OutsideIn(X)
Modular type inference with local assumptions

24 March 2011

Dimitrios Vytiniotis
Microsoft Research

Simon Peyton Jones
Microsoft Research

Tom Schrijvers
Universiteit Gent

Martin Sulzmann
Informatik Consulting Systems AG

Abstract

Advanced type system features, such as GADTs, type classes, and type families have proven to be invaluable language extensions for ensuring data invariants and program correctness among others. Unfortunately, they pose a tough problem for type inference, because they introduce *local* type assumptions.

In this article we present a novel constraint-based type inference approach for local type assumptions. Our system, called OutsideIn(X), is parameterised over the particular underlying constraint domain X, in the same way as HM(X). This stratification allows us to use a common metatheory and inference algorithm.

Going beyond the general framework, we also give a particular constraint solver for X = type classes + GADTs + type families, a non-trivial challenge in its own right.

Contents

1 Introduction 4
2 The challenge we address 6
2.1 Modular type inference and principal types 8
2.2 The challenge of local constraints 8
2.3 The challenge of axiom schemes 9
2.4 Recovering principal types by enriching the type syntax 10
2.5 Summary 11
3 Constraint-based type systems 11
3.1 Syntax 11
3.2 Typing rules 13
3.3 Type soundness 15
3.4 Type inference, informally 16
3.5 Type inference, precisely 17
3.6 Soundness and principality of type inference 20
4 Constraint-based type systems with local assumptions 24
4.1 Data constructors with local constraints 24
4.2 `let` should not be generalized 27
4.3 The lack of principal types 34
5 Type inference with OutsideIn(X) 35
5.1 Type inference, informally 35
5.2 Overview of the OutsideIn(X) solving algorithm 36
5.3 Top-level algorithmic rules 38
5.4 Generating constraints 39
5.5 Solving constraints 40
5.6 Variations on the design 42
5.7 Soundness and principality of type inference 43
6 Incompleteness and ambiguity 46
6.1 Incompleteness due to ambiguity 46
6.2 Incompleteness due to inconsistency 47
6.3 Incompleteness of the OutsideIn(X) strategy 48
6.4 Guess-free completeness 49
6.5 Our position on incompleteness and ambiguity 50
7 Instantiating X for GADTs, type classes, and type families 51
7.1 The entailment relation 52
7.2 Solving equality constraints is tricky 53
7.3 The simplifier 54
7.4 Rewriting constraints 56
7.5 The rule SIMPLES 65
7.6 Soundness and principality 66
7.7 Termination 67
8 Implementation 69
8.1 Evidence 70
8.2 Brief sketch of the implementation 70

9 Related work 72
9.1 Constraint-based type inference 72
9.2 The special case of GADTs 73
9.3 The special case of multi-parameter type classes 75
9.4 Solving equalities involving type families 75
9.5 Let generalization for units of measure and type families 76
9.6 Ambiguity 77
9.7 Is the emphasis on principal types well-justified? 78
10 Future Work 79
References 79

1 Introduction

The Hindley-Milner type system (Damas & Milner, 1982; Milner, 1978) is a masterpiece of design. It offered a big step forward in expressiveness (parametric polymorphism) at very low cost. The cost is low in several dimensions: the type system is technically easy to describe, and a straightforward inference algorithm is both sound and complete with respect to the specification. And it does all this for programs with no type annotations at all!

Over the following thirty years, type systems have advanced rapidly, both in expressiveness and (less happily) in complexity. One particular direction in which they have advanced lies in the *type constraints* that they admit, such as type classes, implicit parameters, record constraints, subtype constraints, and non-structural equality constraints. Type inference obviously involves constraint solving, but it is natural to ask whether one can design a system that is somehow independent of the *particular* constraint system. The HM(X) system embodies precisely such an approach, by abstracting over the constraint domain "X" (Odersky *et al.*, 1999).

However HM(X) is not expressive enough to describe the systems we need. The principal difficulty is caused by so-called *local constraints*. By local constraints we mean type constraints that hold in some parts of the program but not others. Consider, for example, the following program, which uses a Generalised Algebraic Data Type (GADT), a recently-introduced and wildly popular feature of Haskell (Peyton Jones *et al.*, 2006):

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

test (T1 n) _ = n > 0
test T2     r = r
```

The pattern match on `T1` introduces a local constraint that the type of `(T1 n)` be equal to `T1 Bool` *inside the body* `n > 0`. But that constraint need not hold outside that first pattern match. In fact, the second line for `test` allows any type `T a` for its first argument.

What type should be inferred for function `test`? Alas, there are two possible most-general System F types, neither of which is an instance of the other[1]:

$$\texttt{test} \;::\; \forall a \,.\, \texttt{T}\ a \to \texttt{Bool} \to \texttt{Bool}$$
$$\texttt{test} \;::\; \forall a \,.\, \texttt{T}\ a \to a \to a$$

The second type for `test` arises from the fact that if its first argument has type `T a`, then the first branch allows the return type `Bool` to be replaced with `a`, since the local constraint that `T a` is equal to `T Bool` holds in that branch (although not elsewhere). The loss of principal types is both well-known and unavoidable (Cheney & Hinze, 2003).

[1] We write "System F" since the answer to this question varies when more or less expressive types are considered.

A variety of papers have tackled the problem of local constraints in the specific context of GADTs, by a combination of user-supplied type annotations and/or constraint-based inference (Peyton Jones *et al.*, 2006; Pottier & Régis-Gianas, 2006; Simonet & Pottier, 2007; Sulzmann *et al.*, 2008). Unfortunately, none of these approaches is satisfying, even to their originators, for a variety of reasons (Section 9). Simonet and Pottier give an excellent summary in the closing sentences of their paper (Simonet & Pottier, 2007):

> *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 (Peyton Jones* et al.*, 2006), recent work by Pottier and Régis-Gianas (2006) 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.*

Our system produces simpler constraints than the constraints Simonet and Pottier use, and does not rely on an *ad hoc* local shape inference phase. Furthermore, it does this not only for the specific case of GADTs, but for the general case of an arbitrary constraint system in the style of HM(X). Specifically, we make the following contributions:

- We describe a constraint-based type system that, like HM(X), is parameterised over the underlying constraint system X (Section 4), and includes:
  - Data constructors with existential type variables.
  - Data constructors that introduce local constraints, of which GADTs are a special case.
  - Type signatures on local `let`-bound definitions.
  - Top-level axiom schemes (such as Haskell's `instance` declarations).

  These extensions offer substantially improved expressiveness, but at significant cost to the specification and implementation. Local constraints from data constructors or signatures are certainly not part of HM(X); existential variables can sometimes be accommodated by the techniques found in some presentations (Pottier & Rémy, 2005), and top-level axiom schemes is only part of Mark Jones' qualified types (Jones, 1992).
- While developing our type system we show a surprising result: while sound and complete implicit generalisation for local `let` bindings is straightforward in Hindley-Milner, it becomes prohibitively complicated when combined with a rich constraint system that includes local assumptions (Section 4.2). Happily, we demonstrate that local generalisation is almost never used, and when it absolutely has to be used, a local type signature makes these complications go away. Thus motivated, albeit controversially, we propose to simplify the language by removing implicit generalisation of local `let` bindings.
- We give an inference algorithm, OutsideIn(X), that is stratified into (a) an inference engine that is independent of the constraint system X, and (b) a constraint solver for X itself (Section 5). We show that our approach is not *ad-hoc*: any program accepted by our algorithm can be typed with a principal type in the simple natural constraint-based type system. Previous

work (Peyton Jones *et al.*, 2006; Pottier & Régis-Gianas, 2006) only infers principal types with respect to specialized type systems, but not with respect to the natural constraint-based type system.

- A particularly useful, but particularly delicate, class of constraints are *type-equality constraints*, including those introduced by GADTs, and by type-level functions (*type families* in the Haskell jargon). Section 7 gives a concrete instantiation of X with the constraints arising from these features, as well as with the more traditional Haskell type class constraints. Our concrete solver subsumes and simplifies our previous work on solving constraints involving type families (Schrijvers *et al.*, 2008a), and is now part of the GHC implementation.

This paper draws together, in a single uniform framework, the results of a multi-year research project, documented in several earlier papers (Peyton Jones *et al.*, 2006; Schrijvers *et al.*, 2007; Peyton Jones *et al.*, 2004; Schrijvers *et al.*, 2008a; Schrijvers *et al.*, 2009; Vytiniotis *et al.*, 2010). By taking the broader perspective of abstracting over the constraint domain "X" we hope to bring out the core challenges in higher relief, and contribute towards building a single generic solution rather than a multitude of *ad-hoc* compromises.

The principal shortcoming of our system is shared by every other paper on the subject: an unsatisfactory account of *ambiguity*, a notion first characterized by Jones (1993). We discuss the issue, along with a detailed account of incompleteness, in Section 6. There is also a good deal of related work, which we describe in Section 9.

2 The challenge we address

We begin by briefly reviewing part of the type-system landscape, to identify the problems that we tackle.

The vanilla Hindley-Milner system has just one form of type constraint, namely the equality of two types, which we write $\tau_1 \sim \tau_2$. For example, the application of a function of type $\tau_1 \to \tau_2$ to an argument of type τ_3 gives rise to an equality constraint $\tau_1 \sim \tau_3$. These equality constraints are *structural* and hence can be solved easily by unification, with unique most general solutions. However, subsequent developments have added many new forms of type constraints:

- Haskell's type classes add *type-class constraints* (Jones, 1992; Wadler & Blott, 1989; Hall *et al.*, 1996). For example, the constraint `Eq` τ requires that the type τ be an instance of the class `Eq`. Haskell also allows types that quantify over constraints (often called *qualified types* in the HM(X) literature). For example the `member` function has type

```
member :: Eq a => a -> [a] -> Bool
```

  which says that `member` may be called at any type τ, but that the constraint `Eq` τ must be satisfied at the call site.

- An early extension to the original definition of Haskell 98 was to allow *multi-parameter* type classes, which Mark Jones subsequently extended with *functional dependencies* (Jones, 2000). This pair of features turned out to be tremendously useful in practice, and gave rise to a whole cottage industry of programming techniques that amount to performing arbitrary computation at the type level. We omit the details here but the underlying idea was that the conjunction of two class constraints $C\ \tau\ \upsilon_1$ and $C\ \tau\ \upsilon_2$ gives rise to an additional equality constraint $\upsilon_1 \sim \upsilon_2$.
- Generalised Algebraic Data Types (GADTs) added a substantial new twist to equality constraints by supporting *local* equalities introduced by pattern matching (Xi *et al.*, 2003; Peyton Jones *et al.*, 2006). User type signatures with constrained types have a similar effect, also introducing local assumptions. We will discuss GADTs further in Section 2.
- Kennedy's thesis (Kennedy, 1996) describes how to accommodate *units of measure* in the type system so that one may write

  ```
  calcDistance :: num (m/s) -> num s -> num m
  calcDistance speed time = speed * time
  ```

  thereby ensuring that the first argument is a speed in metres/second, and similarly for the other argument and result. The system supports polymorphism, for example

  ```
  (*) :: num u1 -> num u2 -> num (u1*u2)
  ```

  There is, necessarily, a non-structural notion of type equality. For example, to type check the definition of `calcDistance` the type engine must reason that `(m/s)*s` $\sim$ `m`. This is an ordinary equality constraint, but there is now a non-standard equality theory so the constraint solver becomes more complicated.
- More recently, inspired by object-oriented languages, we have proposed and implemented a notion of *associated types* in Haskell (Chakravarty *et al.*, 2005b; Chakravarty *et al.*, 2005a). The core feature is that of a *type family* (Kiselyov *et al.*, 2010). For instance the user may declare type family axioms:

  ```
  type family F :: * -> *

  type instance F [a]  = F a
  type instance F Bool = Int
  ```

  In this example F is a type family with two defining axioms, $F\ [a] \sim F\ a$ and F `Bool` $\sim$ `Int`. This means that any expression of type F [`Bool`] can be considered as an expression of type F `Bool` (using the first axiom), which in turn can be considered as having type `Int` (using the second axiom). Hence, like in the case of units of measure, equalities involving type families are also non-structural.

Type inference for HM(X) is tractable: it boils down to constraint solving for existentially-quantified conjunctions of primitive constraints. However, the type system features discussed above go beyond HM(X), in two particular ways. First,

GADTs bring into scope local type constraints, and existentially-quantified variables (Section 2.2). Second, we must allow top-level *axiom schemes*, such as Haskell's `instance` declarations (Section 2.3). In this paper we address modular type inference with principal types for the aforementioned type system features and beyond. In the following sections we explain the problem in more detail, but we begin with a brief discussion of principal types to set the scene.

2.1 Modular type inference and principal types

Consider this expression, which defines `f` without a type signature, expecting the type inference engine to figure out a suitable type for `f`:

```
let f x = <rhs>
in <body>
```

It is very desirable that type inference should be *modular*; that is, it can infer a type for `f` by looking only at `f`'s *definition*, and not at its *uses* in `<body>`. After all, `<body>` might be very large or, in a system supporting separate compilation, `f`'s definition might be in a library, while its call sites might be in other modules.

In a language supporting polymorphism it is common for `f` to have many types; for example, `reverse` has types [`Int`] → [`Int`] and [`Bool`] → [`Bool`]. For modular type inference to work, it vital for `f` to have a unique *principal type*, that is more general than all its other types. For example `reverse` $:: \forall a\,.\,[a] \to [a]$.

When the programmer supplies an explicit type signature, the issue does not arise: we should simply check that the function indeed has the specified type, and use that type at each call site. However, for modular type inference, we seek a type system for which an un-annotated definition has a unique principal type – or else is rejected altogether. The latter is acceptable, because the programmer can always resolve the ambiguity by adding a type signature.

2.2 The challenge of local constraints

Generalized Algebraic Data Types (GADTs) have proved extremely popular with programmers, but they present the type inference engine with tricky choices. Notably, as mentioned in the introduction, functions involving GADTs may lack a principal type. Recall the example:

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

test (T1 n) _ = n > 0
test T2     r = r
```

One can see that `test` has two possible most-general System F types, neither of which is an instance of the other:

$$\texttt{test} :: \forall a\,.\,\texttt{T}\ a \to \texttt{Bool} \to \texttt{Bool}$$
$$\texttt{test} :: \forall a\,.\,\texttt{T}\ a \to a \to a$$

Since `test` has no principal type we argue that rather than making an arbitrary choice, the type inference engine should reject the program. The programmer can fix the problem by specifying the desired type with an explicit type signature, such as:

```
test :: T a -> Bool -> Bool
```

But exactly *which* GADT programs should be rejected? For example, consider `test2`:

```
test2 (T1 n) _ = n > 0
test2 T2     r = not r
```

Since `T2` is an ordinary (non-GADT) data constructor, the *only* possible type of `r` is `Bool`, so the programmer might be confused at being required to say so.

2.3 The challenge of axiom schemes

A further challenge for type inference are top-level universally quantified constraints, which we refer to as axiom schemes. Early work by Faxén (Faxén, 2003) already shows how the interaction between type signatures and axiom schemes can lead to the loss of principal types. Here is another example of the same phenomenon, taken from (Sulzmann *et al.*, 2006a):

```
class Foo a b where foo :: a -> b -> Int
instance Foo Int b
instance Foo a b => Foo [a] b

g y = let h :: forall c. c -> Int
          h x = foo y x
      in h True
```

In this example the two instance declarations give rise to two axiom schemes. The instance `Foo Int b` provides the constraint `Foo Int b` for *any* possible `b`. Similarly, the second instance provides the constraint `Foo [a] b` for *any* possible `a` and `b`, as long as we can show `Foo a b`.

Suppose now that `y` has type `[Int]`. Then, the inner expression `foo y x` gives rise to the constraint `Foo [Int] c` This constraint can be reduced via the above instance declarations. and thus the program type checks. We can generalize this example and conclude that function `g` can be given the infinite set of types

```
g :: Int → Int
g :: [Int] → Int
g :: [[Int]] → Int
g :: ...
```

but there is no most general Haskell type for `g`.

Here is a similar example, this time showing the interaction of type families with existential data types (Läufer & Odersky, 1994), yet another extension of vanilla HM(X).

```
type family FB :: * -> * -> *
type instance FB Int b = Bool
type instance FB [a] b = FB a b

data Bar a where
   K :: a -> b -> Bar a

h (K x y) = not (fb x y)    -- Assume  fb :: a -> b -> FB a b
                            --         not :: Bool -> Bool
```

To type check the body of `h` we must be able to determine some type variable α_x (for the type of `x`) such that, for any b (the type of `y`), the result type of the call to `fb` is compatible with `not`; that is: `FB` α_x $b \sim$ `Bool`. In combination with the above type instances, function `h` can be given the infinite set of types

```
h  ::  Bar Int → Bool
h  ::  Bar [Int] → Bool
h  ::  Bar [[Int]] → Bool
h  ::  ...
```

Unsurprisingly, axiom schemes combined with local assumptions are no better, even in the absence of GADTs. Consider the following example, using only type classes.

```
class C a
class B a b where op :: a -> b
instance C a => B a [a]

data R a where
  MkR :: C a => a -> T a

k (MkR x) = op x
```

The function `k` has both these (incomparable) types:

$$\texttt{k} \;::\; \forall ab \,.\, \texttt{B}\ a\ b \Rightarrow \texttt{R}\ a \rightarrow b$$
$$\texttt{k} \;::\; \forall a \,.\, \texttt{R}\ a \rightarrow [a]$$

The first is straightforward; it makes no use of the local `(C a)` constraint in `MkR`'s type. The second fixes the return type to be `[a]`, but in return it can make use of the local `(C a)` constraint to discharge the constraint arising from the call of `op`.

