrow ... \rightarrow \tau_p \rightarrow T\ a_1 ... a_m \end{array}$$

$$\begin{array}{lrcl} \text{Terms} & e & ::= & K \mid x \mid \lambda x.e \mid e\ e \\ & & \mid & \textsf{let}\ \{g = e\}\ \textsf{in}\ e \\ & & \mid & \textsf{let}\ \{g :: \sigma\ =\ e_1\}\ \textsf{in}\ e_2 \\ & & \mid & \textsf{case}\ e\ \textsf{of}\ [p_i \rightarrow e_i]_{i \in I} \\ \text{Patterns} & p & ::= & K\ x_1 ... x_n \\ & & & \\ \text{Type envt} & \Gamma & ::= & \{x_1 : \sigma_1, ..., x_n : \sigma_n\} \\ \text{Type variables} & \nu & ::= & \alpha \mid a \\ \text{Monotypes} & \tau, \upsilon & ::= & \nu \mid \tau \rightarrow \upsilon \mid T\ \bar{\tau} \\ \text{Type Schemes} & \sigma & ::= & \tau \mid \forall \bar{a}.C \Rightarrow \tau \\ & & & \\ \text{Constraints} & C, D & ::= & \tau \sim \tau \mid C \wedge C \mid \epsilon \\ \text{Impl. Constraints} & F & ::= & C \mid [\bar{\alpha}](\forall \bar{b}.C \supset F) \mid F \wedge F \\ & & & \\ \text{Unifiers} & \theta & ::= & \emptyset \mid \theta, \{\alpha := \tau\} \\ \text{Substitutions} & \phi & ::= & \emptyset \mid \phi, \{\nu := \tau\} \end{array}$$

$$\begin{array}{rcl} \mathit{fuv}(\tau) & = & \text{The free unification variables of } \tau \\ & & \text{(and similarly } \mathit{fuv}(\Gamma)\text{)} \end{array}$$

Substitution

$$\begin{array}{rcl} \phi(F_1 \wedge F_2) & = & \phi(F_1) \wedge \phi(F_2) \\ \phi([\bar{\alpha}]\forall \bar{b}.C \supset F) & = & [\mathit{fuv}(\phi(\bar{\alpha}))]\forall \bar{b}.\phi(C) \supset \phi(F) \end{array}$$

Substitution on $C$ and $\tau$ is conventional

Abbreviations

$$\begin{array}{rcl} \forall \bar{a}.\tau & \triangleq & \forall \bar{a}.\epsilon \Rightarrow \tau \\ [\bar{\alpha}](\forall \bar{b}.F) & \triangleq & [\bar{\alpha}](\forall \bar{b}.\epsilon \supset F) \end{array}$$

**Figure 1.** Syntax of Programs

You may imagine a value of type `T` $\tau$, built with `T1`, as a heap-allocated object with two fields: a value of type `Int`, and some *evidence* that $\tau \sim$`Bool`. When the value is *constructed* the evidence must be supplied; when the value is *de-constructed* (i.e. matched in a pattern) the evidence becomes available locally. While in many systems, including GHC, this "evidence" has no run-time existence, the vocabulary can still be helpful and GHC does use explicit evidence-passing in its intermediate language (SCPD07).

## 3. Formal setup

Before we can present our approach, we briefly introduce our language, and the general form of its type system.

### 3.1 Syntax

Figure 1 gives the syntax of terms and types, which should look familiar to Haskell programmers. A program consists of a set of data type declarations together with a term $e$. Terms consist of the lambda calculus, together with `let` bindings (perhaps with a user-supplied type signature), and simple `case` expressions to perform pattern matching. A data type declaration introduces a type constructor $T$ and one or more data constructors $K_i$, each of which is given a type signature. As described in Section 2, in the case of

[1] We consistently use "$\sim$" to denote type equalities, because "$=$" is used for too many other things.

[2] GHC allows both forms, and treats them as equivalent.

$$\boxed{C, \Gamma \vdash e : \tau} \qquad \boxed{C, \Gamma \vdash_p p \rightarrow e : \tau \rightarrow \upsilon}$$

$$(\textsc{Var})\ \frac{(x : \forall \bar{a}.\upsilon) \in \Gamma \quad \phi = \{\overline{a := \tau}\}}{C, \Gamma \vdash x : \phi(\upsilon)} \quad (\textsc{Con})\ \frac{\begin{array}{c} K :: \forall \bar{a}.D \Rightarrow \upsilon \\ \phi = \{\overline{a := \tau}\} \quad C \models \phi(D) \end{array}}{C, \Gamma \vdash K : \phi(\upsilon)} \quad (\textsc{Eq})\ \frac{\begin{array}{c} C, \Gamma \vdash e : \tau_1 \\ C \models \tau_1 \sim \tau_2 \end{array}}{C, \Gamma \vdash e : \tau_2}$$

$$(\textsc{Abs})\ \frac{C, \Gamma \cup \{x : \tau_1\} \vdash e : \tau_2}{C, \Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \quad (\textsc{App})\ \frac{C, \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1 e_2 : \tau_2}$$

$$(\textsc{Let})\ \frac{\begin{array}{c} C, \Gamma \cup \{g : \tau_1\} \vdash e_1 : \tau_1 \quad \bar{a} = fv(\tau_1) - fv(C, \Gamma) \\ C, \Gamma \cup \{g : \forall \bar{a}.\tau_1\} \vdash e_2 : \tau_2 \end{array}}{C, \Gamma \vdash \mathsf{let}\ \{g = e_1\}\ \mathsf{in}\ e_2 : \tau_2} \quad (\textsc{LetA})\ \frac{\begin{array}{c} C, \Gamma \cup \{g : \forall \bar{a}.\tau_1\} \vdash e_1 : \tau_1 \\ C, \Gamma \cup \{g : \forall \bar{a}.\tau_1\} \vdash e_2 : \tau_2 \end{array}}{C, \Gamma \vdash \mathsf{let}\ \{g :: \forall \bar{a}.\tau_1\ =\ e_1\}\ \mathsf{in}\ e_2 : \tau_2}$$

$$(\textsc{Case})\ \frac{\begin{array}{c} C, \Gamma \vdash e : \tau_1 \\ C, \Gamma \vdash_p p_i \rightarrow e_i : \tau_1 \rightarrow \tau_2 \quad \mathsf{for}\ i \in I \end{array}}{C, \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} : \tau_2} \quad (\textsc{Pat})\ \frac{\begin{array}{c} K :: \forall \bar{a}, \bar{b}.D \Rightarrow \upsilon_1 \rightarrow ... \rightarrow \upsilon_p \rightarrow T\ \bar{a} \\ fv(C, \Gamma, \overline{\tau}, \tau_r) \cap \bar{b} = \emptyset \quad \phi = \{\overline{a := \tau}\} \quad consistent(C \wedge \phi(D)) \\ C \wedge \phi(D), \Gamma \cup \phi\{x_1 : \upsilon_1, \ldots, x_p : \upsilon_p\} \vdash e : \tau_r \end{array}}{C, \Gamma \vdash_p K\ x_1 ... x_p \rightarrow e : T\ \bar{\tau} \rightarrow \tau_r}$$

**Figure 2.** Simple but over-permissive typing rules

$$(\textsc{True})\ \frac{}{C \models \epsilon} \quad (\textsc{Refl})\ \frac{}{C \models \tau \sim \tau} \quad (\textsc{Sym})\ \frac{C \models \tau \sim \upsilon}{C \models \upsilon \sim \tau}$$

$$(\textsc{Trans})\ \frac{C \models \tau_1 \sim \tau_2 \quad C \models \tau_2 \sim \tau_3}{C \models \tau_1 \sim \tau_3}$$

$$(\textsc{Given})\ \frac{}{C_1 \wedge C_2 \models C_2} \quad (\textsc{Conj})\ \frac{C \models F_1 \quad C \models F_2}{C \models F_1 \wedge F_2}$$

$$(\textsc{Struct})\ \frac{\overline{C \models \tau_i \sim \upsilon_i}}{C \models T\ \bar{\tau}_i \sim T\ \bar{\upsilon}_i} \quad (\textsc{TCon})\ \frac{C \models T\ \bar{\tau}_i \sim T\ \bar{\upsilon}_i}{C \models \tau_i \sim \upsilon_i}$$

$$(\textsc{Impl})\ \frac{C \wedge C_1 \models F \quad \bar{b} \cap fv(C) = \emptyset}{C \models [\bar{\alpha}](\forall \bar{b}.C_1 \supset F)}$$

**Figure 3.** Equality theory

GADTs the data constructor’s type contains a set of constraints $D$, that are brought into scope when $K$ is used in a pattern match, and required when $K$ is used as a constructor.