2.4 Recovering principal types by enriching the type syntax

There is a well-known recipe for recovering principal types ((Simonet & Pottier, 2007)): enrich the language of types to allow quantification over constraint schemes and implications. To be concrete, here are the principal types of the problematic

functions in Sections 2.2 and 2.3:

$$\begin{array}{lll} \texttt{test} & :: & \forall ab\ .\ (a \sim \texttt{Bool} \supset b \sim \texttt{Bool}) \Rightarrow \texttt{T}\ a \rightarrow b \rightarrow b \\ \texttt{g} & :: & \forall b\ .\ (\forall c\ .\ \texttt{Foo}\ b\ c) \Rightarrow b \rightarrow \texttt{Int} \\ \texttt{h} & :: & \forall a\ .\ (\forall b\ .\ (\texttt{FB}\ a\ b \sim \texttt{Bool})) \Rightarrow \texttt{Bar}\ a \rightarrow \texttt{Bool} \\ \texttt{k} & :: & \forall ab\ .\ (\texttt{C}\ a \supset \texttt{B}\ a\ b) \Rightarrow \texttt{R}\ a \rightarrow b \end{array}$$

We have ourselves flirted with quantifying over (implication) constraint schemes, but we will argue in 4.2 that this is not practical.

2.5 Summary

The message of this section is simple: by moving beyond HM(X) in terms of expressiveness without enriching the type syntax, many programs no longer have principal types. Ideally, if a function does not have a principal type, we would like to reject its definition. The challenge is to identify which definitions should be accepted and which should be rejected. Moreover, we want to do this *generically*, for any X. In the rest of this paper we explore a type system and type inference algorithm that addresses these challenges.

3 Constraint-based type systems

To formally describe the problem and our solution in the rest of the paper, we introduce notation and review *constraint-based* type inference (Pottier & Rémy, 2005; Odersky *et al.*, 1999; Jones, 1992). In constraint-based type inference, constraints appear as part of the type system specification, and the implementation works by generating and solving those constraints.

Although the material in this section is quite standard, our base system additionally supports `case` expressions, and top-level type signatures. Moreover, our definitions and metatheory are carefully engineered so that they will later carry over to the extensions outlined in the introduction.

Since our type system supports type signatures, there is one more difference. Earlier work (Pottier & Rémy, 2005; Odersky *et al.*, 1999; Jones, 1992) only considers solving sets of primitive constraints, which we refer to as "wanted" constraints. The presence of type signatures forces us to consider a more general setting, where we solve wanted constraints with respect to a set of "given" constraint assumptions. For example, given

```
palin :: Eq a => [a] -> Bool
palin xs = xs == reverse xs
```

Here the wanted constraint arising from the use of `(==)` is (`Eq` $[a]$); it must be solved from the given assumption (`Eq` a).

3.1 Syntax

Figure 1 gives a Haskell-like syntax for a language that includes constraints. Programs simply consist of a sequence of top-level bindings that may or may not be

Term variables		$\in$	x, y, z, f, g, h
Type variables		$\in$	a, b, c
Data constructors		$\in$	K
	ν	::=	$K \mid x$
Programs	*prog*	::=	$\epsilon \mid f = e, \mathit{prog} \mid f :: \sigma = e, \mathit{prog}$
Expressions	e	::=	$\nu \mid \lambda x \,.\, e \mid e_1\ e_2 \mid \texttt{case}\ e\ \texttt{of}\{\overline{K\,\overline{x} \rightarrow e}\}$
Type schemes	σ	::=	$\forall \overline{a} \,.\, Q \Rightarrow \tau$
Constraints	Q	::=	$\epsilon \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2 \mid \ldots$
Monotypes	τ, υ	::=	$tv \mid \texttt{Int} \mid \texttt{Bool} \mid [\tau] \mid \texttt{T}\ \overline{\tau} \mid \ldots$
	tv	::=	a
Type environments	Γ	::=	$\epsilon \mid (\nu{:}\sigma), \Gamma$
Free type variables	$\mathit{ftv}(\cdot)$		

Γ_0 : Types of vanilla data constructors

$K \quad : \quad \forall \overline{a} \,.\, \overline{\upsilon} \rightarrow \texttt{T}\ \overline{a}$

Top-level axiom schemes

$\mathcal{Q} \quad ::= \quad \epsilon \mid \mathcal{Q} \wedge \mathcal{Q} \mid \forall \overline{a} \,.\, Q \Rightarrow Q$

Fig. 1: Syntax

accompanied with type signatures. The expressions that may be bound consist of the λ-calculus, together with `case` expressions to perform pattern matching. It turns out that local `let` declarations are a tricky point, so we omit them altogether for now, returning to them in Section 4.2. As a convention, we use term variables f, g, h as identifiers for those top-level bindings, and x, y, z for identifiers bound by λ-abstractions and `case` patterns.

Data constructors in Haskell or ML are introduced by algebraic data type declarations. Instead of giving the syntax of such declarations, we simply assume an initial type environment Γ_0, populated by data type declarations, that gives the type of each data constructor. Each such data constructor K has a type of form

$$K :: \forall \overline{a} \,.\, \overline{\upsilon} \rightarrow \texttt{T}\ \overline{a}$$

where $\overline{a}$ are the universally quantified variables of the constructor – the ones appearing in the return type $\texttt{T}\ \overline{a}$. Notice that, for now, data constructors have unconstrained types, unlike the GADT constructors of Section 2.2.

The syntax of types and constraints also appears in Figure 1. The syntax of types is standard, and we use meta variables τ and υ to denote types. We use tv to denote type variables $a, b, \ldots$ (we will later on extend tv to include unification variables introduced by an algorithm). Polymorphic (quantified) types σ may include constraints, and are of the form $\forall \overline{a} \,.\, Q \Rightarrow \tau$.

Constraints Q include type equalities $\tau_1 \sim \tau_2$ and conjunctions $Q_1 \wedge Q_2$. We treat conjunction $Q_1 \wedge Q_2$ as an associative and commutative operator, as is conventional. By design, we leave the syntax of constraints open; hence the "...". This is the "X" part of HM(X) and OutsideIn(X), to be presented later. Types are similarly open (hence "..." in τ), because a constraint system may involve new type forms. For example, dimensional units involve types such as `Float (m/s)`.

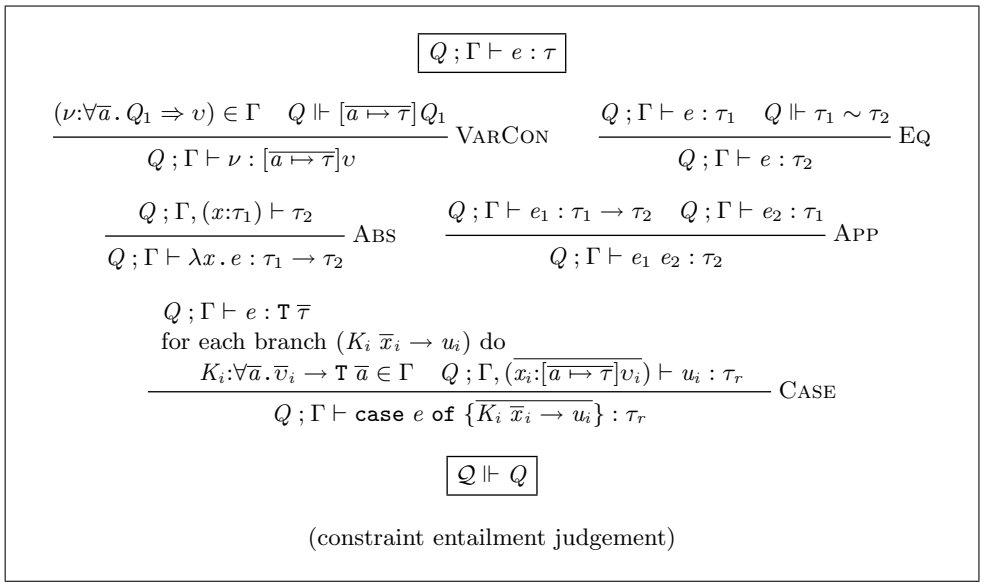

Fig. 2: Vanilla constraint-based type system

A term is typed relative to a set of top-level *axiom schemes* $\mathcal{Q}$, whose syntax is in Figure 1. In Haskell type classes, for example, an instance declaration corresponds to an axiom scheme:

```
instance Eq a => Eq [a] where { ... }
```

As we mentioned in Section 2.4, Notice that types $\forall \overline{a} \,.\, Q \Rightarrow \tau$ are quantified only over *flat* constraints Q and not over constraint *schemes* $\mathcal{Q}$. We noted in Section 2.4 that the latter choice would be more expressive but, as we explain in Section 4.2.1, we do not believe it is feasible in practice. On the other hand, top level axiom schemes are essential to handle Haskell, so we are forced into stratifying the system.

3.2 Typing rules

In a constraint-based type system, the main typing relation takes the form

$$Q ; \Gamma \vdash e : \tau$$

meaning "in a context where the (flat) constraint Q is available, and in a type environment Γ, the term e has type τ". For example, here is a valid judgement:

$$(a \sim \mathtt{Bool}) ; (\mathtt{x}{:}a, \mathtt{not} : \mathtt{Bool} \to \mathtt{Bool}) \vdash \mathtt{not}\ \mathtt{x} : \mathtt{Bool}$$

The judgement only holds because of the availability of the constraint $a \sim \mathtt{Bool}$. Since $\mathtt{x} : a$ and $a \sim \mathtt{Bool}$ we have that $\mathtt{x} : \mathtt{Bool}$, and hence `x` is acceptable as an argument to `not`.

Figure 2 shows a vanilla constraint-based type system for the language of Figure 1. The Figure has two particularly interesting rules.

- Rule VARCON, is used to instantiate the potentially constrained type of a

Reflexivity	$\mathcal{Q} \wedge Q \Vdash Q$	(R1)
Transitivity	$\mathcal{Q} \wedge Q_1 \Vdash Q_2$ and $\mathcal{Q} \wedge Q_2 \Vdash Q_3$ implies $\mathcal{Q} \wedge Q_1 \Vdash Q_3$	(R2)
Substitution	$\mathcal{Q} \Vdash Q_2$ implies $\theta\mathcal{Q} \Vdash \theta Q_2$ where θ is a type substitution	(R3)
Type eq. reflexivity	$\mathcal{Q} \Vdash \tau \sim \tau$	(R4)
Type eq. symmetry	$\mathcal{Q} \Vdash \tau_1 \sim \tau_2$ implies $\mathcal{Q} \Vdash \tau_2 \sim \tau_1$	(R5)
Type eq. transitivity	$\mathcal{Q} \Vdash \tau_1 \sim \tau_2$ and $\mathcal{Q} \Vdash \tau_2 \sim \tau_3$ implies $\mathcal{Q} \Vdash \tau_1 \sim \tau_3$	(R6)
Conjunctions	$\mathcal{Q} \Vdash Q_1$ and $\mathcal{Q} \Vdash Q_2$ implies $\mathcal{Q} \Vdash Q_1 \wedge Q_2$	(R7)
Substitutivity	$\mathcal{Q} \Vdash \tau_1 \sim \tau_2$ implies $\mathcal{Q} \Vdash [a \mapsto \tau_1]\tau \sim [a \mapsto \tau_2]\tau$	(R8)

Fig. 3: Entailment requirements

term variable or constructor. The constraint arising from the instantiation has to be entailed by the available constraint, written $Q \Vdash [\overline{a \mapsto \tau}]Q_1$. For example, if `(==)` : $\forall a\,.\,$`Eq` $a \Rightarrow a \to a \to$ `Bool`, then rule VarCon can be used to instantiate `(==)` to have type `Int` $\to$ `Int` $\to$ `Bool` if $Q =$ `Eq Int`.

- The second interesting rule is Eq, where the entailment "meets" the types of our language. If an expression has type τ_1 and the entailment relation can be used to deduce that $\tau_1 \sim \tau_2$ then we may conclude that the expression has type τ_2.

The judgement $\mathcal{Q} \Vdash Q$ is a *constraint entailment* relation, and should be read as: from the axiom schemes in $\mathcal{Q}$ we can deduce Q. (In Figure 2 we only use flat constraints on the left of the $\Vdash$, but we will use the more general form shortly, in Figure 4.) We leave the details of entailment deliberately unspecified, because it is a parameter of the type system, but we will see in Section 7 a concrete instantiation of such a relation. As a notation convenience, we write

$$\mathcal{Q} \Vdash Q_1 \leftrightarrow Q_2 \quad \text{iff} \quad \mathcal{Q} \wedge Q_1 \Vdash Q_2 \text{ and } \mathcal{Q} \wedge Q_2 \Vdash Q_1$$

Although we do not give a precise definition of entailment here, in Figure 3 we postulate certain properties, sufficient to establish that we can come up with a sound implementation of type inference which moreover infers principal types. Conditions R1 and R2 state that the entailment relation is reflexive and transitive. Condition R3 ensures that entailment is preserved under substitution. Conditions R4, R5 and R6 ensure that the provable type equality is an equivalence relation, and R7 asserts that conjunctions can be proved by proving each of the conjuncts. Condition R8 asserts that provably equal types can be substituted in other types, preserving equivalence. There is nothing surprising in the conditions mentioned in Figure 3, for example Jones identifies similar conditions in his thesis (Jones, 1992), except for the conditions related to type equalities.

Pattern matching is given with rule Case and is also straightforward since the types of data constructors K do not include any constraints (as opposed to the constrained type of the `T1` constructor from the previous section), and hence pattern matching does not introduce any local assumptions.

We now specify a judgement for well-typed *programs*,

$$\mathcal{Q}\,;\Gamma \vdash \mathit{prog}$$

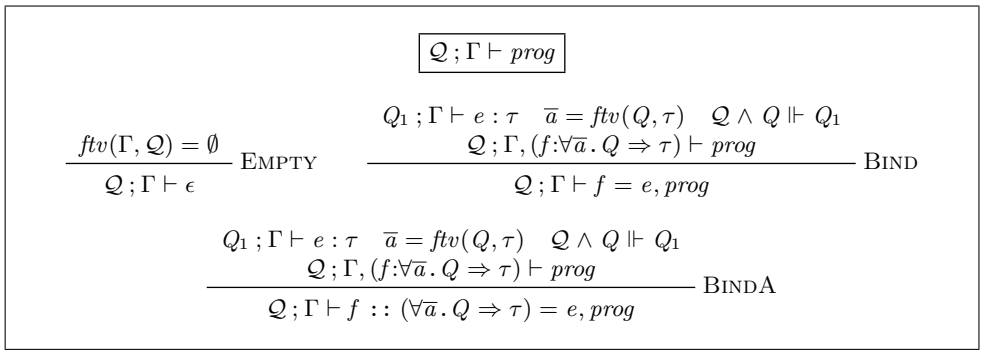

Fig. 4: Well-typed programs

in Figure 4. In a constraint-based type system users may declare top-level universally quantified constraints, or *axiom schemes*, denoted with $\mathcal{Q}$ (Figure 1). These may include, for example, type class or type family instance declarations. The syntax of axiom schemes $\mathcal{Q}$, given in Figure 1, includes universally quantified constraints of the form $\forall \overline{a}\,.\,Q_1 \Rightarrow Q_2$, where Q_1 is allowed to be ϵ, and $\overline{a}$ are the free variables of Q_1 and Q_2. Ordinary Q-constraints can be viewed as a degenerate form of $\mathcal{Q}$ constraints. As an example, here is an axiom arising from an instance declaration:

$$\forall a\,.\,\texttt{Eq}\ a \Rightarrow \texttt{Eq}\ [a]$$

Another one that binds no quantified variables and where Q_1 is empty is simply `Eq Bool`.

The judgement $\mathcal{Q}\,;\Gamma \vdash \mathit{prog}$ (Figure 4) can be read thus: "in the top-level axiom scheme environment $\mathcal{Q}$ and environment Γ, *prog* is a well-typed program".

Rule BindA deals with a top-level binding with a (closed) user-specified type annotation. If the constraint required to type check e is Q_1, then that constraint must follow from (be entailed by) the top-level axiom set $\mathcal{Q}$ and the constraint Q introduced by the type signature.

In the case of an unannotated top-level binding, rule Bind, we appeal to $Q_1\,;\Gamma \vdash e : \tau$ to determine some constraint Q_1 which is required to make e typeable with type τ in Γ. Next, we may allow the possibility of quantifying over a *simplified* version of Q_1, namely Q. This is done with the condition $\mathcal{Q} \wedge Q \Vdash Q_1$. Intuitively, Q is the "extra information", not deducible from $\mathcal{Q}$, that is needed to show the required constraint Q_1. We may then quantify over the free variables of Q and τ, and type check the rest of the program *prog*, binding f to type $\forall \overline{a}\,.\,Q \Rightarrow \tau$.

3.3 Type soundness

Does this type system obey the mantra that "well typed programs do not go wrong"? Yes, type safety typically follows under reasonable additional consistency assumptions from the constraint theory.

Definition 3.1 (Top-level consistency) *An axiom scheme $\mathcal{Q}$ is consistent iff it*

satisfies the following: whenever we have $\mathcal{Q} \Vdash \mathtt{T}_1\ \overline{\tau}_1 \sim \mathtt{T}_2\ \overline{\tau}_2$ *it is the case that* $\mathtt{T}_1 = \mathtt{T}_2$ *and* $\mathcal{Q} \Vdash \overline{\tau}_1 \sim \overline{\tau}_2$.

After all, if $\mathcal{Q}$ contained the assumption `Int` $\sim$ `Bool`, it would be unreasonable to expect a "well-typed" program not to crash. We refrain from discussing type soundness in this paper, but we urge the reader to consult the literature on HM(X) (Sulzmann, 2000; Skalka & Pottier, 2002; Pottier & Rémy, 2005) for details.

3.4 Type inference, informally

Type inference for type systems involving constraints is conventionally carried out in two stages: first *generate constraints* from the program text, and then *solve the constraints* ignoring the program text (Pottier & Rémy, 2005). The generated constraints involve *unification variables*, which stand for as-yet-unknown types, and for which we use letters $\alpha, \beta, \gamma, \ldots$. Solving the constraints produces a *substitution*, θ, that assigns a type to each unification variable. 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:

$$\mathtt{MkP} : \forall ab\,.\, a \to b \to \mathtt{Pair}\ a\ b$$

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

α	type of the entire right-hand side
β	type of `x`
γ_1, γ_2	instantiate a, b respectively, when instantiating the call of `MkP`

From the text we can generate the following equalities:

$\beta \sim \gamma_1$	First argument of `MkP`
`Bool` $\sim \gamma_2$	Second argument of `MkP`
$\alpha \sim$ `Pair` $\gamma_1\ \gamma_2$	Result of `MkP`

These constraints can be solved by unification, yielding the substitution $\theta = [\alpha \mapsto \mathtt{Pair}\ \beta\ \mathtt{Bool}, \gamma_2 \mapsto \mathtt{Bool}, \gamma_1 \mapsto \beta]$. This substitution constitutes a "solution", because under that substitution the constraints are all of form $\tau \sim \tau$.

This two-step approach is very attractive:

- The syntax of a real programming language is large, so the constraint generation code has many cases. But each case is straightforward, and adding a new case (if the language is extended) is easy.
- The syntax of constraints is small — certainly much smaller than that of the programming language. Solving the constraints may be difficult, but at least the language is small and stable.

Unification variables	$\alpha, \beta, \gamma, \ldots$		
Unifiers	θ, φ	$::=$	$[\overline{\alpha \mapsto \tau}]$
Unification or rigid (skolem) variables	tv	$::=$	$\alpha \mid a$
Algorithm-generated constraints	C	$::=$	Q
Free unification variables	$fuv(\cdot)$		
Convert to Q-constraint	$\mathbf{simple}[Q]$	$=$	Q

Fig. 5: Syntax extensions for the algorithm

3.5 Type inference, precisely

We now make precise our informal account of type inference. We first extend the syntax of Figure 1 in Figure 5. Type variables tv now include rigid (skolem) variables $a, b, \ldots$ as before, but also unification variables $\alpha, \beta, \ldots$. We use the letters θ, φ to denote idempotent substitutions whose domain includes *only* unification variables; these substitutions are called *unifiers* in the type inference jargon.

The constraints that arise during the operation of the algorithm will be denoted with C. Later on, our inference algorithm will (unlike more traditional presentations of HM(X)) gather and simplify constraints C that are somewhat richer than the constraints Q that are allowed to appear in the type system and types. For now, however, the constraints C generated by the algorithm have the same syntax as type constraints Q. The function $\mathbf{simple}[\cdot]$ accepts a C-constraint and gives us back a Q-constraint. As Figure 5 shows, $\mathbf{simple}[\cdot]$ is defined – for now – to just be the identity.

As mentioned in Section 3.4, type inference proceeds in two steps:

- *Generate constraints* with the judgement:

$$\Gamma \vdash\!\blacktriangleright e : \tau \rightsquigarrow C$$

  which can be read: "in the environment Γ we may infer type τ for the expression e and generate constraint C" (Section 3.5.1).
- *Solve constraints* for each top-level binding separately, using a simplifier for the constraint entailment relation $\overset{simp}{\vdash\!\blacktriangleright}$ (Section 3.5.2).

The two are combined by the top-level judgement

$$\mathcal{Q} \,;\, \Gamma \vdash\!\blacktriangleright \mathit{prog}$$

which invokes constraint generation and solving, to check that in a closed, top-level set of axiom schemes and a closed environment Γ the program *prog* is well typed (Section 3.5.3). We remark that type annotations are closed and hence contain no unification variables.

3.5.1 Generating constraints

Constraint generation for the language of Figure 1 is given in Figure 6, where τ and C should be viewed as outputs. Rule VarCon instantiates the polymorphic

$$\boxed{\Gamma \vdash\!\blacktriangleright e : \tau \rightsquigarrow C}$$

$$\frac{\overline{\alpha}\ \text{fresh} \quad (\nu{:}\forall \overline{a}\,.\,Q_1 \Rightarrow \tau_1) \in \Gamma}{\Gamma \vdash\!\blacktriangleright \nu : [\overline{a \mapsto \alpha}]\tau_1 \rightsquigarrow [\overline{a \mapsto \alpha}]Q_1}\ \textsc{VarCon}$$

$$\frac{\Gamma \vdash\!\blacktriangleright e_1 : \tau_1 \rightsquigarrow C_1 \quad \Gamma \vdash\!\blacktriangleright e_2 : \tau_2 \rightsquigarrow C_2 \quad \alpha\ \text{fresh}}{\Gamma \vdash\!\blacktriangleright e_1\ e_2 : \alpha \rightsquigarrow C_1 \wedge C_2 \wedge (\tau_1 \sim (\tau_2 \to \alpha))}\ \textsc{App}$$

$$\frac{\alpha\ \text{fresh} \quad \Gamma, (x{:}\alpha) \vdash\!\blacktriangleright e : \tau \rightsquigarrow C}{\Gamma \vdash\!\blacktriangleright \lambda x\,.\,e : \alpha \to \tau \rightsquigarrow C}\ \textsc{Abs}$$

$$\frac{\begin{array}{l}\Gamma \vdash\!\blacktriangleright e : \tau \rightsquigarrow C \quad \beta, \overline{\gamma}\ \text{fresh} \quad C' = (\mathtt{T}\ \overline{\gamma} \sim \tau) \wedge C \\ \text{for each } K_i\ \overline{x}_i \to e_i \text{ do} \\ \quad K_i{:}\forall \overline{a}\,.\,\overline{v}_i \to \mathtt{T}\ \overline{a} \in \Gamma \quad \Gamma, (\overline{x_i{:}[\overline{a \mapsto \gamma}]v_i}) \vdash\!\blacktriangleright e_i : \tau_i \rightsquigarrow C_i \\ \quad C_i' = C_i \wedge \tau_i \sim \beta\end{array}}{\Gamma \vdash\!\blacktriangleright \mathtt{case}\ e\ \mathtt{of}\ \{\overline{K_i\ \overline{x}_i \to e_i}\} : \beta \rightsquigarrow C' \wedge (\bigwedge C_i')}\ \textsc{Case}$$

Fig. 6: Constraint generation

type of a variable or constructor with fresh unification variables and introduces the instantiated constraint of that type. Rule App generates a fresh unification variable for the return type of the application. Rule Abs is straightforward. Rule Case deals with pattern matching. After inferring a constraint C and a type τ for the scrutinee of the case expression, e, we check each branch of the case expression independently. In every branch, we instantiate the universal variables of the constructor K_i to freshly picked unification variables $\overline{\gamma}$ (those are picked once for all possible branches since they have to be the same).

3.5.2 Solving constraints

The inference algorithm relies on a constraint solver (or, rather, simplifier). The constraint simplifier is specific to the particular constraint system X, so all we can give here is the *form* of the constraint-simplifier judgement; its *implementation* is specific to X. The judgement takes the following form:

$$\mathcal{Q}\ ;\ Q_{given} \overset{simp}{\vdash\!\blacktriangleright} Q_{wanted} \rightsquigarrow Q_{residual}\ ;\ \theta$$

It takes as input an axiom scheme set $\mathcal{Q}$ and some constraints Q_{given} that may be available, and tries to simplify the *wanted* constraints Q_{wanted} producing residual constraints that could not be simplified further (such as unsolved class constraints that may need to be quantified over) and a substitution θ that maps unification variables to types.

For example, if $\mathcal{Q}$ includes the scheme $(\forall a\ .\ \mathtt{Eq}\ a \Rightarrow \mathtt{Eq}\ [a])$, a constraint simplifier for Haskell type classes may give

$$\mathcal{Q}\ ;\ \epsilon \overset{simp}{\vdash\!\blacktriangleright} \mathtt{Eq}\ \alpha \wedge [\beta] \sim \alpha \rightsquigarrow \mathtt{Eq}\ \beta\ ;\ [\alpha \mapsto [\beta]]$$

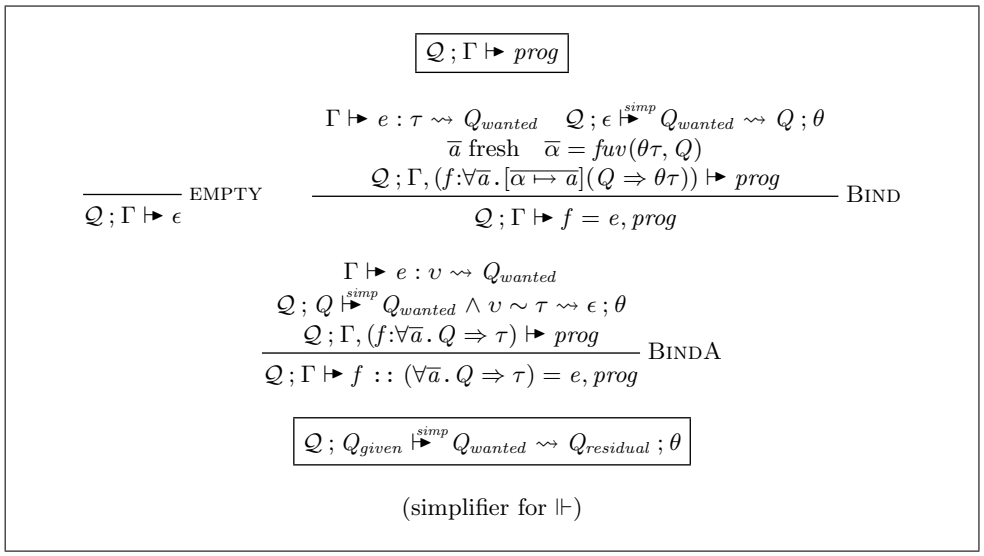

Fig. 7: Top-level algorithmic rules

Naturally, the constraint simplifier must satisfy certain properties (Figure 8), which we explore in Section 3.6.

3.5.3 The top-level inference algorithm

Now we are ready to give the top-level inference algorithm, in Figure 7, which performs type inference on whole programs. It treats each binding independently, using rules BIND and BINDA for un-annotated and type-annotated bindings respectively.

In the case of rule BIND, we first produce a constraint for the expression e, Q_{wanted}, using the constraint-generation judgement. Next, we attempt to *simplify* the constraint Q_{wanted}, producing some residual constraint and a substitution for unification variables. We may then generalize a type for f (which we have written as $\forall\overline{a}\,.\,[\overline{\alpha \mapsto a}](Q \Rightarrow \theta\tau)$), with the substitution $[\overline{\alpha \mapsto a}]$ distributing under $\Rightarrow$) and check the rest of the program. It is precisely this generalization over Q-constraints that allows the simplifier to return a residual unsolved constraint to be quantified over. There is no need to check that $\overline{\alpha}\#\mathit{fuv}(\Gamma)$ since this is a top-level judgement, for which the environment contains no free unification variables. Notice additionally that the call to the simplifier is with $Q_{given} = \epsilon$, as there are no given constraints at top-level.

The case of annotated bindings is similar, but we call the simplifier considering as given the constraint Q from the type signature, and *requiring* that no residual constraint is returned with $\mathcal{Q}; Q \overset{simp}{\vdash\!\blacktriangleright} Q_{wanted} \wedge \upsilon \sim \tau \rightsquigarrow \epsilon;\theta$. Hence, the wanted constraint C along with the equality between υ (the inferred type) and τ (the expected type) must be fully solved by θ using the available axioms $\mathcal{Q}$ and given constraints Q. We assume that type signatures do not contain any unification variables and hence the annotation type cannot be affected by θ.

3.6 Soundness and principality of type inference

In this section we show when the type inference algorithm enjoys soundness and infers principal types. A third question, that of completeness, is particularly tricky for constraint systems that, like ours, supports bindings with type signatures. We discuss the issue in detail later, in Section 6. To address soundness and principality we first introduce an auxiliary definition:

Definition 3.2 (Sound and guess-free solutions) *A pair (Q_r, θ) of a constraint Q_r and an (idempotent) unifier θ, is a sound solution for Q_w under given constraint Q_g and top-level axiom schemes $\mathcal{Q}$ when:*

(S1) $\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash \theta Q_w$, *and*
(S2) $dom(\theta)\#fuv(Q_g)$ *and* $dom(\theta)\#fuv(Q_r)$

The pair (Q_r, θ) is a guess-free solution if, in addition:

(P1) $\mathcal{Q} \wedge Q_g \wedge Q_w \Vdash Q_r \wedge \mathcal{E}_\theta$

where $\mathcal{E}_\theta = \{(\alpha \sim \tau) \mid [\alpha \mapsto \tau] \in \theta\}$ is the equational constraint induced by the substitution θ.

In effect, a guess-free solution (Q_r, θ) of Q_w under Q_g and $\mathcal{Q}$ is one where condition S1 holds, and moreover:

$$\mathcal{Q} \wedge Q_g \Vdash Q_w \leftrightarrow Q_r \wedge \mathcal{E}_\theta$$

We elaborate on the notion of sound and guess-free solutions in Sections 3.6.1 and 3.6.2 respectively, where we discuss soundness and principality of type inference.

3.6.1 Soundness

How should a simplifier behave, so that every accepted program in the algorithm instantiated with the particular simplifier is also typeable in the specification? Figure 8 requires the simplifier to return *a sound solution*. Condition S1 is the crux of soundness, and asserts that the original wanted constraint must be deducible, after we have applied the substitution, from the given constraint Q_g and Q_r. Condition S2 requires the domain of the returned substitution to be disjoint from Q_g, which is a trivial property (for now!) since given constraints Q_g only arise from user type annotations that contain no unification variables (rule BindA in Figure 7). In addition, it requires that the domain of θ be disjoint from Q_r. This requirement is there mainly for technical convenience; we assert that the substitution has already been applied to the residual returned constraint, so that we do not have to re-apply it (rule Bind in Figure 7).

Provided that the entailment relation satisfies Figure 3, and the constraint simplifier satisfies the soundness condition in Figure 8 it is routine to show soundness of the inference algorithm.

Theorem 3.1 (Algorithm soundness) *If the entailment satisfies the conditions of Figure 3 and the simplifier satisfies the soundness condition of Figure 8 then $\mathcal{Q}; \Gamma \mapsto$ prog implies $\mathcal{Q}; \Gamma \vdash$ prog in a closed environment Γ.*

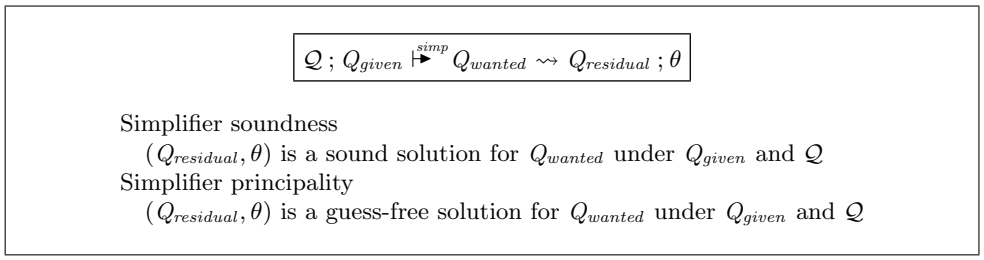

Fig. 8: Simplifier conditions

3.6.2 Principality

We now turn our attention to the *principal types* property, mentioned in Section 2. We will show here that when the algorithm succeeds, it *infers a principal type for a program.* For this to be true, Figure 8 requires a simplifier principality condition, which we motivate and explain below.

In Hindley-Milner type systems the constraints consist of equations between types and the requirement for principality is that the constraint simplifier computes most-general unifiers of these constraints. Consider the constraints for the definition:

```
foo x = x
```

If the variable `x` gets type α and the return type of `foo` is β then the constraint for `foo` is $\alpha \sim \beta$. Of course the solution $[\alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{Int}]$ is a sound one, but it is not the most general one $[\alpha \mapsto \beta]$. The characteristic of the most general solution is that it *makes no guesses*: the most general solution *is entailed directly* from the wanted constraint $\alpha \sim \beta$ (whereas $(\alpha \sim \beta) \not\Vdash (\alpha \sim \texttt{Int}) \wedge (\beta \sim \texttt{Int})$).

Our constraints are generalizations of Hindley-Milner constraints, so we need to come up with an appropriate generalization to the notion of most general solution that "makes no guesses". This generalization is captured with condition P1 in the definition of *guess-free solution* (Definition 3.2), and is reminiscent of similar conditions in abduction-based type inference (Maher, 2005; Sulzmann *et al.*, 2008). We require that the resulting $Q_{residual}$ and θ must follow from the original Q_{wanted} constraint. To better illustrate this definition, consider an example that also involves type classes:

$$\begin{array}{rcl} \mathcal{Q} & = & \texttt{Eq Int} \\ Q_g & = & \texttt{Eq}\ a \end{array}$$

The entailment relation we will use here is a standard entailment relation on equalities, conjunctions and class constraints (we will give a concrete definition in Section 7). If $Q_w = \texttt{Eq}\ \beta$ then the solution $(\epsilon, [\beta \mapsto \texttt{Int}])$ is a sound but *not* necessarily guess-free solution if the entailment cannot deduce that:

$$\mathcal{Q} \wedge Q_g \wedge \texttt{Eq}\ \beta \Vdash \beta \sim \texttt{Int}$$

On the other hand, if:

$$Q_w = \texttt{Eq}\ \beta \wedge [\beta] \sim [a]$$

then $(\epsilon, [\beta \mapsto a])$ is a guess-free solution, provided that

$$\mathcal{Q} \wedge Q_g \wedge \mathtt{Eq}\ \beta \wedge [\beta] \sim [a] \Vdash \beta \sim a$$

To formally state and prove principality, we first have to give a (standard) order relation on constrained types, below. This relation captures when a type is *more general* or *more polymorphic* than another.