The syntax of types, and of constraints, is also given in Figure 1. Note that unification variables $\alpha$ denote unknown types and only appear during type inference, never in the resulting typings. To avoid clutter we use only equality constraints $\tau_1 \sim \tau_2$ in our formalism, although in GHC there are several other sorts of constraint, including implicit parameters and type classes. We treat conjunction ($\wedge$) as a commutative and associative operator as is conventional. Implication constraints $F$ will be introduced in Section 4.4.

### 3.2 Type system

The declarative specification of a type system usually takes the form of a typing judgement

$$\Gamma \vdash e : \tau$$

with the meaning “in type environment $\Gamma$ the term $e$ has type $\tau$”. In a system with GADTs, however, a pattern match may bring into scope some *local* equality constraints. The standard way to express this is with the judgement

$$C, \Gamma \vdash e : \tau$$

meaning “in a context where constraints C are in scope, and type environment $\Gamma$ the term $e$ has type $\tau$”. For example, here is a valid judgement:

$$(a \sim \texttt{Bool}), \{\texttt{x} : a, \texttt{not} : \texttt{Bool} \rightarrow \texttt{Bool}\} \vdash \texttt{not x} : \texttt{Bool}$$

The judgement only holds because of the availability of the local equality $a \sim \texttt{Bool}$.

The type system of Figure 2 takes exactly this form. For example, rule (Con) instantiates the type scheme of a data constructor in the usual way, except that it has the additional premise

$$C \models \phi(D)$$

This requires that the “wanted” constraints $\phi(D)$ must be deducible from the “given” constraints $C$. To be concrete we give the (routine) definition of $\models$ in Figure 3. Compare rule (Con) to rule (Var), where the type scheme does not mention constraints.

Rule (EQ) allows us to use the available constraints $C$ to adjust the result type $\tau_1$ to any equal type $\tau_2$. Finally, a `case` expression uses an auxiliary judgement $\vdash_p$ to typecheck the case alternatives. Notice the way that the local constraints $C$ are extended when going inside a pattern match (in rule (PAT)), just as the type environment is augmented when going inside a lambda-term (in rule (ABS)).

Whenever we go inside a pattern match, we require the given constraint $C \wedge \phi(D)$ to be *consistent*, defined by the rule:

$$(\textsc{Consistent})\ \frac{\exists \phi.\ \epsilon \models \phi(C)}{consistent(C)}$$

i.e. $C$ is consistent if it has a unifier. Consistency implies that we will reject programs with inaccessible case branches.

A second point to notice about rule (PAT) is that a data constructor may have existential type variables $\bar{b}$ as well as universal type variables $\bar{a}$[3]. Rule (PAT) must check that the existential variables

[3] The former are called existential, despite their apparent quantification with $\forall$, because the constructor’s type is isomorphic to $K :: \forall \bar{a}.(\exists \bar{b}.D \times \upsilon_1 \times ... \times \upsilon_p) \rightarrow T\ \bar{a}$.

are not mentioned in the environment $C, \Gamma$, or the scrutinee type $T\ \overline{\tau}$, or the result type $\tau_r$. In the following example, `fx1` is well-typed, but `fx2` is not because the existential variable `b` escapes:

```
data X where
  X1 :: forall b. b -> (b->Int) -> X

fx1 (X1 x f) = f x
fx2 (X1 x f) = x
```

### 3.3 Properties

The type checking problem for GADTs is decidable (CH03; SP07). However, type inference turns out to be extremely difficult. The example from Section 2 shows that GADTs lack principal types. The difficulty is that the type system can type *too many terms*. Hence, our goal is to restrict the type system to reject just enough programs to obtain a tractable type inference system which enjoys principal types. Nevertheless, we regard Figure 2 as the "natural" type system for GADTs, against which any such restricted system should be compared.

## 4. A new approach

In this section we describe our new approach to type inference for GADTs. Type system designers often develop a type inference algorithm hand-in-hand with the specification of the type system: there is no point in a specification that we cannot implement, or an implementation whose specification is incomprehensible. We begin with the inference algorithm.

### 4.1 Type inference by constraint solving

It is well known that type inference can be carried out in two stages: first *generate constraints* from the program text, and then *solve the constraints* ignoring the program text (PR05). The generated constraints involve *unification variables*, which stand for as-yet-unknown types, and solving the constraints produces a *substitution* that assigns a type to each unification variable. The most basic form of constraint is a *type equality constraint* of form $\tau_1 \sim \tau_2$, where $\tau_1$ and $\tau_2$ are types.

For example, consider the definition

```
data Pair :: *->*->* where
  MkP :: a -> b -> Pair a b

f = \x -> MkP x True
```

The data type declaration specifies the type of the constructor `MkP`, thus:

$$\texttt{MkP} : \forall ab.\, a \rightarrow b \rightarrow \texttt{Pair}\ a\ b$$

Now consider the right-hand-side of `f`. The constraint generator makes up unification variables as follows:

| | |
|---|---|
| $\alpha$ | type of the entire right-hand side |
| $\beta_x$ | type of `x` |
| $\gamma_1, \gamma_2$ | instantiate $a, b$ respectively, when instantiating the call of `MkP` |

From the text we can generate the following equalities:

| | |
|---|---|
| $\beta_x \sim \gamma_1$ | First argument of `MkP` |
| $\texttt{Bool} \sim \gamma_2$ | Second argument of `MkP` |
| $\alpha \sim \texttt{Pair}\ \gamma_1\ \gamma_2$ | Result of `MkP` |

These constraints can be solved by unification, yielding the substitution $\{\alpha := \texttt{Pair}\ \beta_x\ \texttt{Bool}, \gamma_2 := \texttt{Bool}, \gamma_1 := \beta_x\}$. This substitution constitutes a "solution", because under that substitution the constraints are all of form $\tau \sim \tau$. Not only that, but the unification algorithm finds the *most general* substitution that solves the constraints. Temporarily leaving aside the question of generalization that's all there is to type inference for ML.

### 4.2 Constraint solving with GADTs

What happens when GADTs enter the picture? Consider our standard example term:

```
\x -> case x of { T1 n -> n>0 }
```

recalling the type of `T1`:

$$\texttt{T1} : \forall a.(\texttt{Bool} \sim a) \Rightarrow \texttt{Int} \rightarrow \texttt{T}\ a$$

Again we make up fresh unification variables for any unknown types:

| | |
|---|---|
| $\alpha$ | type of the entire right-hand side |
| $\beta_x$ | type of `x` |

Matching $x$ against a constructor from type `T` imposes the constraint $\beta_x \sim T\ \gamma$, for some new unification variable $\gamma$. From the term `n>0` we get the constraint $\alpha \sim \texttt{Bool}$, but that arises inside the branch of a `case` that brings into scope the constraint $\gamma \sim \texttt{Bool}$. We combine these two into a new sort of constraint, called an *implication constraint*:

$$\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool}$$

Now our difficulty becomes clear: there is no most-general unifier for implication constraints. The substitutions

$$\{\alpha := \texttt{Bool}\} \quad \text{and} \quad \{\alpha := \gamma\}$$

are both solutions, but neither is more general than the other.

On the other hand, sometimes there obviously *is* a unique solution. Consider `f2` from Section 2:

```
\x -> case x of { T1 n -> n>0; T2 xs -> null xs }
```

From the two alternatives of the `case` we get two constraints, respectively:

$$\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool} \quad \textit{and} \quad \alpha \sim \texttt{Bool}$$

Since the second constraint can be solved only by $\{\alpha := \texttt{Bool}\}$, there is a unique most-general unifier to this system of constraints.

### 4.3 GADT type inference is undecidable

Multiple pattern clauses give rise to a conjunction of implication constraints

$$(C_1 \supset C_1') \wedge ... \wedge (C_n \supset C_n')$$

The task of GADT type inference is to find a substitution $\theta$ such that each $\theta(C_i')$ follows from $\theta(C_i)$. This problem is identical to the simultaneous rigid E-unification problem which is known to be undecidable (DV95). Hence, we can immediately conclude that GADT type inference is undecidable in the unrestricted type system. To restore decidability and most general solutions, we consider a restricted implication solver algorithm.

### 4.4 The OutsideIn solving algorithm

Our idea is a simple one: *we must refrain from unifying a global unification variable under a local equality constraint*. By "global" we mean "free in the type environment[4]", and we must record that information in the implication constraint itself, thus

$$[\alpha]\ (\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool})$$

because $\alpha$ is free in the type environment. Here $\gamma \sim \texttt{Bool}$ is a given equality constraint that may *only locally* be assumed to hold, i.e., to solve the constraint to the right of the implication sign: $\alpha \sim \texttt{Bool}$.

[4] We must treat the result type as part of the type environment.

When solving this constraint we must refrain from unifying $\{\alpha := \texttt{Bool}\}$; hence, the constraint by itself is insoluble. It can be solved only if there is some *other* constraint that binds $\alpha$.