Definition 3.3 (More general quantified type) *We say that type* $\forall \overline{a}\,.\, Q_1 \Rightarrow \tau_1$ *is more general than* $\forall \overline{b}\,.\, Q_2 \Rightarrow \tau_2$ *under axiom scheme environment* $\mathcal{Q}$ *iff:*

$$\frac{\mathcal{Q} \wedge Q_2 \Vdash [\overline{a \mapsto \tau}]Q_1 \quad \mathcal{Q} \wedge Q_2 \Vdash [\overline{a \mapsto \tau}]\tau_1 \sim \tau_2 \quad \overline{b} \# \mathit{ftv}(\forall \overline{a}\,.\, Q_1 \Rightarrow \tau_1, \mathcal{Q})}{\mathcal{Q} \vdash \forall \overline{a}\,.\, Q_1 \Rightarrow \tau_1 \preceq \forall \overline{b}\,.\, Q_2 \Rightarrow \tau_2}\ \textsc{subs}$$

In other words, an instantiation of Q_1 must follow from the axiom scheme environment and Q_2, and moreover an instantiation of τ_1 must be equal to τ_2 given the axiom scheme environment $\mathcal{Q}$ and Q_2. It is easy to show that this relation is *reflexive* and *transitive* using the conditions of Figure 3.

To make the notation more compact in the following technical development, we first abbreviate generalization for a top-level binding with the following rule:

$$\frac{Q_1\,;\Gamma \vdash e : \tau \quad \mathcal{Q} \wedge Q \Vdash Q_1 \quad \overline{a} = \mathit{ftv}(Q, \tau)}{\mathcal{Q}\,;\Gamma \overset{gen}{\vdash} e : \forall \overline{a}\,.\, Q \Rightarrow \tau}\ \textsc{GenTop}$$

Correspondingly, we may define algorithmically a generalization step:

$$\frac{\Gamma \overset{\blacktriangleright}{\vdash} e : \tau \rightsquigarrow Q_{wanted} \quad \mathcal{Q}\,;\epsilon \overset{\blacktriangleright}{\vdash}^{simp} Q_{wanted} \rightsquigarrow Q\,;\theta \quad \overline{a}\ \text{fresh} \quad \overline{\alpha} = \mathit{fuv}(\theta\tau, Q)}{\mathcal{Q}\,;\Gamma \overset{\blacktriangleright}{\vdash}^{gen} e : \forall \overline{a}\,.\, [\overline{\alpha \mapsto a}](Q \Rightarrow \theta\tau)}\ \textsc{GenTopAlg}$$

Rules GenTop and GenTopAlg are the generalization steps inlined in rules Bind for top-level bindings in Figure 4 and Figure 7. Since all the types in Γ are closed, we do not have to assert that the unification variables $\overline{\alpha}$ do not appear in Γ.

To show that the algorithm infers principal types we first show that even when no simplification happens *at all*, the inferred type for an unannotated top-level binding is the most general possible.

Lemma 3.1 (Principality of inferred constraint) *If* $Q\,;\Gamma \vdash e : \tau$ *then* $\Gamma \overset{\blacktriangleright}{\vdash} e : \upsilon \rightsquigarrow C$*, and there exists a* θ *with* $\mathit{dom}(\theta) \# \mathit{fuv}(\Gamma)$ *such that* $Q \Vdash \mathbf{simple}[\theta C]$ *and* $Q \Vdash \theta\upsilon \sim \tau$*.*

Proof
Easy induction appealing to the properties of Figure 3. □

Lemma 3.2 *Let* $\mathcal{Q}\,;\Gamma \overset{\blacktriangleright}{\vdash} e : \tau \rightsquigarrow C$*,* $\overline{a}$ *be fresh and corresponding to* $\mathit{fuv}(C, \tau)$*, and* $\varphi = [\mathit{fuv}(C, \tau) \mapsto \overline{a}]$*. If* $\mathcal{Q}\,;\Gamma \overset{gen}{\vdash} e : \sigma$ *then* $\mathcal{Q} \vdash \forall \overline{a}\,.\, \varphi(\mathbf{simple}[C]) \Rightarrow \varphi\tau \preceq \sigma$*.*

Proof
Easy induction, appealing to Lemma 3.1. □

Theorem 3.2 (Algorithm infers principal types) *If* $fuv(\Gamma) = \emptyset$ *and* $\mathcal{Q};\Gamma \overset{gen}{\mapsto} e : \sigma_0$ *then for all* σ *such that* $\mathcal{Q} ; \Gamma \overset{gen}{\vdash} e : \sigma$ *it is the case that* $\mathcal{Q} \vdash \sigma_0 \preceq \sigma$.

Proof
Assume that $\mathcal{Q} ; \Gamma \overset{gen}{\mapsto} e : \forall \overline{a} . [\overline{\alpha \mapsto a}](Q \Rightarrow \theta\tau)$, and by inversion on rule GEN-TOPALG we get:

$$\Gamma \mapsto e : \tau \rightsquigarrow C \tag{1}$$

$$\mathcal{Q} ; \epsilon \overset{simp}{\mapsto} \mathbf{simple}[C] \rightsquigarrow Q ; \theta \tag{2}$$

$$\overline{\alpha} = fuv(\theta\tau, Q) \tag{3}$$

Now, appealing to Lemma 3.2 and transitivity of entailment (Condition R2) it suffices to show that:

$$\mathcal{Q} \vdash \forall \overline{a} . [\overline{\alpha \mapsto a}] Q \Rightarrow [\overline{\alpha \mapsto a}]\theta\tau \preceq \forall \overline{b} . [fuv(C, \tau) \mapsto \overline{b}](\mathbf{simple}[C] \Rightarrow \tau)$$

where we assume without loss of generality that $\overline{b}$ are entirely fresh from any of the type variables of the left-hand-side. To show this we need to find a substitution $[\overline{a \mapsto v}]$ such that:

$$\mathcal{Q} \wedge [fuv(C, \tau) \mapsto \overline{b}](\mathbf{simple}[C]) \Vdash [\overline{a \mapsto v}][\overline{\alpha \mapsto a}] Q$$
$$\mathcal{Q} \wedge [fuv(C, \tau) \mapsto \overline{b}](\mathbf{simple}[C]) \Vdash ([fuv(C, \tau) \mapsto \overline{b}]\tau) \sim [\overline{a \mapsto v}][\overline{\alpha \mapsto a}]\theta\tau$$

Or, equivalently,

$$\mathcal{Q} \wedge [fuv(C, \tau) \mapsto \overline{b}](\mathbf{simple}[C]) \Vdash [\overline{\alpha \mapsto v}] Q \tag{4}$$

$$\mathcal{Q} \wedge [fuv(C, \tau) \mapsto \overline{b}](\mathbf{simple}[C]) \Vdash [\overline{\alpha \mapsto v}]\theta\tau \sim ([fuv(C, \tau) \mapsto \overline{b}]\tau) \tag{5}$$

But, from the simplifier principality condition and the properties of the entailment we get that $\mathcal{Q} \wedge \mathbf{simple}[C] \Vdash Q$ and $\mathcal{Q} \wedge \mathbf{simple}[C] \Vdash \mathcal{E}_\theta$. From the former, (4) follows, by picking $\overline{v} = [fuv(C, \tau) \mapsto \overline{b}]\overline{\alpha}$. For (5), by the reflexivity requirement of entailment we know that $\mathcal{Q} \wedge \mathbf{simple}[C] \Vdash \tau \sim \tau$, and moreover by the simplifier principality requirement we have $\mathcal{Q} \wedge \mathbf{simple}[C] \Vdash \mathcal{E}_\theta$. By substitutivity of entailment and the fact that type equality is an equivalence relation we know that $\mathcal{Q} \wedge \mathbf{simple}[C] \Vdash \theta\tau \sim \tau$ and with the appropriate freshening of unification variables (which uses property R3) we get the result. □

The fact that the algorithm infers principal types is important, but weaker than the actual *principal types* property, which can be formally stated as follows.

Definition 3.4 (Principal types) *If* $\mathcal{Q} ; \Gamma \overset{gen}{\vdash} e : \sigma$ *then there exists a* σ_0 *such that* $\mathcal{Q};\Gamma \overset{gen}{\vdash} e : \sigma_0$*, and for all* σ_1 *with* $\mathcal{Q};\Gamma \overset{gen}{\vdash} e : \sigma_1$ *it is the case that* $\mathcal{Q} \vdash \sigma_0 \preceq \sigma_1$.

In particular, Theorem 3.2 says nothing about the situation where the algorithm fails to produce a type, or the simplifier does not terminate — that is, it says nothing about completeness. Indeed, as we discuss in Section 6, any guess-free solver will necessarily be incomplete with respect to a natural type system specification. In the light of this observation, Theorem 3.2 is remarkable: Even when the algorithm is incomplete (and, as we will see later, even when the type system lacks

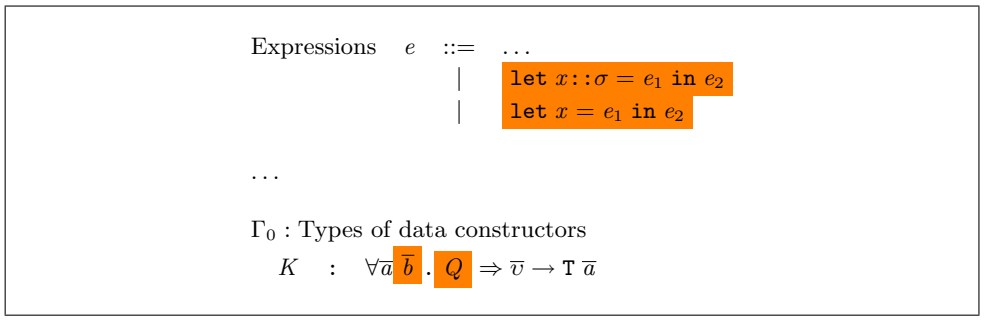

Fig. 9: Syntax extensions that introduce local assumptions

principal types), the aforementioned lightweight conditions on the simplifier and the entailment will guarantee that unannotated bindings that are accepted by the algorithm do have principal types, as modularity mandates.

4 Constraint-based type systems with local assumptions

Now that we have established our baseline, we are ready to introduce local type assumptions, the main focus of this paper. The changes appear modest, but have a far-reaching impact. Figure 9 gives the extended syntax, highlighting the changes compared to Figure 1, while Figure 10 does the same for the typing rules, again highlighting changes compared with Figure 2.

There are three main changes. First, we add local `let`-bound definitions that are accompanied with user-supplied, potentially polymorphic type signatures. The corresponding typing rule LetA is quite straightforward: It makes the constraint Q_1 from the type signature of an annotated local `let`-bound definition available for type checking the right-hand side of the definition, e_1.

The second modification is an innocent-looking extension of the types of data constructors, and the corresponding typing rule. This is where GADTs manifest themselves, as we discuss in Section 4.1.

The third change is the addition of *un-annotated* `let`-bound definitions; that is, ones unaccompanied by a type signature. Rule Let is unusually simple because, in contrast to a traditional Hindley-Milner type system, it performs no generalisation. We devote the whole of Section 4.2 to an explanation of this unconventional design choice.

4.1 Data constructors with local constraints

The key feature of GADTs is that *a GADT pattern match brings local type-equality constraints into scope*. For example, given the GADT

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

when pattern-matching on constructor `T1` we know, in that `case` branch only, that the scrutinee has type `T Bool`. While the declaration for the GADT `T` above is very

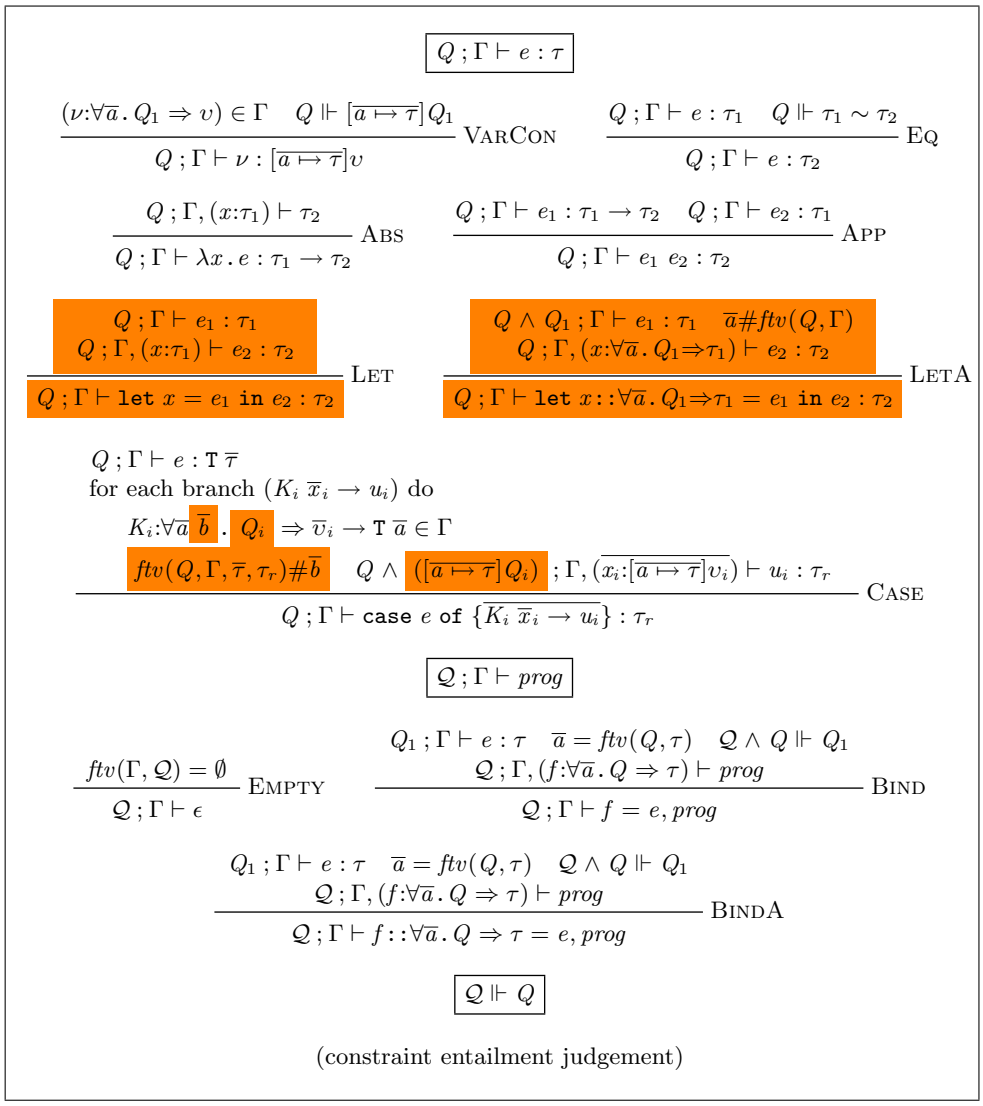

Fig. 10: Natural but over-permissive typing rules

convenient for the programmer, it is helpful for our understanding to re-express it with an explicit equality constraint, like this:

```
data T :: * -> * where
  T1 :: (a ~ Bool) => Int -> T a
  T2 :: T a
```

(GHC allows both forms, and treats them as equivalent.) You may imagine a value of type `T` τ, 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 in the body of the pattern match. 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 (Section 8).

In general (see Figure 9) the type of a data constructor K in a data type T must take the form

$$K : \forall \overline{a}\, \overline{b} \,.\, Q \Rightarrow \overline{\upsilon} \rightarrow \mathtt{T}\ \overline{a}$$

Compared with Figure 1, constructor types have two new features, each of which is reflected in rule Case:

1. The type variables $\overline{b}$ are the *existential* variables of the constructor, that, unlike the $\overline{a}$, do not appear in the return type of the constructor.[2] For instance, the `b` variable in the following definition is one such variable:

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

   The side condition $ftv(Q, \Gamma, \overline{\tau}, \tau_r) \# \overline{b}$ in rule Case checks that the existential variables do not escape in the environment Q, Γ, or the scrutinee type $\mathtt{T}\ \overline{\tau}$, or the result type τ_r. In the following example, `fx1` is well-typed, but `fx2` is not because the existential variable `b` escapes:

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

2. The constraint Q in K's type must be *satisfied* at calls of K, but *becomes available* when pattern matching against K. So, in rule Case, the constraint Q_i from K_i's type, suitably instantiated, is added to the ambient constraints used for typing the case alternative u_i. Note that there is no need for a consistency requirement in the constraint $Q \wedge ([\overline{a \mapsto \tau}]\, Q_i)$, since an inconsistent constraint simply means unreachable code operationally. Unreachable code is always safe, no matter how it is typed.