The syntax of implication constraints $F$ is given in Figure 1. An implication constraint is an ordinary constraint $C$, or a conjunction of implications, or has the form $[\bar{\alpha}]\forall\bar{b}.C \supset F$. We call the set of unification variables $\bar{\alpha}$ the *untouchables* of the constraint, and the set of type variables $\bar{b}$ the *skolems* of the constraint. Applying a substitution to an implication constraint requires a moment's thought, because an untouchable might be mapped to a type by the substitution, so we must take the free unification variables of the result; see Figure 1. We often omit the untouchables, skolems, or $C$ when they are empty.

More precisely, to solve a set of implication constraints $F$, proceed as follows:

1. Split $F$ into $F_g \wedge F_s$, where all the constraints in $F_g$ are proper implications, and $F_s$ are all simple. An implication is *simple* if it does not involve any local equalities, and *proper* otherwise:

$$\begin{array}{lcll} F_g & ::= & F_g \wedge F_g \mid [\bar{\alpha}]\forall\bar{b}.C \supset F & C \not\equiv \epsilon \\ F_s & ::= & F_s \wedge F_s \mid C \mid [\bar{\alpha}]\forall\bar{b}.\epsilon \supset F_s & \end{array}$$

2. Solve the simple constraints $F_s$ by ordinary unification, yielding a substitution $\theta$.
3. Now apply $\theta$ to $F_g$, and solve each implication in $\theta(F_g)$.

In the last step, how do we solve a proper implication $[\bar{\alpha}]\forall\bar{b}.C \Rightarrow F$? Simply find $\phi$, the most general unifier of $C$, and solve $\phi(F)$, under the restriction that the solution must not bind $\bar{\alpha}$.

This algorithm is conservative: if it finds a unifier, that solution will be most general, but the converse is not true. For example, the algorithm fails to solve the constraint

$$[\alpha]\ (\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Int})$$

but the constraint actually has a unique solution, namely $\{\alpha := \texttt{Int}\}$.

## 5. The OutsideIn approach in detail

It is time to nail down the details. Our approach relies on constraint generation and constraint solving. We specify top-level constraint generation with $\Gamma \vdash_W e : \tau, F$ to be read as: in the environment $\Gamma$, we may infer type $\tau$ for the expression $e$ and generate constraint $F$. Solving a constraint $F$ to produce a substitution $\theta$ is specified with $\vdash_s F : \theta$. The top-level inference algorithm is then given by the judgement $\vdash_{\mathsf{inf}}$ in Figure 4.[5]

We start by discussing constraint generation (Section 5.1) and constraint solving (Section 5.2). Subsequently we present the high-level declarative type system (Section 6).

### 5.1 Generating implication constraints

The constraint generation algorithm is given in Figure 4 with the judgement

$$\Gamma \vdash_W e : \tau, F$$

In this judgement, thought of as an algorithm, $\Gamma$ and $e$ are inputs, while $\tau$ and $F$ are outputs.

Rules (VAR), (CON), (ABS), and (APP) are straightforward. Rule (PAT) generates an implication constraint, as described informally in Section 4.2. Rule (CASE) "peeks" inside the pattern match alternatives to determine the constructor type $T$ (by calling $\mathit{constructor}(p_i)$) and subsequently pushes the type of $e$ ($T\ \bar{\alpha}$) in the typing clause for each alternative (rule (PAT)). Finally rule (CASE) returns the constraints arising from the alternatives.

$$\boxed{\vdash_{\mathsf{inf}} e : \tau}$$

$$(\text{INFER})\quad \frac{\vdash_W e : \tau, F \qquad \vdash_s F : \theta}{\vdash_{\mathsf{inf}} e : \theta(\tau)}$$

$$\boxed{\Gamma \vdash_W e : \tau, F}$$

$$(\text{VAR})\quad \frac{(x : \forall\bar{a}.\tau) \in \Gamma \quad \bar{\alpha}\ \text{fresh} \quad \phi = \{\overline{a := \alpha}\}}{\Gamma \vdash_W x : \phi(\tau), \epsilon}$$

$$(\text{CON})\quad \frac{K :: \forall\bar{a}.C \Rightarrow \tau \quad \bar{\alpha}\ \text{fresh} \quad \phi = \{\overline{a := \alpha}\}}{\Gamma \vdash_W K : \phi(\tau), \phi(C)}$$

$$(\text{APP})\quad \frac{\begin{array}{c}\Gamma \vdash_W e_1 : \tau_1, F_1 \quad \Gamma \vdash_W e_2 : \tau_2, F_2 \\ \alpha\ \text{fresh} \quad F = F_1 \wedge F_2 \wedge (\tau_1 \sim \tau_2 \rightarrow \alpha)\end{array}}{\Gamma \vdash_W e_1\ e_2 : \alpha, F}$$

$$(\text{ABS})\quad \frac{\alpha\ \text{fresh} \quad \Gamma \cup \{x : \alpha\} \vdash_W e : \tau, F}{\Gamma \vdash_W \lambda x.e : \alpha \rightarrow \tau, F}$$

$$(\text{LETA})\quad \frac{\begin{array}{c}\Gamma \cup \{g : \forall\bar{a}.\tau\} \vdash_W e_1 : \tau', F_1 \\ \Gamma \cup \{g : \forall\bar{a}.\tau\} \vdash_W e_2 : \upsilon, F_2 \\ F = F_2 \wedge [\mathit{fuv}(\Gamma)](\forall\bar{a}.F_1 \wedge \tau \sim \tau'))\end{array}}{\Gamma \vdash_W \mathsf{let}\ \{g :: \forall\bar{a}.\tau\ =\ e_1\}\ \mathsf{in}\ e_2 : \upsilon, F}$$

$$(\text{LET})\quad \frac{\begin{array}{c}\alpha\ \text{fresh} \quad \Gamma \cup \{g : \alpha\} \vdash_W e_1 : \tau, F_1 \\ F_1' = F_1 \wedge \alpha \sim \tau \quad F_s = \mathit{simple}(F_1') \quad \vdash_s F_s : \phi_s \\ \overline{\beta} = \mathit{fuv}(\phi_s(\tau)) - \mathit{fuv}(\phi_s(\Gamma)) \\ \bar{b}\ \text{fresh} \quad \theta_k = \{\overline{\beta := b}\} \\ \Gamma \cup \{g : \forall\bar{b}.\theta_k(\phi_s(\tau))\} \vdash_W e_2 : \upsilon, F_2 \\ F = F_2 \wedge [\mathit{fuv}(\Gamma)]\forall\bar{b}.\theta_k(F_1')\end{array}}{\Gamma \vdash_W \mathsf{let}\ \{g\ =\ e_1\}\ \mathsf{in}\ e_2 : \upsilon, F}$$

$$(\text{CASE})\quad \frac{\begin{array}{c}T = \mathit{constructor}(p_i) \quad \text{for}\ i \in I \\ \Gamma \vdash_W e : \tau_e, F_e \quad \bar{\alpha}, \beta\ \text{fresh} \\ \Gamma \vdash_P p_i \rightarrow e_i : T\ \bar{\alpha} \rightarrow \beta, F_i \quad \text{for}\ i \in I \\ F = F_e \wedge \tau_e \sim T\ \bar{\alpha} \wedge \bigwedge_{i \in I} F_i\end{array}}{\Gamma \vdash_W \mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I} : \alpha, F}$$

$$\boxed{\Gamma \vdash_P p \rightarrow e : \tau \rightarrow \upsilon, F}$$

$$(\text{PAT})\quad \frac{\begin{array}{c}K :: \forall\bar{a},\bar{b}.D \Rightarrow \upsilon_1 \rightarrow \ldots \rightarrow \upsilon_p \rightarrow T\ \bar{a} \\ \bar{b} \notin ftv(\Gamma, \tau_r) \quad \phi = \{\overline{a := \tau}\} \\ \Gamma \cup \phi\{x_1 : \upsilon_1, \ldots, x_p : \upsilon_p\} \vdash_W e : \tau_e, F_e \\ F = [\overline{\alpha} \cup \mathit{fuv}(\Gamma, \tau_e)](\forall\bar{b}.\phi(D) \supset F_e \wedge \tau_e \sim \tau_r)\end{array}}{\Gamma \vdash_P K\ x_1 \ldots x_p \rightarrow e : T\ \overline{\tau} \rightarrow \tau_r, F}$$

**Figure 4.** Translation to Constraints

Rule (LETA) generates implication constraints for an *annotated* let-binding. Pay attention to two details: (a) the inferred type $\tau'$ must

[5] We start with an initially-empty environment, informally relying on a fixed, implicit global environment to specify the types of each data constructor.

equate to the declared type $\tau$, and (b) the universally quantified variables $\bar{a}$ must not escape their scope. The first is captured in an additional equality constraint $\tau \sim \tau'$, and the latter in the degenerate implication constraint.