So far we have focused on GADTs, but it is completely natural to generalise the idea of GADTs, in which constructors can have a type involving an equality constraint, to allow *arbitrary* constraints in the type of data constructors. to go to extra trouble to prevent such a possibility. For example, we might write

```
data Showable where
  MkShowable :: (Show a) => a -> Showable

display :: Showable -> String
display (MkShowable x) = show x ++ "\n"
  -- Recall that show :: Show a => a -> String
```

[2] They are called existential, despite their apparent quantification with $\forall$, because the constructor's type is isomorphic to $\forall \overline{a} \,.\, (\exists \overline{b} \,.\, Q_i \times \overline{\upsilon}_i) \rightarrow \mathtt{T}\ \overline{a}$.

Here, `Showable` is an existential package, pairing a value of type `a` with a dictionary for `(Show a)`. Pattern matching on a `Showable` gives access to the dictionary: `display` does not have a `Show` constraint, despite the use of `show` in its body, because the constraint is discharged by the pattern match.

In `Showable` the class constraint affected the existential variable. But in this generalized setting we may also constrain the type parameter of the data type:

```
data Set :: * -> * where
  MkSet :: (Ord a) => [a] -> Set a

union :: Set a -> Set a -> Set a
union (MkSet xs1) (MkSet xs2) = MkSet (merge xs1 xs2)
  -- Assume merge :: Ord a => [a] -> [a] -> [a]

empty :: Ord a => Set a
empty = MkSet []
```

A `MkSet` constructor packages an `Ord a` dictionary with the list. This dictionary is used to discharge the `Ord` constraint required by `merge`. This very natural generalisation is new to GHC.

4.2 `let` *should not be generalized*

A central feature of the Hindley-Milner system is that `let`-bound definitions are *generalised*. For example, consider the slightly artificial definition

```
f x = let g y = (x,y) in (g 3, g False)
```

The definition for `g` is typed in an environment in which `x` : a, and the inferred type for `g` is $\forall b \;.\; b \to (a, b)$. This type is polymorphic in b, but not in a, because the latter is free in the type environment at the definition of `g`. This side condition, that `g` should be generalised only over variables that are not free in the type environment, is the only tricky point in the entire Hindley-Milner type system.

Recall now the GADT of Section 2:

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

and consider the following function definition:

```
fr :: a -> T a -> Bool
fr x y = let gr z = not x  -- not :: Bool -> Bool
         in case y of
             T1 _ -> gr ()
             T2   -> True
```

The reader is urged to pause for a moment to consider whether `fr`'s definition is type-safe. After all, `x` clearly has type `a`, and it is passed as an argument to the

boolean function `not`. Any normal Hindley-Milner type checker would unify `a` with `Bool` and produce a type error and reject the program.

Yet the program *is* type safe – there is a type for `gr` that makes the program type check, namely

```
gr :: forall b. (a ~ Bool) => b -> Bool
```

Rather than *rejecting* the constraint `a`$\sim$`Bool`, we *abstract over it*, thereby deferring the (potential) type error to `gr`'s call site. At any such call site, we must provide evidence that `a`$\sim$`Bool`, and indeed we can do so in this case, since we are in the `T1` branch of the match on `y`. In short, to find the most general type for `gr`, we must abstract over the equality constraints that arise in `gr`'s right hand side.

However, we *do not seek* this outcome: in our opinion, most programmers would expect `fr`'s definition to be rejected. But the fact is that in a system admitting equality constraints, and which allows quantification over constraints, the principal type for `gr` is the one written above.

The very same issue arises with type-class constraints. Consider this definition of `fs`, which uses the data type `Set` from Section 4.1:

```
fs :: a -> Set a -> Bool
fs x y = let gs z = x > z
         in case y of
            MkSet vs -> gs (head vs)
```

Again, the most general type of `gs` is

```
gs :: (Ord a) => a -> String   -- Not polymorphic in a
```

where we abstract over the `(Ord a)` constraint even though `gs` is not polymorphic in `a`. Given this type, the call to `gs` is well typed, as is the whole definition of `fs`. It should be obvious that the two examples differ only in the kind of constraint that is involved.

So what is the problem? Typically, for a `let` binding, we infer the type τ_1 of the right-hand side, gathering its type constraints Q_1 at the same time. Then we may *generalise* the type, by universally quantifying over the type variables $\overline{a}$ that are free in τ_1 but are not mentioned in the type environment. But what about Q_1? We discuss next the various ways in which it can be treated.

For the sake of simplicity, in the discussion below we will ignore the top-level axiom set $\mathcal{Q}$: it only makes things more problematic still.

4.2.1 **GenAll***: abstract over all the constraints*

One robust and consistent choice (made, for example, by Pottier (Pottier & Rémy, 2005; Simonet & Pottier, 2007)) is to abstract over the *whole* constraint Q_1, regardless of whether the constraint mentions the quantified type variables $\overline{a}$, to form the

type $\forall \overline{a}\,.\,Q_1 \Rightarrow \tau_1$. Here is the typing rule for `let` under the **GenAll** approach:

Qualified types: **Yes**, Generalization: **Yes**

$$\frac{Q_1\,;\Gamma \vdash e_1 : \tau_1 \quad \overline{a} = ftv(Q_1,\tau_1) - ftv(Q,\Gamma) \qquad Q\,;\Gamma,(x{:}\forall \overline{a}\;.\;Q_1 \Rightarrow \tau_1) \vdash e_2 : \tau_2}{Q\,;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{LET}$$

However **GenAll** has serious disadvantages, of two kinds. First, and most important, there are costs to the programmer:

- It leads to unexpectedly complicated types, such as those for function `gr`. The larger the right-hand side, the more type constraints will be gathered and abstracted over. For type-class constraints this might be acceptable, but equality constraints are generated in large numbers by ordinary unification. Although they do not appear in the program text, these types may be *shown* to the programmer by an IDE; and must be *understood* by the programmer if she is to know which programs will typecheck and which will not.
- There are strong software-engineering reasons not to generalise constraints unnecessarily, because doing so postpones type errors from the definition of `gr` to (each of) its occurrences. If, for example, `gr` had been called in the `T2` branch of `fr`, as well as the `T1` branch, a mystifying error would ensue: "Cannot unify `a` with `Bool`". Why? Because the call to `gr` would require $\texttt{a} \sim \texttt{Bool}$ to be satisfied, and in the `T2` branch no local knowledge is available about a, yielding the constraint unsatisfiable. To understand such errors the programmer will have to construct in her head the principal type for `gr`, which is no easy matter. Moreover, one such incomprehensible error will be reported for each call to `gr`.
- In an inference algorithm, it turns out that we need a new form of constraint, an *implication constraint*, that embodies deferred typing problems (Section 5). Under **GenAll** it is necessary to abstract over implication constraints too, which further complicates the programmer's life (because she sees these weird types). This raises the question of whether implication constraints should additionally be allowed in valid type signatures, which in turn leads to open research problems in tractable solver procedures for constraints with implications in their assumptions (Simonet & Pottier, 2007).

Second, there are costs to the type inference engine:

- At *each call site* of a generalized expression, the previously abstracted large constraints have to be solved separately. This makes efficient type inference harder to implement.
- Almost all existing Haskell type inference engines (with the exception of Helium (Heeren *et al.*, 2003)) use the standard Hindley-Milner algorithm, whereby unification (equality) constraints are solved "on the fly" using in-place update of mutable type variables (Peyton Jones *et al.*, 2007). This is simple and efficient, which is important since equality constraints are numer-

ous. (In contrast the less-common type-class constraints are gathered separately, and solved later.)
Under **GenAll**, we can no longer eagerly solve *any unification constraint whatsoever* on the fly. An equality $a \sim \tau$ must be suspended (i.e. not solved) if a is free in the environment at some enclosing `let` declaration.
Moreover, in compilers with a typed intermediate language, such as GHC, each abstracted constraint leads to an extra type or value parameter to the function, and an extra type or value argument at its occurrences.

These costs might be worth bearing if there was a payoff. But in fact the payoff is close to zero:

- Programmers do not expect `fr` and `fs` to typecheck, and will hardly be delirious if they do so in the future. Indeed, GHC currently rejects both `fr` and `fs`, with no user complaints.
- The generality of `gr` and `gs` made a difference only because their occurrences were under a pattern-match that bound a new, local constraint. Such pattern matches are rare, so in almost all cases the additional generalisation is fruitless. But it cannot be omitted (at least not without a rather ad-hoc pre-pass) because when processing the perfectly vanilla definition of `gr` the type checker does not know whether or not `gr`'s occurrences are under pattern-matches that bind constraints.

In short, we claim that generalising over all constraints carries significant costs, and negligible benefits. Probably the only true benefit is that **GenAll** validates *let-expansion*; that is, `let` $x = e$ `in` u typechecks if and only if $[x \mapsto e]u$ typechecks. The reader is invited to return to `fr` and `fs`, to observe that both do indeed typecheck with no complications if `gr` (resp `gs`) is simply inlined. Let-expansion is a property cherished by type theorists and sometimes useful for automatic code refactoring tools, but we believe that its price has become too high.

4.2.2 **NoQual***: Generalization without qualified types*

The undesirability of **GenAll** concerned the abstraction of constraints, rather than generalisation *per se*. What if the specification simply insisted that the type inferred for a `let` binding was always of the form $\forall \overline{a}\ .\ \tau$, with no "$Q \Rightarrow$" part? This is easy to specify:

Qualified types: **No**, Generalization: **Yes**

$$\frac{Q\,;\Gamma \vdash e_1 : \tau_1 \quad \overline{a} = ftv(\tau_1) - ftv(Q,\Gamma) \qquad Q\,;\Gamma, (x{:}\forall \overline{a}\,.\,\tau_1) \vdash e_2 : \tau_2}{Q\,;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{LET}$$

When Q is empty, this is the usual rule for the Hindley-Milner system. In terms of an inference algorithm, what happens in Hindley-Milner is this. Equality constraints are gathered from the right-hand side, but are *completely solved* before generalisation. A unique solution is guaranteed to exist, namely the most general

unifier. (In Hindley-Milner the constraints are typically solved on-the-fly but that is incidental.)

As previous work shows (Schrijvers *et al.*, 2009), this approach continues to work for a system that has GADTs only. Again, a given set of constraints can always be uniquely solved (if a solution exists) by first-order unification.

Alas, adding type classes makes the system fail, in the sense of lacking principal types, because type-class constraints do not have unique solutions in the way that equality constraints do. For example, suppose that in the definition `let` $x = e_1$ `in` e_2, that

- The type of e_1 is $b \to b$.
- b is not free in the type environment.
- The constraints arising from e_1 are `Eq` b.

We cannot solve the constraint without knowing more about b – but in this case we propose to quantify over b. If we quantify over b the only reasonable type to attribute to x is

$$x :: \forall b \;.\; \texttt{Eq}\ b \Rightarrow b \to b$$

That is illegal under **NoQual**. As a result, x has many incomparable types, such as `Int` $\to$ `Int` and `Bool` $\to$ `Bool`, but no principal type.

4.2.3 **PartQual***: Restricted qualified types and generalization*

We have learned that, if we are to generalise `let`-bound variables we must quantify over their type-class constraints (**NoQual** did not work); but we have argued that it is undesirable to quantify over all constraints (i.e. **GenAll**). The obvious alternative is to quantify over type-class constraints, but *not* over equality constraints. More generally, can we identify a particular kinds of constraints over which the specification is allowed to abstract? We call this choice **PartQual**, and use a predicate $good(Q)$ to identify abstractable constraints:

Qualified types: **Restricted**, Generalization: **Yes**

$$\frac{\begin{array}{c} Q, Q_1 \,;\, \Gamma \vdash e_1 : \tau_1 \quad \overline{a} = ftv(Q_1, \tau_1) - ftv(Q, \Gamma) \\ good(Q_1) \quad Q \,;\, \Gamma, (x{:}\forall \overline{a} \,.\, Q_1 \Rightarrow \tau_1) \vdash e_2 : \tau_2 \end{array}}{Q \,;\, \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{LET}$$

The problem with this approach is that it is not clear what such class of constraints would be. It is not enough to pick out equality constraints, because some class constraints may behave like equality constraints, such as *type classes with equality superclasses* as:

```
class (a ~ b) => REq a b
```

Worse, some class constraints with *functional dependencies* may give rise to extra equality constraints, only when found in certain contexts:

```
class C a b | a -> b
```

If two constraints `C Int b` and `C Int Char` appear in the same context, a new equality must hold, namely that `b ~ Char`.

4.2.4 **NoGen***: no generalization!*

It seems clear that **NoQual** and **PartQual** are non-starters, and we have argued that **GenAll**, while technically straightforward is practically undesirable. The last, and much the simplest choice, is to perform no generalisation whatsoever for inferred `let` bindings. The typing rule for our option, **NoGen**, is very simple:

Qualified types: **No**, Generalization: **No**

$$\frac{Q\,;\Gamma \vdash e_1 : \tau_1 \qquad Q\,;\Gamma, (x{:}\tau_1) \vdash e_2 : \tau_2}{Q\,;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \textsc{let}$$

Hence **NoGen** omits the entire generalization step; the definition becomes completely monomorphic. Notice though that **NoGen** applies only for *local* and *unannotated* `let`-bindings. For *annotated* local `let`-bindings, $\texttt{let}\ x :: \sigma = e_1\ \texttt{in}\ e_2$, where the programmer supplies a (possibly polymorphic) type signature σ, the type system may use that type signature (rule LetA in Figure 10). For top-level bindings generalization poses no difficulties since there are no free type variables in a top-level environment (rule Bind, Figure 10).

Under **NoGen**, both `fr` and `fs` are rejected, which is fine; we did not seek to accept them in the first place. But hang on! **NoGen** means that some vanilla ML or Haskell 98 functions that use polymorphic local definitions, such as the `f` function in the very beginning of Section 4.2, will be rejected. That is, **NoGen** is not a conservative extension of Haskell. Surely programmers will hate that? Actually not. In the next Section (4.2.5) we will present evidence that programmers almost never use locally-defined values in a polymorphic way (without having provided a type signature). In the rare cases where a local value *has to be* used polymorphically, the programmer can readily evade **NoGen** by simply supplying a type signature.

In summary, generalisation of local `let` bindings (without annotations) is a device that is almost never used, and its abolition yields a dramatic simplification in both the specification and implementation of a typechecker. The situation is strongly reminiscent of the debate over ML's *value restriction*. In conjunction with assignment, unconditional generalisation is unsound. Tofte proposed a sophisticated work-around (Tofte, 1990). But Wright subsequently proposed the value restriction, whereby only syntactic values are generalised (Wright, 1995). The reduction in complexity was substantial, and the loss of expressiveness was minor, and Wright's proposal was adopted.

4.2.5 Impact on existing Haskell programs

Our **NoGen** proposal will reject some programs that would be accepted by any Haskell or ML compiler. This is bad in two ways:

Backward compatibility. Existing programs will break. But how many programs break? And how easy is it to fix them?

Convenience. Even for newly-written programs, automatic generalisation is convenient. But how inconvenient is programming without it?

To get some quantitative handle on these questions we added a flag to GHC that implements **NoGen**, and performed the following two experiments.

The libraries. We compiled all of the Haskell libraries that are built as part of the standard GHC build process, and fixed all failures due to **NoGen**. These libraries comprise some 30 packages, containing 533 modules, and 94,954 lines of Haskell code (including comments). In total we found that 20 modules (3.7%) needed modification. The changes affected a total of 127 lines of code (0.13%), and were of three main kinds:

- There are a few occurrences of a polymorphic function that could be defined at top level, but was actually defined locally. For example `Control.Arrow.second` has a local definition for

  ```
  swap ~(x,y) = (y,x)
  ```

- One programmer made repeated use of the following pattern

  ```
  mappend a b = ConfigFlags {
    profLib     = combine profLib,
    constraints = combine constraints,
    ...
  }
    where combine :: Monoid t => (ConfigFlags->t) -> t
          combine field = field a `mappend` field b
  ```

  (The type signature was added by ourselves.) Notice that `a` and `b` are free in `combine`, but that `combine` is used for fields of many different types; for example, `profLib`::`Flag Bool`, but `constraints`::`[Dependency]`. This pattern was repeated in many functions. We fixed the code by adding a type signature, but it would arguably be nicer to make `combine` a top-level function, and pass `a` and `b` to it.

- The third pattern was this:

  ```
  let { k = ...blah... } in gmapT k z xs
  ```

  where `gmapT` is a function with a rank-2 type:

  $$\texttt{gmapT} :: \forall a \,.\, \texttt{Data}\ a \Rightarrow (\forall b \,.\, \texttt{Data}\ b \Rightarrow b \to b) \to a \to a$$

  Here, `k` really must be polymorphic, because it is passed to `gmapT`. GHC's libraries include the Scrap Your Boilerplate library of generic-programming (Lämmel & Peyton Jones, 2003; Lämmel & Peyton Jones, 2005) functions that make heavy use of higher rank functions (Peyton Jones *et al.*, 2007), but in vanilla Haskell code one would expect them to be much less common. Still, such errors can be fixed by providing a type signature for `k`.

Packages on Hackage As a second, and much larger-scale, experiment we compiled all of the third-party Haskell packages on the Hackage library, both with and without **NoGen**, and recorded whether or not the package compiled successfully with the **NoGen** flag on. We found 793 packages that compiled faultlessly with the baseline compiler that we used. When we disabled generalisation for local `let` bindings, 95 of the 793 (12%) failed to compile. We made no attempt to investigate what individual changes would be needed to make the failed ones compile. Since the chances of an entire package compiling without modification decreases *exponentially* with the size of the package, so one would expect a much larger proportion of *packages* to fail than of *modules* (c.f. the 3.7% of base-package modules that required modification).

Summary Although there is more work to do, to see how many type signatures are required to fix the failing third-party packages, we regard these numbers as very promising: even in the higher-rank-rich base library, only a vanishingly small number of lines needed changing. We conclude that local `let` generalisation is rarely used. Moreover, as a matter of taste, in almost all cases we believe that the extra type signatures in the modified base-library code have improved readability. Finally, although our experiments involve Haskell programs, we conjecture that the situation is similar for ML variants.

4.3 The lack of principal types

We have seen how local assumptions may be treated in the typing rules and how they affect local `let` generalization in Figure 10. However, as discussed in Section 2, the addition of local assumptions means that the system now lacks principal types. Recall the `test` function from the Introduction:

```
data T :: * -> * where       test (T1 n) r = n > 0
  T1 :: Int -> T Bool        test _      r = r
  T2 :: T a
```

Rule Bind of Figure 10 allows `test` to enter the environment with either type $\forall a.\texttt{T}\ a \rightarrow \texttt{Bool} \rightarrow \texttt{Bool}$ or $\forall a.\texttt{T}\ a \rightarrow a \rightarrow a$, depending on whether the local assumptions from the pattern matching are used or not. In this case, there is no type (quantified or not) that can be assigned to `test` and that is more general than the other two. Our natural type system for local assumptions accepts programs that have no principal types (despite the lack of local `let` generalization, which is irrelevant here).

The type system of Figure 10 accepts programs that have no principal types.[3] Sadly, we do not know how to devise a simple declarative type system without these problems (Section 6), and hence we embark in the rest of the paper to the detailed description of an algorithmic strategy that type checks *fewer* programs –

[3] Equally badly, it admits programs that exhibit the *ambiguity* problem, that we explain in Section 6.

but only ones that can be assigned principal types. Principal types seem important for software engineering, but Section 9.7 also presents some subtle points related to this design choice. Still, we regard Figure 10 as the natural type system for local assumptions against which any such algorithm or restricted type system should be compared.

5 Type inference with OutsideIn(X)

We seek a tractable and efficient type inference algorithm that accepts only programs with principal types in the natural type system of Figure 10. We describe such an algorithm in this section.

5.1 Type inference, informally

Let us consider what happens in terms of constraint generation and solving when local assumptions from GADTs enter the picture. Here is an 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:

α	type of the entire body of the function
β_x	type of `x`

Matching x against a constructor from type `T` imposes the constraint $\beta_x \sim \texttt{T}\ \gamma$, for some new unification variable γ. 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

$$[\alpha \mapsto \texttt{Bool}] \quad \text{and} \quad [\alpha \mapsto \gamma]$$

are both solutions, but neither is more general than the other. Each solution leads to a distinct incomparable type for the expression.

On the other hand, sometimes there obviously *is* a unique solution. Consider `test2` from Section 2:

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

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

$$(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool}) \quad \wedge \quad (\alpha \sim \texttt{Bool})$$

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

In general, multiple pattern clauses give rise to a conjunction of implication constraints and the task of type inference is to find a substitution that solves each of the conjunctions. In the next section we sketch our idea of how to get a decidable algorithm that infers most general solutions (and, ultimately, principal types), by considering a restricted implication solver.

5.2 Overview of the OutsideIn(X) *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]". In the example

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

both α and γ are global unification variables and we must refrain from unifying them when solving the constraint. Hence, the constraint by itself is insoluble. It can be solved only if there is some *other* constraint that fixes α.

On the other hand, some of the unification variables in an implication may be entirely local to this implication. Consider the following variation:

```
\x -> case x of { T3 n -> null n }
```

where `T3` is yet another data constructor of type:

$$\texttt{T3} : \forall a \,.\, (\texttt{Bool} \sim a) \Rightarrow \texttt{[Int]} \rightarrow \texttt{T}\ a$$

Via the same reasoning as before we'd get that from the given assumption $\gamma \sim \texttt{Bool}$, it must follow that $\alpha \sim \texttt{Bool}$ where α is the return type of the branch. However from the instantiation of the `null` function of type $\forall d \,.\, [d] \rightarrow \texttt{Bool}$ with a fresh unification variable δ and the application `null n` we get an additional constraint $\delta \sim \texttt{Int}$. In total, we have[5]:

$$\gamma \sim \texttt{Bool} \supset (\alpha \sim \texttt{Bool} \wedge \delta \sim \texttt{Int})$$

Now δ is an entirely local variable to this implication constraint, and hence (unlike α and γ) it does not matter what type we unify it with. We record this information in the syntax of implications, using an existential quantifier that binds these local unification variables:

$$\exists \delta.(\gamma \sim \texttt{Bool} \supset \alpha \sim \texttt{Bool} \wedge \delta \sim \texttt{Int})$$

When solving this constraint, we are free to unify $[\delta \mapsto \texttt{Int}]$ but *not* α nor γ.

The formal syntax of the constraints generated by the algorithm, C, is now given in Figure 11, which is identical to Figure 5, except for the new highlighted form of *implication constraint*. An implication constraint is of the form $\exists \overline{\alpha}.(Q \supset C)$ where

[4] We must treat the result type or the type of the expression we are pattern matching against as part of the type environment.

[5] We will treat $\supset$ as a very low precedence operator, so the parentheses are redundant

Unification variables	$\alpha, \beta, \gamma, \ldots$		
Unification or skolem variables	tv	$::=$	$\alpha \mid a$
Algorithm-generated constraints	C	$::=$	$Q \mid C_1 \wedge C_2 \mid \exists \overline{\alpha}.(Q \supset C)$
Free unification variables	$fuv(\cdot)$		
	$\mathbf{simple}[Q]$	$=$	Q
	$\mathbf{simple}[C_1 \wedge C_2]$	$=$	$\mathbf{simple}[C_1] \wedge \mathbf{simple}[C_2]$
	$\mathbf{simple}[\exists \overline{\alpha}.(Q \supset C)]$	$=$	ϵ
	$\mathbf{implic}[Q]$	$=$	ϵ
	$\mathbf{implic}[C_1 \wedge C_2]$	$=$	$\mathbf{implic}[C_1] \wedge \mathbf{implic}[C_2]$
	$\mathbf{implic}[\exists \overline{\alpha}.(Q \supset C)]$	$=$	$\exists \overline{\alpha}.(Q \supset C)$

Fig. 11: Syntax extensions for OUTSIDEIN(X)

we call the $\overline{\alpha}$ variables the *touchables* of the constraint. These are the variables that we are allowed to unify when solving the implication constraint. Notice that the assumption of the implication constraint is always a Q constraint, as an implication enters life by a pattern match against a constructor or a type signature – which both introduce Q-constraints (not C-constraints). As a convenience, we will often omit the $\exists \overline{\alpha}$ part of an implication if it $\overline{\alpha}$ is empty. The function $\mathbf{simple}[C]$ returns the simple (that is, non-implication) constraints of C, whereas $\mathbf{implic}[C]$ returns the implications, so that $C = \mathbf{simple}[C] \wedge \mathbf{implic}[C]$.

To solve a constraint C, we may proceed as follows:

1. Split C into the implications of C, $\mathbf{implic}[C]$, and the rest, $\mathbf{simple}[C]$.
2. We solve the simple constraints by using some solver for X, which takes care of the Q constraints.
3. We use the information generated by solving the simple constraints (such as a substitution for unification variables) to solve each implication, one at a time, taking care to allow only unification of its touchable variables.

This algorithm is conservative: there may exist constraints that admit a unique solution, which it may fail to solve.

Example 5.1 (OUTSIDEIN(X) conservativity) *The algorithm fails to solve the constraint*

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

because α is not a touchable variable, but the constraint actually has a unique solution, namely $[\alpha \mapsto \texttt{Int}]$.

However, we argue that failing to solve Example 5.1 is really not so bad. For, in the presence of a top-level axiom $F\ \texttt{Bool} \sim \texttt{Int}$, the constraint would be ambiguous after all: there would exist an incomparable solution, $[\alpha \mapsto F\ \gamma]$. Hence, our design decision to not unify global variables under a local constraint in fact makes the algorithm *robust with respect to an open world* where new axioms can be added

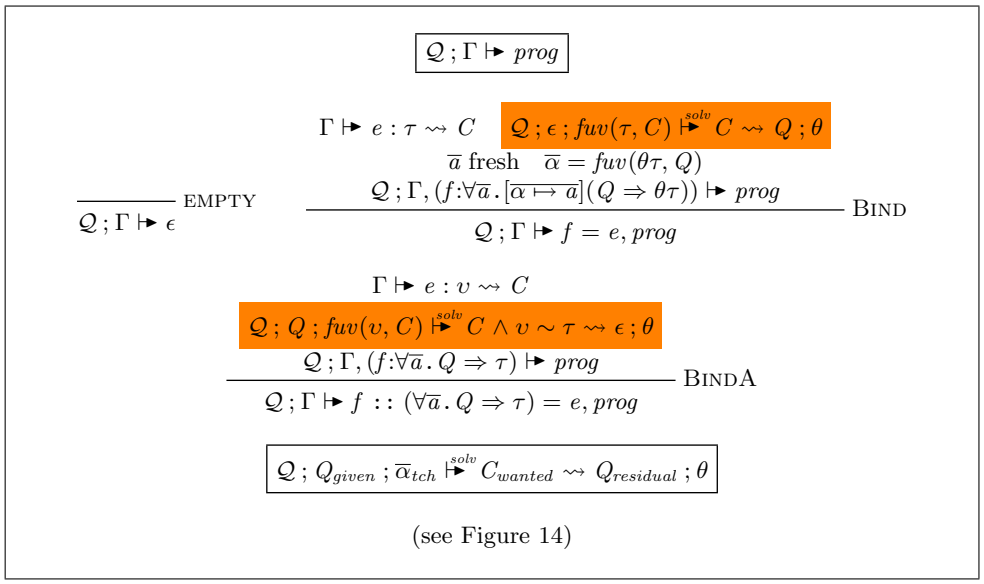

Fig. 12: Top-level algorithmic rules

at any time. Any unique principal solution obtained by our algorithm remains a unique principal solution with respect to any consistent extension of the axiom set. Does the design decision make sense for a constraint like

$$(\texttt{Eq}\ \gamma \supset \alpha \sim \texttt{Int})$$

where the local constraint is not an equality constraint? Surely `Eq` γ cannot contribute to solving $\alpha \sim$ `Int`, and the only sensible solution is $[\alpha \mapsto$ `Int`$]$? Indeed, this is the case for all choices of the constraint language X that have been extensively studied in the literature. However, OutsideIn(X) is parameteric in the particular choice of X and thus prepared for all possible shapes of axioms, even less obvious ones. In the current example, for instance, the unusually-shaped axiom $\forall x$. `Eq` $x \Rightarrow F\ x \sim$ `Int` would give rise to an alternate solution $[\alpha \mapsto F\ \gamma]$. Besides, Schrijvers et al. (2008b) have already shown the merits of this shape of axiom and argued for their adoption. OutsideIn(X) already comes perpared for this and other, as of yet unanticipated, extensions of the constraint language.

5.3 Top-level algorithmic rules

As in the vanilla language of Section 3, our approach relies on constraint generation and solving, for expressions and top-level bindings, with the judgements

$$\Gamma \vdash\!\blacktriangleright e : \tau \rightsquigarrow C \qquad\qquad \mathcal{Q}\,;\Gamma \vdash\!\blacktriangleright \mathit{prog}$$

The top-level algorithmic rules are given in Figure 12. The judgement $\mathcal{Q};\Gamma \vdash\!\blacktriangleright \mathit{prog}$ looks very much like the vanilla judgement in Figure 7, but notice the highlighted differences – since OutsideIn(X) has to deal with implication constraints and untouchable variables, it is natural to rely on a different, more elaborate, solver, which we present in Section 5.5.

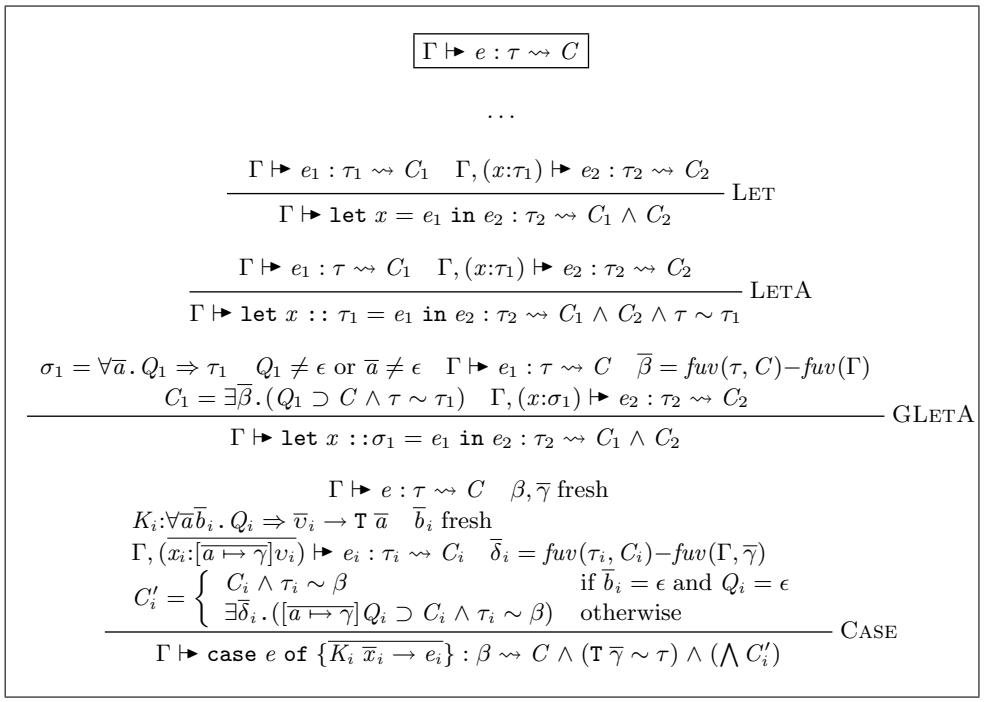

Fig. 13: Constraint generation

5.4 Generating constraints

Constraint generation is the same as Figure 6, with extensions for the new syntax forms and modifications shown in Figure 13.

Note that Let does not generalize the binding, according to the discussion in Section 4.2. For annotated `let`-bound definitions we consider two cases:

- The first case (rule LetA) triggers when the annotation is monomorphic. In that case we have to gather the constraints but also record the fact that the inferred type for the `let`-bound definition, τ, is equal to the required type τ_1, with the constraint $\tau \sim \tau_1$.
- In the case where the annotation is polymorphic (rule GLetA) and hence introduces some quantified variables $\overline{a}$ and potentially some constraint Q_1 we must first infer a type τ and constraint C for e_1. At this point the constraint C must be provable by the local assumption Q_1 introduced by the type annotation and hence we emit an implication constraint $\exists \overline{\beta}\,.\,(Q_1 \supset C \wedge \tau \sim \tau_1)$, where $\overline{\beta}$ are the variables that we are allowed to unify. We then proceed to check e_2 using the annotation signature for the `let`-bound definition.

The reader may be surprised to see that the quantified variables $\overline{a}$ do not explicitly appear in the emitted constraints in rule GLetA. After all, for type safety we should be preventing any variable from the environment to be unified with those quantified variables. But note that we are *only* allowed to unify the local variables of the implication, $\overline{\beta}$. Hence hence it is plainly impossible to unify some variable

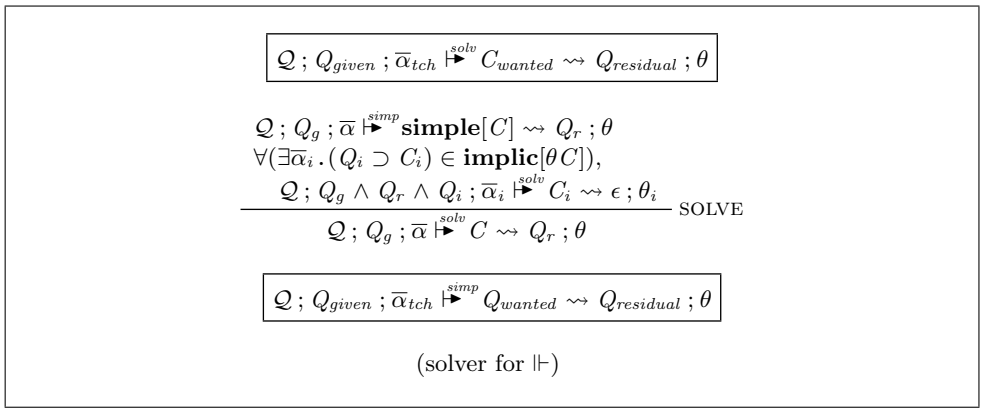

Fig. 14: Solver infrastructure

from the environment to *any type* at all, including $\overline{a}$! This behavior is more strict than the current Haskell implementations of polymorphic signatures, an issue that we return to in Section 5.6.1.

Rule Case deals with pattern matching. First, we generate a type τ and constraint C for the scrutinee of the case expression e. Now we generate a constraint $(\mathtt{T}\ \overline{\gamma} \sim \tau)$, for fresh unification variables $\overline{\gamma}$, to reflect the fact that the scrutinee's must match the return type constructor T of the patterns. Now, contrary to the vanilla rule of Figure 6, the modified rule considers two cases:

- If constructor K_i brings no existential variables or constraints into scope ($\overline{b} = \epsilon$ and $Q_i = \epsilon$) then all is straightforward, as in rule Case of Figure 6.
- However, if the data constructor K does bring some constraints or existential variables in scope ($Q_i \neq \epsilon$ or $\overline{b} \neq \epsilon$) then we may treat this branch as a GADT branch, by introducing an implication constraint that records the touchable variables of the branch, $\overline{\delta}_i$, and the local assumptions $[\overline{a \mapsto \gamma}]\,Q_i$.