Rule (LET) for *unannotated* let-expressions is much trickier. First, it derives the constraint $F_1$ for the bound expression $e_1$. The conventional thing to do at this point is to create fresh type variables $\bar{b}$ for the variables $\bar{\beta}$ that are not free in the environment with a substitution $\theta_k = \{\overline{\beta := b}\}$, and abstract over the constraints, inferring the following type for g (SP07):

$$g : \forall \bar{b}.\theta_k(F_1 \Rightarrow \tau)$$

This is correct, but by postponing all solving until the second phase we get unexpectedly complicated types for simple definitions. For example, from the definition

```
g = \x -> x && True
```

we would infer the type

$$g :: \forall b.b \sim \texttt{Bool} \Rightarrow b \rightarrow \texttt{Bool}$$

when the programmer would expect the equivalent but simpler type `Bool` $\rightarrow$ `Bool`. Furthermore, this approach obviously requires that types can take the form $F \Rightarrow \tau$ — including the possibility that $F$ is itself an implication! It all works fine (see (SP07) for example), but it makes the types significantly more complicated and, in an evidence-passing internal language such as that used by GHC, creates much larger elaborated terms.

Instead, we interleave constraint *generation* and constraint *solving* in rule (LET), thus[6]:

- Generate constraints for $e_1$ under the assumption that $g : \alpha$ (we allow recursion in `let`).
- Add the constraint $\alpha \sim \tau$ to tie the recursive knot in the usual way, forming $F_1'$.
- We cannot, at this stage, guarantee to solve *all* the constraints in $F_1'$, because the latter might include implications that can only be solved in the presence of information from elsewhere. So we extract from $F_1'$ the "simple" constraints, $F_s$:

$$\begin{array}{rcl} simple(C) & = & C \\ simple(F_1 \wedge F_2) & = & simple(F_1) \wedge simple(F_2) \\ simple([\bar{\alpha}](\forall \bar{b}.F)) & = & [\bar{\alpha}]\forall \bar{b}.simple(F) \\ simple([\bar{\alpha}](\forall \bar{b}.C \supset F)) & = & \epsilon \quad C \not\equiv \epsilon \end{array}$$

- Solve $F_s$ appealing to our solver $\vdash F_s : \phi_s$. Notice that this $\phi_s$ may bind skolems, a point that we will return to. Moreover, notice that if unification fails for a simple constraint (such as `Bool` $\sim$ `Int`) then the program is definitely untypeable.
- Apply the solving substitution $\phi_s$ to $\tau$ and $\Gamma$, and compute the set of variables $\bar{\beta}$ over which to quantify in the usual way.
- Skolemise the variables we can quantify over, using a substitution $\theta_k$.
- Typecheck the body of the `let`, with a suitable type for $g$.
- Lastly we figure out the constraint $F$ to return. It includes $F_2$ of course, and $F_1'$ suitably wrapped in a $\forall$ to account for the skolemized variables just as in (LETA).

There are two tricky points in this process.

$$\boxed{\vdash_s \; F : \phi}$$

$$\text{(S-SOLVE)} \; \frac{simple(F) = F_s \quad \vdash_s^\star F_s : \phi \quad \vdash_s^\star \phi(F) : \theta}{\vdash_s^\star F : \theta \cdot \phi}$$

$$\boxed{\vdash_s^\star F : \phi}$$

$$\text{(S-EMPTY)} \; \frac{}{\vdash_s^\star \epsilon : \emptyset} \qquad \text{(S-SPLIT)} \; \frac{\begin{array}{c}\vdash_s^\star F_1 : \phi_1 \\ \vdash_s^\star \phi_1(F_2) : \phi_2\end{array}}{\vdash_s^\star F_1 \wedge F_2 : \phi_2 \cdot \phi_1}$$

$$\text{(S-REFL)} \; \frac{}{\vdash_s^\star \tau \sim \tau : \emptyset} \qquad \text{(S-CONS)} \; \frac{\vdash_s^\star (\bigwedge_i \tau_i \sim \tau_i') : \phi}{\vdash_s^\star T\,\bar{\tau} \sim T\,\bar{\tau}' : \phi}$$

$$\text{(S-UL)} \; \frac{\begin{array}{c}\nu \notin \tau \\ \phi = \{\nu := \tau\}\end{array}}{\vdash_s^\star \nu \sim \tau : \phi} \qquad \text{(S-UR)} \; \frac{\begin{array}{c}\nu \notin \tau \\ \phi = \{\nu := \tau\}\end{array}}{\vdash_s^\star \tau \sim \nu : \phi}$$

$$\text{(S-SIMPL)} \; \frac{\vdash_s^\star F : \phi \quad fv(\phi(\bar{\alpha})) \cap \bar{b} = \emptyset \quad \bar{b} \cap dom(\phi) = \emptyset}{\vdash_s^\star [\bar{\alpha}](\forall \bar{b}.F) : \phi}$$

$$\text{(S-PIMPL)} \; \frac{\begin{array}{c}C \not\equiv \epsilon \quad \vdash_s^\star C : \phi \\ \vdash_s \; \phi(F) : \theta \quad \bar{\alpha} \cap dom(\theta) = \emptyset\end{array}}{\vdash_s^\star [\bar{\alpha}](\forall \bar{b}.C \supset F) : \theta}$$

**Figure 5.** Solver algorithm

- First, notice that the substitution returned by solving $F_s$ is a $\phi$-substitution and not merely a $\theta$-substitution, and hence can bind skolem variables. This is perhaps unintuitive—after all in ordinary Hindley-Milner we would require that the substitution binds only unification variables. Nevertheless, in the presence of given equations this is not enough. Consider:

```
data T where
  MkT :: forall a b. (a ~ b) => a -> b -> T

foo = case e of
        MkT y z -> let h = [y,z] in ()
```

  Constraint generation for the inner `let` definition produces the constraint $a \sim b$, where $a$ and $b$ are the existential variables introduced by the pattern match. But we must not fail at this point and hence the solver of the $F_s$ constraint must be prepared to encounter equalities between skolem variables.
- Second, notice that we do *not* apply $\phi_s$ to $F_1'$. Why not? Because $\phi_s$ may bind variables free in $\Gamma$, and we must not lose that information. But the very same information is present in the original $F_1'$, so if we return $F_1'$ unchanged (apart from applying the skolemizing substitution $\theta_k$, then the rest of the derivation will be able to "see" it too.

Finally, notice that there is quite a bit of "junk" in $F_1'$. Consider the definition of g given earlier in this subsection. We will get

$$\tau = \beta \rightarrow \texttt{Bool}, \; F_1' = \beta \sim \texttt{Bool}, \; \theta_s = \{\beta := \texttt{Bool}\}$$

Now $\beta$ plays no part in the rest of the program, but still lurks in $F_1'$. Because of our freshness assumptions, however, it does no harm either.

### 5.2 The OutsideIn Implication Solver

Figure 5 presents the rules of our implication solver. The solver judgement is of the form $\vdash_s \; F : \phi$. This judgement should

[6] This interleaving is not so unusual: every Haskell compiler does the same for type-class constraints. Alternatively, Pottier and Rémy (PR05) show how to defer quantification to the solving phase and avoid interleaving.

be thought of as taking $F$ as input and producing a $\phi$, such that $\models \phi(F)$ according to the equational theory of Figure 3. The judgement appeals to $simple(F)$ first, to extract the simple part of the constraint $F_s$. It solves the simple part using the auxiliary judgement $\vdash_s^\star F : \phi$. It applies the substitution to the original constraint and tries to solve the returned constraint.[7]

Notice that the solver returns a $\phi$ substitution, which can bind both skolem variables and unification variables. As discussed in the previous section, being able to handle equalities between skolem variables is important for the interleaving of solving and constraint generation in rule (LET). Nevertheless, only a $\theta$ is returned the second time we attempt to solve the constraints. This is because the second time the solver will attempt to solve the proper implications that remain – and solutions to those may only bind unification variables as we shall shortly see (rule (S-PIMPL)).

The judgement $\vdash_s^\star F : \phi$ is the core of our constraint solver. Rules (S-EMPTY) and (S-SPLIT) are straightforward. The remaining rules deal with a single equality constraint. Rule (S-CONS) deconstructs a type constructor application, and Rule (S-REFL) discharges trivial equality constraints. Rules (S-UL) and (S-UR) actually instantiate a type variable variable $\nu$ with a type $\tau$. They must be careful not to violate the occurs-check ($\nu \notin \tau$).

Simple implication constraints, i.e. with empty given constraints, are treated by the (S-SIMPL) rule. A simple implication constraint is treated almost as if it were just a basic constraint with two differences. First, we make sure that the returned $\phi$ does not unify any of the skolemized variables of the constraint – it would be unsound to do otherwise. Second, we must never instantiate any of the variables captured in $[\bar{\alpha}]$ with a type that contains some of the skolemized variables $\bar{b}$. (In this case "untouchables" for the $\bar{\alpha}$ variables is a bad name. For example, it is fine – indeed essential – to unify $\alpha$ in $[\alpha]\forall b.\alpha \sim \mathtt{Bool}$.)

Proper implication constraints are tackled by the (S-PIMPL) rule. First it computes $\phi$ that solves the assumptions $C$ — if there is no solution, the implication constraint originates from a dead code branch. Next, it applies it to $F$ and solves $F$ recursively yielding $\theta$. Finally, it checks that the solution $\theta$ does not touch any of the untouchables.

There are several tricky points:

- There is some non-determinism in rule (S-SPLIT), but it is harmless. When solving simple constraints, the order of solving them does not matter; and when solving conjunctions of proper constraints solutions from one can never affect the other.
- Some non-determinism appears in (S-PIMPL). For example, consider the constraint $[\alpha]\forall.C \supset \alpha \sim \beta$. The recursive invocation of $\vdash_s$ could return either $\phi = \{\alpha := \beta\}$ or $\{\beta := \alpha\}$, but only one will satisfy the untouchables check. In contrast, *any* most-general unifier of $C$ will do for $\phi$. Similarly, in a simple constraint $[]\forall c.\beta \sim c$ there is a choice to bind either $\beta$ or $c$ when we solve the constraint $\beta \sim c$. However, because of the conditions in rule (S-SIMPL), only the solution $\{\beta := c\}$ is acceptable.
- Rule (S-PIMPL) does not need the skolem-escape check that appears in (S-SIMPL). Because $\theta$ does not affect $\alpha$, such a check cannot fail.
- In (S-PIMPL), solving $C$ requires us to bind skolem variables as well as unification variables, and hence we return a $\phi$. This is important to type constraints whose assumptions involve skolem variables, such as $(a \sim \mathtt{Int} \supset \mathtt{Int} \sim a)$. Furthermore, in rule (S-PIMPL) the solution of the right-hand side of the constraint is required to be a $\theta$. The reason is because this rule is triggered whenever we are trying to solve a proper implication constraint, and hence the solver is *not* called from rule (LET), but rather *after* constraint generation has finished. In order to solve such a constraint at the end, it is unsound to bind skolem variables: Equalities that involve skolems may only be discharged by given equalities. Hence we must not return a $\phi$, but a substitution that binds unification variables only (i.e. a $\theta$).

### 5.3 Example

Consider again our standard example

```
\x -> case x of { T1 n -> n>0 }
```

for which the type $\alpha_x \rightarrow \beta$ is derived, and the constraint

$$F = \alpha_x \sim \mathtt{T}\ \alpha \wedge [\alpha, \alpha_x, \beta](\alpha \sim \mathtt{Bool} \supset \beta \sim \mathtt{Bool})$$

If we solve first the simple constraint on the left, we get the substitution $[\alpha_x := \mathtt{T}\ \alpha]$. We apply this substitution on the implication constraint, yielding $[\alpha, \beta](\alpha \sim \mathtt{Bool} \supset \beta \sim \mathtt{Bool})$. Next, we try to solve the implication constraint. Firstly, applying the mgu of $\alpha \sim \mathtt{Bool}$, i.e. $[\alpha := \mathtt{Bool}]$, to $\beta \sim \mathtt{Bool})$ has no impact. Secondly, we try to solve $\beta \sim \mathtt{Bool}$ by substituting $\beta$ for $\mathtt{Bool}$. Yet this fails, because $\beta$ is an "untouchable". Hence, our algorithm rejects the program.

Now let's add a second branch to the example

```
\x -> case x of { T1 n -> n>0 ; T2 xs -> null xs }
```

Again the type $\alpha_x \rightarrow \beta$ is derived, now with the constraint

$$F' = F \wedge \alpha_x \sim \mathtt{T}\ \alpha' \wedge \mathtt{[}\alpha'\mathtt{]} \sim \mathtt{[}\alpha''\mathtt{]} \wedge \beta \sim \mathtt{Bool}$$

The first additional constraint originates from the pattern `T2 xs`, the second and third from `null xs`. Solving all simple constraints first, we get the substitution $[\alpha_x := \mathtt{T}\ \alpha, \alpha' := \alpha, \alpha'' := \alpha, \beta := \mathtt{Bool}]$. These reduce the implication constraint to $[\alpha](\alpha \sim \mathtt{Bool} \supset \mathtt{Bool} \sim \mathtt{Bool})$, which is now readily solved. Hence, the expression is accepted with type $\mathtt{T}\ \alpha \rightarrow \mathtt{Bool}$.

## 6. Specifying the restricted type system

It is all very well having an inference algorithm, but must also explain to the programmer which programs are accepted by the type checker and which are not. Every GADT inference algorithm has difficulty with this point, and ours is no exception.

Figure 6 presents the rules of the restricted type system. The top-level typing judgement is $C, \Gamma \vdash_R e : \tau$ which asserts that expression $e$ has type $\tau$ with respect to environment $\Gamma$ and type constraints $C$. This judgement is defined by rule (R-MAIN) which in turn is defined in terms of the auxiliary judgement

$$C, \Gamma \vdash_r e : \tau, P$$

which should be read "under constraints $C$ and type environment $\Gamma$, the term $e$ has type $\tau$ and suspended typing judgements $P$".

What are these suspended judgements? The idea is that we type-check the original term *simply ignoring* any GADT case alternatives. Instead, these ignored alternatives, along with the current environment and result type, are collected in a set $P$ of tuples $\langle C, \Gamma, e, \tau \rangle$. Suppose the original top-level program term can be typed, so that

$$\Gamma \vdash_r e : \tau, P$$

holds. Then, for *every* such typing (or, more realistically for the principal typing) we require that *all* the suspended typing problems in $P$ are soluble. That is what the (rather complicated) rule

[7] A more realistic implementation would split the constraint, solve the simple part and use that substitution to solve the proper part – no need to re-solve the simple part. We chose our current formalism as it saves us the definition of splitting.

$$\boxed{C, \Gamma \vdash_R e : \tau}$$

$$\text{(R-M}\scriptstyle\text{AIN}\textstyle\text{)}\quad \frac{\begin{array}{c} consistent(C) \quad C, \Gamma \vdash_r e : \tau, P \\ \forall \tau', P'.\ (C, \Gamma \vdash_r e : \tau', P') \Rightarrow \forall \langle C_i, \Gamma_i, e_i, \tau_i \rangle \in P'.\ C_i, \Gamma_i \vdash_R e_i : \tau_i \end{array}}{C, \Gamma \vdash_R e : \tau}$$

$$\boxed{C, \Gamma \vdash_r e : \tau, P}$$

$$\text{(R-VAR)}\quad \frac{(x : \forall \bar{a}.\upsilon) \in \Gamma \quad \phi = \{\overline{a := \tau}\}}{C, \Gamma \vdash_r x : \phi(\upsilon), \emptyset} \qquad \text{(R-CON)}\quad \frac{\begin{array}{c} K :: \forall \bar{a}.D \Rightarrow \upsilon \\ \phi = \{\overline{a := \tau}\} \qquad C \models \phi(D) \end{array}}{C, \Gamma \vdash_r K : \phi(\upsilon), \emptyset} \qquad \text{(R-EQ)}\quad \frac{\begin{array}{c} C, \Gamma \vdash_r e : \tau_1, P \\ C \models \tau_1 \sim \tau_2 \end{array}}{C, \Gamma \vdash_r e : \tau_2, P}$$

$$\text{(R-APP)}\quad \frac{C, \Gamma \vdash_r e_1 : \tau_1 \to \tau_2, P_1 \quad C, \Gamma \vdash_r e_2 : \tau_1, P_2}{C, \Gamma \vdash_r e_1\, e_2 : \tau_2, P_1 \cup P_2} \qquad \text{(R-ABS)}\quad \frac{C, \Gamma \cup \{x : \tau_1\} \vdash_r e : \tau_2, P}{C, \Gamma \vdash_r \lambda x.e : \tau_1 \to \tau_2, P}$$

$$\text{(R-LET)}\quad \frac{\begin{array}{c} C, \Gamma \cup \{g : \tau_1\} \vdash_r e_1 : \tau_1, P_1 \quad \bar{a} = fv(\tau_1) - fv(C, \Gamma) \\ C, \Gamma \cup \{g : \forall \bar{a}.\tau_1\} \vdash_r e_2 : \tau_2, P_2 \end{array}}{C, \Gamma \vdash_r \mathsf{let}\ \{g = e_1\}\ \mathsf{in}\ e_2 : \tau_2, P_1 \cup P_2} \qquad \text{(R-LETA)}\quad \frac{\begin{array}{c} C, \Gamma \cup \{g : \forall \bar{a}.\tau_1\} \vdash_r e_1 : \tau_1, P_1 \\ C, \Gamma \cup \{g : \forall \bar{a}.\tau_1\}, \vdash_r e_2 : \tau_2, P_2 \end{array}}{C, \Gamma \vdash_r \mathsf{let}\ \{g :: \forall \bar{a}.\tau_1 \ = \ e_1\}\ \mathsf{in}\ e_2 : \tau_2, P_1 \cup P_2}$$