Once again, the careful reader may be surprised to see that the existential variables $\overline{b}$ are not mentioned explicitly somewhere in the resulting constraint. For type safety we must make sure that they do not escape in the return type of the branch or the environment. But, as in the case of rule GLetA, any environment variable is entirely untouchable, which prevents their unification from inside the implication constraint with any type at all, including $\overline{b}$. Once again, this behavior is more strict than the current Haskell implementations of pattern matching against constructors with existential variables, an issue that we return to in Section 5.6.1.

5.5 Solving constraints

We now turn to the internals of the main solver judgement, which has signature

$$\mathcal{Q}\,;\,Q_{given}\,;\,\overline{\alpha}_{tch} \overset{solv}{\mapsto} C_{wanted} \rightsquigarrow Q_{residual}\,;\,\theta$$

In this signature, the *inputs* are:

- The top-level axiom set $\mathcal{Q}$,

- the given (simple) constraints Q_{given} that arise from type annotations (or pattern matching),
- the *touchable* unification variables $\overline{\alpha}_{tch}$ that the solver is allowed to unify, and
- the constraint C_{wanted} that the solver is requested to solve.

The *outputs* are:

- A set of (simple) constraints $Q_{residual}$ that the solver did not solve, and
- a substitution θ, with $dom(\theta) \subseteq \overline{\alpha}_{tch}$.

As before, notice that the solver is not required to always fully discharge C_{wanted} via a substitution for the touchable unification variables; but it may instead return a residual $Q_{residual}$. Notice though that $Q_{residual}$ is a Q-constraint, which means that, at the very least we must have discharged all the implication constraints in C_{wanted}. On the other hand, in the case of annotated bindings (rule BINDA) we have to fully solve the wanted constraint, producing no residual constraints whatsoever.

As before, we assume a constraint simplifier for the underlying constraint domain X. However our constraints C are richer than X: they include implications. Our algorithm provides a single simplifier for C-constraints; this simplifier deals with the implications, and in turn relies on a provided solver for flat X-constraints. This algorithm is given by rule SOLVE in Figure 14. The judgement first appeals to the domain-specific simplifier for the simple part of the constraint $\mathbf{simple}[C]$, producing a residual constraint Q_r and a substitution θ. Subsequently, it applies θ to each of the implication constraints in C. This operation may be simply defined as:

$$\theta(\exists\overline{\alpha}.(Q \supset C) \equiv \exists\overline{\alpha}.(\theta Q \supset \theta C) \qquad \text{where } \overline{\alpha}\#fuv(\theta)$$

The side-condition is not significant algorithmically. The reason is that algorithmically, there is no need for renaming of $\overline{\alpha}$, since $\overline{\alpha}$ cannot possibly appear inside θ (they were generated after all the variables of θ have been generated). Finally we may recursively solve each of the implications having updated the given constraints.

Each constraint C_i in a recursive call to *solve* must be completely solved (which is ensured with the condition $\mathcal{Q}\,;\, Q_g \wedge Q_r \wedge Q_i\,;\, \overline{\alpha}_i \overset{solv}{\mapsto} C_i \rightsquigarrow \epsilon\,;\, \theta_i$ in rule SOLVE). The reason is that the residual constraint returned from *solve* may only be a simple (non-implication) constraint since we are not allowed to quantify over implication constraints. Moreover, the domain of each θ_i only involves internal touchable variables of the implication constraint we are solving, and hence there is no point in returning those θ_i substitutions along with θ in the conclusion of rule SOLVE. Finally, notice the invariant that in all the calls to the solver it is $\overline{\alpha}_{tch}\#fuv(Q_{given})$.

The main solver appeals to a domain-specific simplifier for the X theory, with the following signature:

$$\mathcal{Q}\,;\, Q_{given}\,;\, \overline{\alpha}_{tch} \overset{simp}{\mapsto} Q_{wanted} \rightsquigarrow Q_{residual}\,;\, \theta$$

Its signature is the same as the signature of the simplifier from Figure 7, except for the extra input $\overline{\alpha}_{tch}$ which records the touchable variables – those that may appear in the domain of θ.

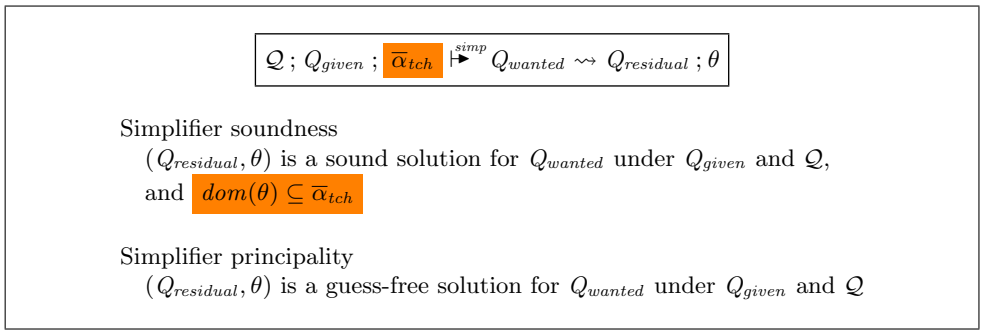

Fig. 15: Touchable-aware simplifier conditions

To describe the desired interface, we may extend the conditions of Figure 8, to the touchable-aware simplifier, highlighting the differences in Figure 15. The conditions are almost unchanged, except for a new highlighted soundness condition which only allows touchables $\overline{\alpha}_{tch}$ in the domain of θ.

5.6 Variations on the design

5.6.1 Design choice: skolem escape checks

Consider the program:

```
data Ex where
 Ex :: forall b. b -> Ex

f = case (Ex 3) of Ex _ -> False
```

Since there is an existential variable introduced by the constructor `Ex`, we would create a (degenerate) implication constraint ($\epsilon \supset \alpha \sim$ `Bool`) where α is the return type of the branch. Using OutsideIn(X), since α is an untouchable variable, the constraint is not solvable, and hence the only way to make this program type check is by adding a type signature to `f :: Bool`. This is somewhat unsatisfactory, since the constructor does not bring any constraints in scope, and hence there is "obviously" only one solution for the right-hand side, namely that $\alpha \mapsto$ `Bool`. Furthermore, one can then easily check this solution to make sure that the existential variable `b` did not escape in the type of the scrutinee nor in the return type of the branch.

A possible extension is this: when the "given" constraints of an implication do not entail any equalities, any "wanted" equalities that do not mention the existential variables can be floated outside the implication. In the example above, we could float out the wanted equality $\alpha \sim$ `Bool`, thereby moving it into the outer scope, where α is touchable. (This strategy is a slight generalisation of the original OutsideIn(X) presentation (Schrijvers *et al.*, 2009), which defined *simple implications* whose given constraints are ϵ.)

Although this extension seems important in practice, and it forms part of our released implementation, it also seems somewhat *ad hoc*, so we have refrained from formalizing it here, instead leaving it as future work.

5.6.2 Design choice: which constraints are really implications

If an implication has originally been generated as ($\alpha \sim$ `Bool` $\supset \beta \sim$ `Int`), it will still be treated as an implication *even* in the presence of another constraint $\alpha \sim$ `Bool`. However, if we had used that information to simplify the givens of the implication constraint, we'd see that we could treat it as a simple constraint $\beta \sim$ `Int`, which would allow the unification of $\beta \mapsto$ `Int`. Here is a concrete example:

```
bar :: forall a. T a -> [a] -> ()
f t xs z = let z1 = bar t xs  -- forces t :: T a, xs :: [a]
               z2 = True:xs   -- generates (a ~ Bool)
               z3 = case t of -- generates (a ~ Bool => b ~ Int)
                     T1 _ -> z + 1
           in ()
```

The code in the definition of `z1` forces `t` to get type `T` α and `xs` to get type $[\alpha]$, where `T` is the GADT from the introduction. The definition of `z2` generates $\alpha \sim$ `Bool`. The case expression generates a constraint $\alpha \sim$ `Bool` $\supset \beta \sim$ `Int` where β is the type of `z`. However, β is not touchable for this implication and hence we can't unify it to anything. If instead we had used the fact that the outer constraint is solvable with $\theta = [\alpha \mapsto$ `Bool`$]$ we could transform the implication to a simple constraint $\beta \sim$ `Int`, which we could solve.

Such a modification is conceivable. We did not follow this path because our current story gives the programmer a purely syntactic understanding of which parts of their programs are treated as implications and which not. Allowing simplifications on the givens to determine which constraints are "really" implications and which not, would make this reasoning potentially more complicated (still, entirely possible).

5.7 Soundness and principality of type inference

We now return to the properties shown in Section 3.6 for the simpler version of our system that did not include local assumptions, and show that the same results are true for the OutsideIn(X) algorithm with respect to the natural type system for local assumptions.

We will assume in this section that the entailment satisfies the conditions of Figure 3 and the simplifier satisfies the conditions of Figure 15. We start with soundness of the OutsideIn(X) algorithm (in analogy with Theorem 3.1).

Theorem 5.1 (Algorithm soundness) *If* $\mathcal{Q} ; \Gamma \vdash\!\blacktriangleright$ *prog then* $\mathcal{Q} ; \Gamma \vdash$ *prog in a closed top-level* Γ.

Proof
Straightforward induction relying on Lemma 5.1. □

This theorem relies on the following auxiliary lemma.

Lemma 5.1 *Assume that* $\Gamma \vdash\!\blacktriangleright e : \tau \rightsquigarrow C$. *Then, for all* C_{ext}, *if* $\mathcal{Q}\,;\, Q_g\,;\, \overline{\beta} \overset{solv}{\vdash\!\blacktriangleright} C \wedge C_{ext} \rightsquigarrow Q_r\,;\theta$ *then there exists* Q *such that* $Q\,;\theta\Gamma \vdash e : \theta\tau$ *and* $\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash Q$.

Proof
By induction on the size of the term e. We consider cases corresponding to which rule was used to derive $\Gamma \vdash\!\blacktriangleright e : \tau \rightsquigarrow C$.

- Case VarCon. We have in this case that $\Gamma \vdash\!\blacktriangleright \nu : [\overline{a \mapsto \alpha}]\tau_1 \rightsquigarrow [\overline{a \mapsto \alpha}]Q_1$ given that $\nu{:}\forall\overline{a}\,.\,Q_1 \Rightarrow \tau_1 \in \Gamma$. Moreover, $\mathcal{Q}\,;\, Q_g\,;\, \overline{\beta} \overset{solv}{\vdash\!\blacktriangleright} [\overline{a \mapsto \alpha}]Q_1 \wedge C_{ext} \rightsquigarrow Q_r\,;\,\theta$. Hence, $\nu{:}\forall\overline{a}\,.\,\theta Q_1 \Rightarrow \theta\tau_1 \in \theta\Gamma$ (without loss of generality assume that $\overline{a}$ do not appear in the domain or range of θ). Consider the substitution $[\overline{a \mapsto \theta\alpha}]$. Then, using rule VarCon and reflexivity of entailment we get $[\overline{a \mapsto \theta\alpha}]\theta Q_1\,;\,\theta\Gamma \vdash \nu : [\overline{a \mapsto \theta\alpha}]\theta\tau_1$, or, equivalently: $[\overline{a \mapsto \theta\alpha}]\theta Q_1\,;\,\theta\Gamma \vdash \nu : \theta([\overline{a \mapsto \alpha}]Q_1)$. By soundness of the simplifier we additionally get $\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash \theta([\overline{a \mapsto \alpha}]Q_1)$ as required.
- Case App. We have that $\Gamma \vdash\!\blacktriangleright e_1\ e_2 : \alpha \rightsquigarrow C_1 \wedge C_2 \wedge (\tau_1 \sim \tau_2 \to \alpha)$ given that $\Gamma \vdash\!\blacktriangleright e_1 : \tau_1 \rightsquigarrow C_1$ and $\Gamma \vdash\!\blacktriangleright e_2 : \tau_2 \rightsquigarrow C_2$. Moreover we know that $\mathcal{Q}\,;\, Q_g\,;\, \overline{\beta} \overset{solv}{\vdash\!\blacktriangleright} C_1 \wedge C_2 \wedge (\tau_1 \sim \tau_2 \to \alpha) \wedge C_{ext} \rightsquigarrow Q_r\,;\,\theta$. Hence, by induction hypothesis there exist Q_1 and Q_2 such that $Q_1\,;\,\theta\Gamma \vdash e_1 : \theta\tau_1$ and $Q_2\,;\,\theta\Gamma \vdash e_2 : \theta\tau_2$, and $\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash Q_1 \wedge Q_2$. By the soundness of the simplifier we additionally have $\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash \theta\tau_1 \sim \theta\tau_2 \to \theta\alpha$. Let $Q = Q_1 \wedge Q_2 \wedge \theta\tau_1 \sim \theta\tau_2 \to \theta\alpha$. It follows that $\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash Q$. Furthermore, it must be that $Q\,;\,\theta\Gamma \vdash e_1 : \theta\tau_1$ and $Q\,;\,\theta\Gamma \vdash e_2 : \theta\tau_2$, which by rule Eq gives $Q\,;\,\theta\Gamma \vdash e_2 : \theta\tau_1 \to \theta\alpha$. Applying rule App gives that $Q\,;\,\theta\Gamma \vdash e_1\ e_2 : \theta\alpha$ as required.
- Case Abs. We have that $\Gamma \vdash\!\blacktriangleright \lambda x\,.\,e : \alpha \to \tau \rightsquigarrow C$, given that $\Gamma, (x{:}\alpha) \vdash\!\blacktriangleright e : \tau \rightsquigarrow C$. Moreover $\mathcal{Q}\,;\, Q_g\,;\, \overline{\beta} \overset{solv}{\vdash\!\blacktriangleright} C \wedge C_{ext} \rightsquigarrow Q_r\,;\,\theta$. By induction hypothesis there exists a Q_1 such that $\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash Q_1$ and $Q_1\,;\,\theta\Gamma, (x{:}\theta\alpha) \vdash e : \theta\tau$. Applying rule Abs gives $Q_1\,;\,\theta\Gamma \vdash \lambda x\,.\,e : \theta(\alpha \to \tau)$ which finishes this case.
- Case Let. Similar to the cases App and Abs.
- Case LetA. Similar to the cases App and Abs.
- Case Case. We have in this case that $\Gamma \vdash\!\blacktriangleright \texttt{case}\ e\ \texttt{of}\ \{\overline{K_i\ \overline{x}_i \to e_i}\} : \beta \rightsquigarrow C' \wedge (\bigwedge C'_i)$ given that

$$\Gamma \vdash\!\blacktriangleright e : \tau \rightsquigarrow C \tag{6}$$

(7)

Moreover, for each branch $K_i\ \overline{x}_i \to e_i$ we have that:

$$K_i{:}\forall\overline{a}\overline{b}\,.\,Q_i \Rightarrow \overline{\upsilon}_i \to \mathtt{T}\ \overline{a} \tag{8}$$

$$\Gamma, (\overline{x_i{:}[\overline{a \mapsto \gamma}]\upsilon_i}) \vdash\!\blacktriangleright e_i : \tau_i \rightsquigarrow C_i \tag{9}$$

$$\text{if } Q_i \neq \epsilon \text{ or } \overline{b} \neq \epsilon \text{ then } C'_i = \exists\overline{\delta}_i\,.\,([\overline{a \mapsto \gamma}]Q_i \supset C_i \wedge \tau_i \sim \beta) \tag{10}$$

$$\text{else } C'_i = C_i \wedge \tau_i \sim \beta$$

Finally, from the assumptions we have that:

$$\mathcal{Q}\,;\, Q_g\,;\, \overline{\beta} \overset{solv}{\vdash\!\blacktriangleright} C \wedge (\mathtt{T}\ \overline{\gamma} \sim \tau) \wedge (\bigwedge C'_i) \wedge C_{ext} \rightsquigarrow Q_r\,;\,\theta \tag{11}$$

Using (6), (11), and the induction hypothesis we get that there exists a Q_e such that

$$Q_e \,;\, \theta\Gamma \vdash e : \theta\tau \tag{12}$$

$$\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash Q_e \tag{13}$$

By the soundness of the simplifier it follows that

$$\mathcal{Q} \wedge Q_g \wedge Q_r \Vdash \mathtt{T}\ \theta\overline{\gamma} \sim \theta\tau \tag{14}$$

Let $Q_e^\star$ be $Q_g \wedge Q_r \wedge Q_{inst}$ where Q_{inst} is the finite set of instances from $\mathcal{Q}$ used in the derivations of (12) and (14). It must be that $Q_e^\star \,;\, \theta\Gamma \vdash e : \theta\tau$ and $Q_e^\star \Vdash \mathtt{T}\ \theta\overline{\gamma} \sim \theta\tau$, and hence by rule EQ also

$$Q_e^\star \,;\, \theta\Gamma \vdash e : \mathtt{T}\ \theta\overline{\gamma} \tag{15}$$

We now need to consider each branch. Branches with both $\overline{b} = \epsilon$ and $Q_i = \epsilon$ are easy and we omit showing the case for those (they are treated essentially as in ABS). Assume now that either $\overline{b} \neq \epsilon$ or $Q_i \neq \epsilon$. By an easy substitutivity lemma for the algorithm and (9) we get that

$$\theta\Gamma, (\overline{x_i{:}[\overline{a \mapsto \theta\gamma}]\upsilon_i}) \mapsto e_i : \theta\tau_i \rightsquigarrow \theta\, C_i \tag{16}$$

In this case, C_i' is an implication constraint, and by the definition of the solver it must be that

$$\mathcal{Q} \,;\, Q_g \wedge [\overline{a \mapsto \theta\gamma}]\, Q_i \wedge Q_r \,;\, \overline{\delta}_i \overset{solv}{\mapsto} \theta\, C_i \wedge \theta\tau_i \sim \theta\beta \rightsquigarrow \epsilon \,;\, \theta_i \tag{17}$$

By (16) and (17) and the induction hypothesis there exists a Q_i' such that $Q_i' \,;\, \theta_i(\theta\Gamma, (\overline{x{:}[\overline{a \mapsto \theta\gamma}]\upsilon})) \vdash e_i : \theta_i\theta\tau_i$. By using (17) and the fact that the simplifier only unifies from $\overline{\delta}_i$, we rewrite this as: $Q_i' \,;\, \theta\Gamma, (\overline{x_i{:}[\overline{a \mapsto \theta\gamma}]\upsilon_i}) \vdash e_i : \theta_i\theta\tau_i$. Moreover, from the induction hypothesis we learn that $\mathcal{Q} \wedge Q_g \wedge Q_r \wedge [\overline{a \mapsto \theta\Gamma}]\, Q_i \Vdash Q_i'$, which in turn means that $Q_e^\star \wedge [\overline{a \mapsto \theta\gamma}]\, Q_i \Vdash Q_i'$. Hence

$$Q_e^\star \wedge [\overline{a \mapsto \theta\gamma}]\, Q_i \,;\, \theta\Gamma, (\overline{x_i{:}[\overline{a \mapsto \theta\gamma}]\upsilon_i}) \vdash e_i : \theta_i\theta\tau_i \tag{18}$$

By (17) we have that $\mathcal{Q} \wedge Q_g \wedge Q_r \wedge [\overline{a \mapsto \theta\gamma}]\, Q_i \Vdash \theta_i\theta\tau_i \sim \theta_i\theta\beta$, and using the fact that the simplifier only unifies from $\overline{\delta}_i$ we have that $Q_e^\star \wedge [\overline{a \mapsto \theta\gamma}]\, Q_i \Vdash \theta_i\theta\tau \sim \theta\beta$. From this, (18), and rule EQ we have that $Q_e^\star \wedge [\overline{a \mapsto \theta\gamma}]\, Q_i \,;\, \theta\Gamma, (\overline{x_i{:}[\overline{a \mapsto \theta\gamma}]\upsilon_i}) \vdash e_i : \theta\beta$. Likewise we can show that each branch (implication or not) is well typed. Hence, with (15) CASE becomes applicable and the case is finished.