$$\text{(R-CASE)}\quad \frac{C, \Gamma \vdash_r e : \tau_1, P \quad C, \Gamma \vdash_{rp} p_i \to e_i : \tau_1 \to \tau_2, P_i \quad \mathsf{for}\ i \in I}{C, \Gamma \vdash_r \mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I} : \tau_2, P \cup \bigcup_{i \in I} P_i}$$

$$\boxed{C, \Gamma \vdash_{rp} p \to e : \tau \to \upsilon, P}$$

$$\text{(R-VPAT)}\quad \frac{\begin{array}{c} K : \forall \bar{a}, \bar{b}.\epsilon \Rightarrow \upsilon_1 \to \ldots \to \upsilon_p \to T\, \bar{a} \\ fv(C, \Gamma, \bar{\tau}, \tau_r) \cap \bar{b} = \emptyset \quad \phi = \{\overline{a := \tau}\} \\ C, \Gamma \cup \phi\{x_1 : \upsilon_1, \ldots, x_p : \upsilon_p\} \vdash_r e : \tau_r, P \end{array}}{C, \Gamma \vdash_{rp} K\, x_1 \ldots x_p \to e : T\, \bar{\tau} \to \tau_r, P} \qquad \text{(R-GPAT)}\quad \frac{\begin{array}{c} K : \forall \bar{a}, \bar{b}.D \Rightarrow \upsilon_1 \to \ldots \to \upsilon_p \to T\, \bar{a} \quad D \neq \epsilon \\ fv(C, \Gamma, \bar{\tau}, \tau_r) \cap \bar{b} = \emptyset \quad \phi = \{\overline{a := \tau}\} \\ P = \{\langle C \wedge \phi(D),\, \Gamma \cup \phi\{x_1 : \upsilon_1, \ldots, x_l : \upsilon_l\},\, e,\, \tau_r \rangle\} \end{array}}{C, \Gamma \vdash_{rp} K\, x_1 \ldots x_l \to e : T\, \bar{\tau} \to \tau_r, P}$$

**Figure 6.** Typing Rules for the Restricted Type System

(R-MAIN) says. It ensures that typing information from inside a GADT match does not influence the typing of code outside that match — just as the algorithm does. Observe the recursive nature of rule (R-MAIN), which defers and processes nested case expressions one layer at a time.

The only rule that adds a deferred typings to $P$ is R-GPAT; it defers the typing of a branch of a case expression that matches a GADT constructor pattern. This rule only applies to GADT constructors that bring a type equality into scope. In all other cases, when no new type equalities are brought into scope, the rule R-PAT applies, which does not defer the typing.

## 7. Formal properties

In this section we describe the properties of our type system and its inference algorithm.

### 7.1 Properties of the type system

As we have discussed, implication constraints arising from program text may have a finite or infinite set of incomparable solutions. This ambiguity makes type inference hard. Even in the case when the solutions are finite (but cannot be described by a common most general solution) modular type inference is impossible. Our restricted system however imposes conditions on the typeable programs of the unrestricted system, which ensure that we can perform tractable type inference without having to search the complete space of possibly incomparable solutions for the arising constraints.

First, the restricted type system is sound wrt. the unrestricted type system:

THEOREM 7.1 (Soundness). *If $\epsilon, \Gamma \vdash_R e : \tau$ in the restricted type system (Figure 6), then $\epsilon, \Gamma \vdash e : \tau$ in the unrestricted type system (Figure 2).*

It is fairly easy to see that this theorem holds. In addition to all the constraints of the unrestricted type system, the restricted type system imposes one more constraint on well-typing: the universal well-typing of the deferred typings discussed above.

Moreover, the restricted type system has the important property that it only admits expressions that have a principal type.

THEOREM 7.2 (Principal Typing in the Restricted Type System). *If an expression e is typeable in the restricted type system wrt. a type environment $\Gamma$, then there is a principal type $\tau_p$ such that $\epsilon, \Gamma \vdash_R e : \tau_p$ and such that for any other $\tau$ for which $\epsilon, \Gamma \vdash_R e : \tau$, there exists a substitution $\phi$ such that $\phi(\tau_p) = \tau$.*

Note that the principality is not an artifact of the restricted type system. A principal type in the restricted type system is also a principal type in the unrestricted type system:

THEOREM 7.3 (Principal Typing in the Unrestricted Type System). *Assume that the type $\tau_p$ is the principal type of e wrt. a type environment $\Gamma$ in the restricted type system. Then, for for any other $\tau$ for which $\epsilon, \Gamma \vdash e : \tau$, there exists a substitution $\phi$ such that $\phi(\tau_p) = \tau$.*

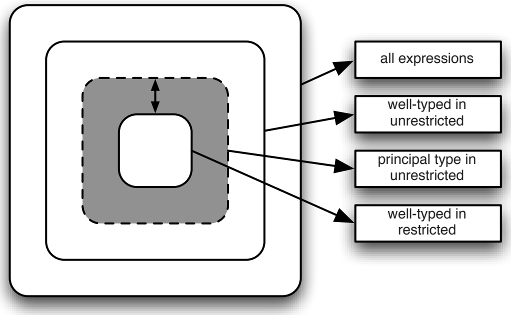

**Figure 7.** The space of programs

Note that not all programs with a principal type in the unrestricted type system, are accepted in the unrestricted type system.

Consider the following program:

```
data T a where MkT :: (a ~ Bool) => T a

f :: T a -> Char
f x = let h = case x of MkT -> 3
      in 'a'
```

The principal type for `h` in the unrestricted type system is `Int`. The restricted, ignoring the case branch, attempts to assign $\forall b.b$ as the principal type for `h`. However, this does not allow the implication $(a \sim \texttt{Bool}) \supset (b \sim \texttt{Int})$ to be solved. Hence, the program is rejected.

Hence, the gray area in Figure 7 is non-empty. We leave it as a challenge for future work to expand the innermost area towards the dashed line.

Finally, we observe that nearly all well-typings in the unrestricted type system can be recovered in the restricted type system by adding additional type annotations to the program. Because our language does not provide a means to name existential type variables brought into scope by GADT pattern matches, we cannot recover thos well-typings that require mentioning them. Open type annotations[8] would lift that limitation.

### 7.2 Properties of the inference algorithm

The solver algorithm has a number of vital properties. First, the search for solution always terminates, either in failure or success.

THEOREM 7.4 (Termination). *The solver algorithm terminates.*

Second, when a solution is found, it is a proper well-typing in the restricted type system.

THEOREM 7.5 (Soundness). *If* $\vdash_{inf} e : \tau$ *then* $\epsilon, \emptyset \vdash_R e : \tau$

Third, when a solution is found, it is not an arbitrary solution, but the principal solution.

THEOREM 7.6 (Principality). *The inferred type is the principal type in the restricted type system: If* $\vdash_{inf} e : \tau$ *and* $\epsilon, \emptyset \vdash_R e : \upsilon$ *then* $\upsilon = \phi(\tau)$ *for some* $\phi$*.*

Finally, if an expression is well-typed, then the solver algorithm finds a solution.

THEOREM 7.7 (Completeness). *If* $\epsilon, \emptyset \vdash_R e : \tau$ *then* $\vdash_{inf} e : \upsilon$*.*

Of course, in order to prove the solver algorithm properties, we have to generalize appropriately the statements of the theorems in this section, but we refrain from presenting the generalized statements for the sake of clarity of exposition.

## 8. Implementation Aspects

A key property of **OutsideIn** is that it is easy to implement, and the implementation is efficient. To substantiate this claim we briefly describe our implementation of **OutsideIn** in Haskell. Our implementation is available for download from

```
http://research.microsoft.com/people/dimitris/
```

and additionally supports bidirectional type checking, open type annotations, and type annotations with constraints.

We introduce a datatype `MetaTv` for unification variables $\alpha, \beta, \ldots$ and a datatype `TyVar` for skolem variables $a, b, \ldots$. As in traditional implementations (PVWS07), the `MetaTv` contains a reference cell that may contain a type to which the variable is bound:

```
data    MetaTv = Meta Name (IORef (Maybe Type))
newtype TyVar  = TyV Name
```

The main type checker is written in a monad `Tc a`, which is a function from environments `TcEnv` and encapsulates IO and threading of error messages.

```
newtype Tc a = Tc (TcEnv -> IO (Either ErrMsg a))
data TcEnv
  = TcEnv { var_env      :: Map Name Type
          , lie_env      :: IORef [Constraint]
          , untouchables :: [MetaTv]
          , ... }
```

Among other fields, the `TcEnv` environment contains a typing environment `var_env`, which is a map from term variable names to types. The field `lie_env` collects the set of `Constraint`s that arise during type inference.[9] The `Constraint` datatype holds equality and implication constraints.

### 8.1 Constraint generation

In traditional implementations, unification variables are typically *eagerly* unified to types as type inference proceeds. In contrast, the algorithm of Figure 4 first generates (lots of) constraints, and then solves them, which is much less efficient. In our implementation we choose an intermediate path, which results in much more compact generated constraints. The environment `TcEnv` is equipped with the `untouchables` field, which records the untouchable variables. As type inference proceeds we perform eager unification by side effect in the usual way, *except* that we refrain from unifying a variable $\alpha$ from the untouchable set to a type $\tau$. In that case, we defer the constraint $\alpha \sim \tau$, to be dealt with after constraint generation is finished. Hence, the unifier has signature:

```
unify :: Type -> Type -> Tc [Constraint]
```

It accepts, two types to unify, unifies them (perhaps using side effects on `MetaTv`s that are not untouchable), and returns a list of deferred equalities for variables that belong in the `untouchables` field of the environment.

How does the `untouchables` environment field get updated? Whenever we perform type inference for a pattern match clause with non-empty given equations, the main type checker:

1. extends the `untouchables` field with the unification variables of the scrutinee and the environment and the return type, as required by Figure 4,

[8] i.e. containing free occurrences of lexically scoped type variables.

[9] The name `lie_env` is folklore from type class implementations, where it stands for Local Instance Environment.

2. performs type inference for the right-hand-side of the clause and returns the deferred constraints, and
3. defers an implication constraint whose right-hand side consists of the aforementioned deferred constraints.

### 8.2 Constraint solving

During type inference we need to solve the generated constraints at two points: when the constraints for the complete program have been generated (rule (INFER), Figure 4), but also, more subtly, when we encounter a let-bound definition with no annotation (rule (LET), Figure 4) – in the latter case we must only solve the simple constraints.

***Post-constraint-generation-solving*** After constraint generation is finished, the `lie_env` field holds the set of deferred constraints. At this point we may appeal to our constraint simplifier, which is written in a lightweight error-threading monad, implemented with Haskell's `Either` datatype. By design, this monad is pure in the sense that it does not support in-place updating of `MetaTvs`.

```
solveConstraints :: [MetaTv] -> [CConstraint] ->
                    Either SimplifierError ()
solveConstraints untch cs
  = do { let (simples, propers) = splitCConstrs cs
       ; subst <- solveSimples (Unif untch) simples
       ; spropers <- substToCConstrs subst propers
       ; mapM_ solveProper spropers }
```

The function `solveConstraints` accepts a list of untouchable variables and a list of constraints (`cs`)[10] to simplify. It splits the constraints to `simples` and `propers`, solves the `simples`, applies the substitution to the propers (yielding `spropers`) and solves `spropers`. The function `solveConstraints` is the final step of type inference, and we need not return any value back – hence the `()` return type.

We will return to `solveSimples` and the meaning of the argument `(Unif untch)`, but for now let us focus on the `solvePropers` function:

```
solveProper :: CConstraint -> Either SimplifierError ()
```

The function `solveProper` accepts a single proper implication and solves it. Notice that, since the argument is a proper implication, its solution cannot affect anything in the environment and hence we may simply return `()` – the substitution in Figure 5, rule (S-PIMPL) is only there to enable clean statements and proofs of some formal properties.

We now turn to `solveSimples`, below:

```
solveSimples :: SimplifierMode -> [CConstraint] ->
                Either SimplifierError Substitution
solveSimples mode = foldM (solveSimple mode) emptySubst

solveSimple :: SimplifierMode ->
               Substitution -> CConstraint ->
               Either SimplifierError Substitution
```

The `Substitution` datatype denotes substitutions from *either* `MetaTv` or `TyVar` variables to types. Notice that `solveSimples` is defined as a *fold*, that starts-off with the empty substitution and updates it as it solves each simple constraint. In contrast, we use `mapM_` to solve each proper constraint independently in `solveConstraints`, because they cannot affect each other. The `SimplifierMode` argument to `solveSimple` stands for the mode of operation:

- If the flag is `(Unif untch)` then the returned `Substitution` binds *only* unification variables that do not appear in the list of untouchables, `untch`.[11]
- If the flag is `All` then the returned `Substitution` binds invariably skolem and unification variables. Notice that it would be *wrong* to apply this substitution to unification variables as a side-effect – for example it is definitely wrong in the context of solving the local assumptions of an implication constraint.

One reason we need the flag `All` is because `solveSimple` has to unify both skolem and unification variables when called on the *given* equalities of implication constraints. Concretely, here is the definition of `solveProper` (simplified):

```
solveProper (CImplicConstraint envs sks gs ws)
  = do { subst <- solveSimples All gs
       ; ws_cs <- substToCConstrs subst ws
       -- find untouchables from envs and subst
       ; let all_envs = ...
       ; solveConstraints all_envs ws_cs }
```

The datatype `CImplicConstraint envs sks gs ws` stands for the proper constraint $[\mathtt{envs}]\forall\mathtt{sks}.\mathtt{gs} \supset \mathtt{ws}$. Notice characteristically that the givens gs are unified using the `All` flag. Subsequently, the substitution is applied to `ws`, and the new untouchable variables are calculated (`all_envs`). Finally, `solveConstraints` is recursively called with the new list of untouchables. The function `solveConstraints` returns `()` but that's all that we need when solving the right-hand side of proper implication constraints: Any solution for a proper implication constraint could only bind "internal" variables to that constraint (i.e. not in the untouchables) and consequently does not affect any other constraint.

It is because of this recursive call to `solveConstraints` from inside of solving a proper implication that we need to pass it the list of untouchable variables — our top-level call to `solveConstraints` is with empty untouchable variables.

***Solving for let-bound definitions*** When type checking a `let`-bound definition, rule (LET) in Figure 4 requires that we solve the generated simple constraints. We already have the mechanism for doing so, via `solveSimples`. We give below a code excerpt from type checking let bindings.

```
-- type check binding
...
-- call the simplifier
; (propers,  phi) <- simplifyTc $
    do { ...
       -- cs: the constraints from type checking
       ; let (simples,propers) = splitCConstrs cs
       ; phi <- solveSimples All simples
       ; return (propers, subst) }
...
-- compute quantified vars
; let forall_tvs = ...
-- create new skolems
; sks <- ...
-- skolemizing substitution forall_tvs |-> sks
; let thetak = ...
; let phi_thetak  = thetak ‘composeSubst‘ phi
;     spropers = map (substToConstr phi_thetak) propers
-- write back constraints
; deferred  <- eqCheckSubst phi_thetak
; deferImpl env_tvs sks [] (spropers ++ deferred)
-- quantify and return type
...
```

[10] The `CConstraint` datatype is a "canonicalized" variant of `Constraint`, and we can ignore their differences below.

[11] We could in principle apply the returned substitution as side-effect but we chose to not do so in order to treat this case uniformly with the case when the flag is `All`, to be described next.

We first type check the binding and get the resulting constraints and its type. Subsequently, we call the simplifier: we split the constraints to `simples` and `propers`, we solve the `simples` (using mode `All`) and return the `propers` and the resulting substitution, `subst`. Next, we compute the variables to quantify and the skolemizing substitution $\theta_k$ of rule (LET) in Figure 4 (`thetak`).

Next, we need to extend the `lie_env` with a constraint. At this point we could in principle return the original constraint to which we have applied $\theta_k$, wrapped as a simple implication constraint, as in rule (LET). As an optimization however, we call `unify` on each binding $\nu := \tau$ in `phi_thetak`; in the common case where $\nu$ is a (touchable) unification variable $\alpha$ `unify` will update $\alpha$ in-place, otherwise it will defer the constraints. Those deferred constraints are bound to `deferred`, and finally return a simple implication constraint that contains the skolemized proper part of the original (`spropers`) and those `deferred`.

# 9. Related Work

Since GADTs have become popular there has been a flurry of papers on inference algorithms to support them in a practical programming language.

## 9.1 Fully-annotated programs

One approach is to assume that the program is fully type-annotated, i.e. each sub-expression carries explicit type information. Under this (strong) assumption, we speak of type *checking* rather than *inference*. Type checking boils down to unification which is decidable. Hence, we can conclude that type checking for GADTs is decidable. For example, consider (CH03) and (SP07).

## 9.2 Entirely unannotated programs

Type inference for unannotated programs turns out to be extremely hard. The difficulty lies in the fact that GADT pattern matches bring into scope local type assumptions (Section 2). Following the standard route of reducing type inference to constraint solving, GADTs require implication constraints to capture the inference problem precisely (SSS08).

Unification is no longer sufficient to solve such constraints. We require more complicated solving methods such as constraint abduction (Mah05) and E-unification (GNRS92). It is fairly straightward to construct examples which show that no principal solutions (and therefore no principal types) exist. We can even conclude that GADT inference is undecidable by reduction to simultaneous rigid E-unification problem which is known to be undecidable (DV95).

How do previous inference approaches tackle these problems?

Simonet and Pottier (SP07) solve the inference problem by admitting (much) richer constraints. They sidestep the problems of undecidability and lack of principal types altogether by reducing type inference to type checking. Their inference approach only accumulates (implication) constraints and refrains from solving them. As a result, implications may appear in type schemes, which is a serious complication for the poor programmer (we elaborate in Section 5.1). Furthermore, no tractable solving algorithm is known for the constraints they generate, largely because of the (absolutely necessary) use of implications.

Sulzmann *et al* (SSS08) go the other direction, by keeping constraints (in types) simple, and instead apply a very powerful (abductive) solving mechanism. To avoid undecidability, they only consider a selected set of "intuitive" solutions. However they give only an inference algorithm, and it is not clear how to give a declarative description that specifies which programs are well-typed and which are not. Furthermore their system lacks principal types.

## 9.3 Practical compromises

We conclude that tractable type inference for completely unannotated programs is impossible. It is therefore acceptable to demand a certain amount of user-provided type information. We know of two well-documented approaches:

**Régis-Gianas and Pottier** stratify type inference into two passes. The first figures out the "shape" of types involving GADTs, while the second performs more-or-less conventional type inference (PRG06). Régis-Gianas and Pottier present two different shape analysis procedures, the *Wob* and *Inst* systems. The *Wob* system has similar expressiveness and need for annotation as in (PVWW06). The *Ibis* system on the other hand has similar expressiveness as our system, with a very aggressive iterated shape analysis process. This is reminiscent of our unification of simple constraints arising potentially from far-away in the program text, prior to solving a particular proper constraint. In terms of expressiveness, the vast majority of programs typeable by our system are typeable in *Ibis* but we conjecture that there exist programs typeable in our system not typeable in *Ibis*, because unification of simple (global) constraints may be able to figure more out about the types of expressions than the preprocessing shape analysis of *Ibis*. On the other hand, *Ibis* lacks a declarative specification that does not force the programmer to understand the intricacies of shape propagation.

**Peyton Jones *et al*** require that the scrutinee of a GADT match has a "rigid" type, known to the type checker *ab initio*. A number of *ad hoc* rules describe how a type signature is propagated to control rigidity (PVWW06). Because rigidity analysis is more aggressive in our system we type many more programs than in (PVWW06), including the carefully-chosen Example 7.2 from (PRG06). On the other hand a program fails to type check in our approach if the type of a case branch is not determined by some "outer" constraint:

```
data Eq a b where { Refl :: forall a. Eq a a }

test :: forall a b. Eq a b -> Int
test x = let funny_id = \z -> case x of Refl -> z
         in funny_id 3
```

By contrast this program is typeable in (PVWW06). Arguably, though, this program *should* be rejected, because there are several incomparable types for `funny_id` (in the unrestricted system of Figure 2), including $\forall c.c \to c$ and $a \to b$.

The implementation of GHC is a slight variation that requires that the right-hand-side of a pattern match clause be typed in a rigid environment[12]. Hence, it would reject the previous example. Our system is strictly more expressive than this variation:

```
test :: forall a b. Eq a b -> Int
test x = (\z -> case x of Eq -> z) 34
```

The above program would fail to type check in GHC, as the "wobbly" variable `z` cannot be used in the right-hand-side of a pattern match clause, but in our system it would be typeable because the "outer" constraint forces `z` to get type `Int`.

In both approaches, inferred types are maximal, but *not necessarily principal* in the unrestricted natural GADT type system. The choice for a particular maximal type over others relies on the ad hoc rigidity analysis or shape pre-processing. By contrast, in our system only programs that enjoy principal types in the unrestricted type system are accepted.

[12] GHC's algorithm is described in an Appendix to the online version of the paper, available from:
`http://research.microsoft.com/people/simonpj/papers/gadt`

Moreover, in both approaches the programmer is required to understand an entirely new concept (shape or rigidity repectively), with somewhat complex and *ad hoc* rules (e.g. Figure 6 of (PRG06)). Nor is the implementation straightforward; e.g., GHC's implementation of (PVWW06) is known to be flawed in a non-trivial way.

## 10. Further work

Although we have focused exclusively on GADTs, we intend to apply our ideas in the context of Haskell, and more specifically of the Glasgow Haskell Compiler. The latter embodies numerous extensions to Haskell 98, some of which are highly relevant. Notably, a data constructor can bring into scope a local type-class constraint:

```
class Eq a where { (==) :: a -> a -> Bool }
data D a where { D1 :: Eq a => a -> D a }

h :: a -> D a -> Bool
h x (D1 y) = x==y
```

The pattern match on D1 brings the (Eq a) constraint into scope, which can be used to discharge the (Eq a) constraint that arises from the occurrence of (==). Note that D1 is not a GADT; it brings into scope no new type equalities. The same thing may happen with Haskell's implicit parameters (LLMS00).

Since type inference for Haskell already involves gathering and solving type-class constraints, the constraint-gathering approach to inference is quite natural. The above extension to Haskell generalises the idea of *local* type constraints to constraints other than equalities, and these naturally map to the same implication constraints we need for GADTs.

More ambitiously, GHC also supports *indexed type families* and type-equality constraints between them (SJCS08). So we may write

```
type family F :: * -> *
type instance F Int = Int
type instance F [a] = F a

data E a where { E1 :: (F a ~ Int) => a -> E a }
```

Here, when we match on E1 we get the local constraint that F $a \sim$ Int, which in turn gives rise to new questions for the solver (SJCS08).

Unsurprisingly, these extensions raise similar issues that we found with simple equality constraints. For example, it turns out that type classes suffer from the same lack of principal types as equality constraints (SSS06). Consider this function:

```
data T a where { MkT :: Eq a => T a }

f x y = case x of { MkT -> y==y } :: Bool
```

What type should be inferred for f? Here are two, neither of which are more general than the other:

$$\begin{array}{l} \texttt{f} :: \forall a.\texttt{T}\ a \to a \to \texttt{Bool} \\ \texttt{f} :: \forall ab.\texttt{Eq}\ b \Rightarrow \texttt{T}\ a \to b \to \texttt{Bool} \end{array}$$

Hence, the declarative specification of the type system and the inference algorithm must be extended to cope with additional kinds of constraints.

Finally, in practice, our type checker algorithm would have to be augmented with the generation of *evidence*, witnessing that the wanted constraints hold. In GHC's intermediate language, evidence for equality constraints takes the form of type equality coercions, while dictionaries are the evidence for type class constraints. We have omitted evidence handling here so as not to distract from the essence of the **OutsideIn** algorithm.

***Acknowledgements*** We are grateful to the anonymous ICFP 2009 reviewers, and to James McKinna's team for their comments.

## References

[CH03] J. Cheney and R. Hinze. First-class phantom types. TR 1901, Cornell University, 2003.

[DV95] A. Degtyarev and A. Voronkov. Simultaneous regid E-unification is undecidable. In *Proc. of CSL'95*, volume 1092 of *LNCS*, pages 178–190. Springer, 1995.

[GNRS92] J. H. Gallier, P. Narendran, S. Raatz, and W. Snyder. Theorem proving using equational matings and rigid e-unification. *J. ACM*, 39(2):377–429, 1992.

[LLMS00] J. R. Lewis, J. Launchbury, E. Meijer, and M. Shields. Implicit parameters: Dynamic scoping with static types. In *POPL*, pages 108–118, 2000.

[Mah05] M. Maher. Herbrand constraint abduction. In *Proc. of LICS'05*, pages 397–406. IEEE Comp. Soc., 2005.

[PR05] F. Pottier and D. Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

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

[PVWS07] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. of Func. Prog.*, 17:1–82, January 2007.

[PVWW06] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of ICFP'06*, pages 50–61. ACM, 2006.

[SCPD07] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Proc. of TLDI'07*. ACM, 2007.

[SJCS08] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. *SIGPLAN Not.*, 43(9):51–62, 2008.

[SP07] V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Prog. Languages Systems*, 29(1), January 2007.

[SSS06] M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Principal type inference for GHC-style multi-parameter type classes. In *Proc. of APLAS'06*, volume 4279 of *LNCS*, pages 26–43. Springer, 2006.

[SSS08] M. Sulzmann, T. Schrijvers, and P. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Report CW 507, K.U.Leuven, Belgium, 2008.