- Case GLETA. This case is similar (only simpler) than the case for rule CASE.

□

We have seen that the natural type system for local assumptions in Figure 10 lacks principal types – nevertheless the next theorem asserts that our algorithm accepts only programs with principal types.

Lemma 5.2 (Principality of inferred constraint) *If* $Q\,;\Gamma \vdash e : \tau$ *then* $\Gamma \vdash\!\!\blacktriangleright e : \upsilon \rightsquigarrow C$*, and there exists a* θ *with* $dom(\theta)\#fuv(\Gamma)$ *such that* $Q \Vdash \mathbf{simple}[\theta C]$ *and* $Q \Vdash \theta\upsilon \sim \tau$*.*

Proof
Easy induction. □

Using this property, the corresponding versions of Lemma 3.2 and Theorem 3.2 are proved similarly to the vanilla constraint-based system in Section 3. We repeat the statement of the final theorem.

Theorem 5.2 (Algorithm infers principal types) *If* $fuv(\Gamma) = \emptyset$ *and* $\mathcal{Q};\Gamma \vdash\!\!\blacktriangleright^{gen} e : \sigma_0$ *then for all* σ *such that* $\mathcal{Q}\,;\Gamma \vdash^{gen} e : \sigma$ *it is the case that* $\mathcal{Q} \vdash \sigma_0 \preceq \sigma$ [6]*.*

Proof
Similar to the proof of Theorem 3.2 appealing to Lemma 5.2. □

6 Incompleteness and ambiguity

We now return to the issue of completeness and we introduce yet another problem inherent in most constraint-based type systems, that of *ambiguity*. In the type-system community it is traditional to supply

- A (relatively simple) *declarative specification* of the type system that nails down exactly which programs are well-typed and which are not.
- A (more complicated) *type inference algorithm* that decides whether a given program is well-typed or not.
- A proof that the algorithm is *sound* (if it succeeds, then the program is well-typed according to the specification) and *complete* (if a program is well-typed according to the specification then the inference algorithm succeeds).

The soundness and principality conditions that we have presented so far do not guarantee completeness. Some of the problems have to do with the algorithm accepting *too few programs*, and some with the type system accepting *too many* programs. In this section we explain how the problems with completeness arise, why we do not believe that the traditional approach can succeed, and our approach to resolving the difficulty.

6.1 Incompleteness due to ambiguity

The simplifier soundness and principality conditions are unfortunately not sufficient to guarantee completeness for *annotated* top-level bindings (even in the absence of local constraints), as the type class example below demonstrates.

Example 6.1 (Type class incompleteness, due to ambiguity)

[6] Where $\vdash\!\!\blacktriangleright^{gen}$ refers to the generalization step of rule BIND in Figure 12.

```
show :: forall a. Show a => a -> String
read :: forall a. Show a => String -> a

flop :: String -> String
flop s = show (read s)
```

When type checking the top-level binding `flop`, the constraint solver would be left with a type class constraint `Show` α that it cannot discharge, where α is otherwise unconstrained. However `flop` is typeable in the specification, which simply guesses α to be `Int` or `Bool` – either will do if there exist instance declarations `Show Int` and `Show Bool`. Example 6.1 demonstrates incompleteness associated with the celebrated *ambiguity* problem (Jones, 1992).

Such programs must be rejected, because ambiguity possibly implies that the meaning of a program may be affected by the arbitrary choices the type checker makes, and indeed every type inference algorithm for Haskell does reject such programs. However, the program is accepted by the "natural" type system of Section 3. This is a bug in the type system, but it is not an easy one to fix: we know of no elegant type system that excludes such typings (but see Section 9.6).

Furthermore, even if we were to allow ambiguity, a complete algorithm would have to *search* in the top-level axiom scheme environment for `instance` declarations matching unsolved constraints. Search is undesirable as it may be (i) prohibitively expensive (there may be many interacting choices to be made, so backtracking seems unavoidable), and (ii) contradictory to Haskell's *open world* assumption where the set of declared instances is considered open to extension (from different modules, introduced at link time).

It is worth noting that ambiguity-like problems also arise with equalities involving type families. Suppose there is a wanted constraint $F\ \beta \sim$ `Int` with the top-level axioms F `Int` $\sim$ `Int` and F `Bool` $\sim$ `Int`; then the constraint could be solved with $[\beta \mapsto$ `Int`$]$ or $[\beta \mapsto$ `Bool`$]$, but doing so involves a search. Even if there is only one declared axiom for F, for example F `Int` $\sim$ `Int`, we should not expect the algorithm to deduce that $[\beta \mapsto$ `Int`$]$. Under an open world assumption new axioms could later (at link time) be introduced that no longer justify our choice to make β equal to `Int`. Even more worryingly for completeness, such ambiguous constraints may additionally appear nested inside implication constraints, for example $\exists\beta.(Q \supset \ldots \wedge F\ \beta \sim$ `Int` $\wedge \ldots)$.

6.2 Incompleteness due to inconsistency

Consider the following GADT example.

```
data R a where
 R1 :: (a ~ Int) => a -> R a
 R2 :: (a ~ Bool) => a -> R a

foo :: R Int -> Int
foo x = case x of
```

```
R1 y -> y
R2 y -> False
```

Notice that local assumption introduced in the second branch, matching constructor `R2` amounts to the equality `Int` $\sim$ `Bool`, which is inconsistent. However, nothing prevents the typing rules of Figure 10 to accept this program, if the entailment relation is allowed to use inconsistent assumptions. Inconsistent assumptions like this, operationally, imply that the branch `R2` is unreachable, since it will be impossible to construct evidence of the equality `Int` $\sim$ `Bool`. Hence, the right-hand side of the `R2` branch is dead code. It is therefore type-safe to accept the program, even without requiring that the right-hand side of the `R2` branch is well-typed.

From a software engineering point of view, however, rejecting the program is more desirable than accepting it, for early detection of dead code, and hence an algorithm that exhibits this behavior would necessarily be incomplete.

We could attempt to remedy this situation by requiring in the specification that *every* local assumption introduced is consistent when combined with the top-level axiom schemes. To express this, we extend our definition of top-level consistency from Section 3.3 to deal with local assumptions.

Definition 6.1 (Consistency with local assumptions) *A constraint Q is consistent with respect to $\mathcal{Q}$ iff there exists a ground substitution θ for all the free variables of Q such that whenever $\mathcal{Q} \wedge \theta Q \Vdash \mathtt{T}_1\ \overline{\tau}_1 \sim \mathtt{T}_2\ \overline{\tau}_2$ it is $\mathtt{T}_1 = \mathtt{T}_2$ and $\mathcal{Q} \wedge \theta Q \Vdash \overline{\tau}_1 \sim \overline{\tau}_2$.*

However, in the presence of type family axioms, a simplifier would generally have to perform theorem proving to detect inconsistencies. Consider a function `foo` with signature `Add a (S Z) ~ a => a -> a`, where `Add` is a type family encoding addition. It is clear that the local assumption `Add a (S Z)` $\sim$ `a` is inconsistent (for example consider the ground substitution [`a` $\mapsto$ `Z`]), but an algorithm can only detect this by employing theorem proving techniques. What this means is that, were we to require that the local assumptions be consistent in our specification, the algorithm, unable to always detect inconsistency, would be *unsound* with respect to the specification (i.e. it would accept programs which the specification would reject, such as the definition of `foo`) typing rules.

Our conclusion is this: the specification remain as it stands, accepting some programs with unreachable branches, because we'd rather have an algorithm that is incomplete, rather than an algorithm that is neither complete nor sound with respect to the specification.

6.3 Incompleteness of the OutsideIn(X) *strategy*

Even if we disregard the incompleteness due to ambiguity and inconsistency, there is *still* a completeness gap between the OutsideIn(X) algorithm and the specification of Figure 10. Although the OutsideIn(X) algorithm accepts only programs with principal types in the specification of Figure 10 (Theorem 5.2), it does not accept *all* of them:

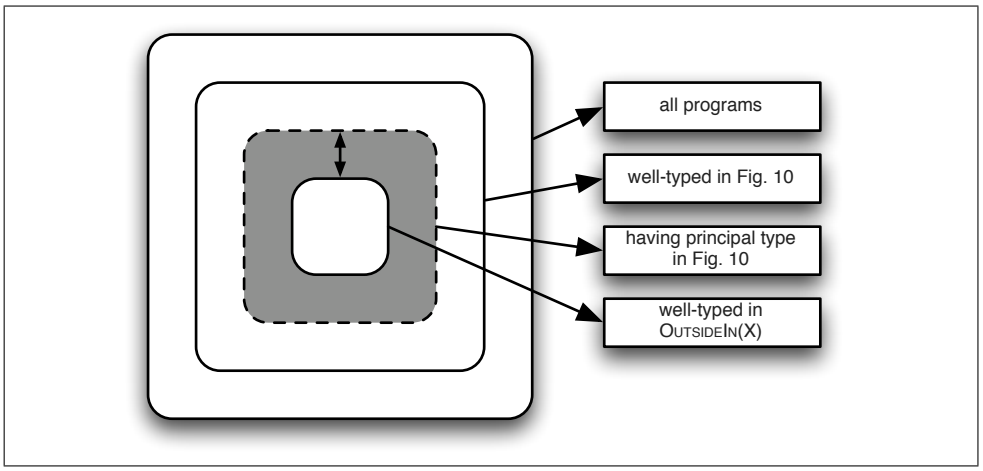

Fig. 16: The space of programs

Example 6.2 (OutsideIn(X) incompleteness) *Consider the following GADT program:*

```
data R a where
  RBool :: (a ~ Bool) => R a

foo rx = case rx of
           RBool -> 42
```

The return type of `foo` can only be `Int` and the program has a principal type, nevertheless the OutsideIn(X) strategy will reject it, because the return type of the branch is not fixed from outside (although it can only be `Int`). In terms of constraints, the constraint arising from the definitionf of `foo` is precisely the one described in Example 5.1, so one might argue that, despite the incompleteness, rejecting the program is more robust in an open world than accepting it.

Overall, the space of typeable programs looks like Figure 16, with a big completeness gap between the programs typed by the algorithm and those accepted by the specification (which accepts *too many* programs).

6.4 Guess-free completeness

Finally, there is yet another potential threat to completeness: perhaps the constraint simplifier does not try hard enough. In particular, the no-op simplifier, which merely returns $Q_{residual} = Q_{wanted}$ and $\theta = \epsilon$, satisfies the conditions of Figure 15. Instantiating our algorithm with the no-op simplifier would be terrible: almost every top-level annotated binding would be rejected! For example:

```
f :: Eq a => a -> Bool
f x = (x == x)
```

We get a $Q_{wanted} = Eq\ a$ and our given constraint is $Q_{given} = Eq\ a$, but the no-op simplifier does not discharge Q_{wanted}!

Happily, in this case we can express an intuitive completeness property: given a consistent set of assumptions, if a constraint can be solved without guessing then the algorithm should solve it. It is easy to formalise what we mean by "guessing":

Definition 6.2 (Guess-free completeness requirement) *A simplifier is guess-free complete iff the following holds for any Q_{wanted}, Q_{given} and $\mathcal{Q}$: If (ϵ, θ) is a guess-free solution of Q_{wanted} under Q_{given}, Q_{given} is consistent with respect to $\mathcal{Q}$, and $\mathcal{Q}$ and $dom(\theta) \subseteq \overline{\alpha}_{tch}$, then the simplifier will return a guess-free solution of the form (ϵ, φ), where $dom(\varphi) \subseteq \overline{\alpha}_{tch}$.*

To see what restrictions this condition imposes on the simplifier, consider first the wanted constraint $\texttt{Eq}\ \alpha$. This constraint does not imply that $[\alpha \mapsto \texttt{Int}]$, and hence a guess-free complete solver is allowed to fail on it (in fact, it *must* fail on it, to satisfy the simplifier principality condition). On the other hand, the constraint $\texttt{Eq}\ \alpha \wedge [\alpha] \sim [\texttt{Int}]$ is solvable by $[\alpha \mapsto \texttt{Int}]$ and a guess-free complete simplifier *must* be able to solve it. The consistency assumption in Definition 6.2 is only making our requirements more realistic, since inconsistent assumptions often can lead solvers to non-termination (we will see how this may happen in Section 7.7), and are extremely difficult or impossible to detect without arbitrarily complex theorem proving (we have seen this already with the `Add` example in Section 6.2).

Unfortunately, Definition 6.2 characterizes algorithms, not type systems. But at the very least, a guess-free complete algorithm clearly rejects all ambiguous programs. Moreover, guess-free completeness can, under some circumstances, give us some completeness guarantees with respect to the natural type system of Figure 10. Assume that a program *prog* is well typed in the type system of Figure 10, and we had a way to add enough annotations on the program that fix all unification variables (for instance, we would need ways to bind the existential variables of constructors, and open type annotations). Let us call the annotated program *prog′*. Moreover, assume that in the typing derivation of *prog′*, all the constraints appearing in the left of $\vdash$ are consistent with respect to $\mathcal{Q}$. If all these conditions are met, and the algorithm satisfies the guess-free completeness requirement then it follows that the annotated program *prog′* will be accepted by the algorithm.

6.5 Our position on incompleteness and ambiguity

To sum up, our specification accepts some "bad" programs (ones that are ambiguous, or lack a principal type), and the OutsideIn(X) algorithm rejects some "good" ones.

The latter is no great surprise. For example, the Hindley-Milner algorithm accepts only λ-abstractions whose binder has a monotype. We accept that a tractable algorithm cannot work magic, so instead we tighten the specification so that it matches what the algorithm can achieve.

The obvious way to restore completeness is is to tighten up the specification, so that it rejects both (a) bad programs and (b) programs that the inference algorithm cannot type. *The trouble is that the cure is worse than the disease*: the specification

becomes as complicated and hard to understand as the algorithm. For one such attempt the reader is encouraged to read our earlier version of the OutsideIn(X) algorithm, which had a fairly complicated specification, and one that worked only for the special case of GADTs in (Schrijvers *et al.*, 2009), and neglected ambiguity entirely. For the general case of arbitrary constraint domains and local constraints, we are not optimistic about this approach.

In this paper we have taken a different tack:

- We give a specification that is simple, general, and comprehensible, but which types too many programs (Section 4). For example, it regards the `read/show` example as well typed.
- We give an inference algorithm that is sound, but not complete, with respect to this specification (Section 5). That is, if the algorithm accepts the program, then the program is indeed well-typed according to the specification, but not vice versa. The absence of completeness is by design: for example, we positively want to reject the `read/show` example, and the examples from Section 2 that lack principal types.
- Although the algorithm is incomplete, we can still offer the following guarantee: if the algorithm accepts a definition, then that definition has a principal type, and that type is what the algorithm finds (Theorem 5.2 in Section 5.7).

A consequence is this: the precise details of which programs are accepted by the specification but rejected by the algorithm is given only by the algorithm itself. While this is unsatisfying in principle, we are willing to live with it for two reasons. First, we know of no better alternative. Second, by explaining that the inference algorithm does not "guess" types, or "search" among possible substitutions, we have found that programmers can, after some experience, accurately predict what should and should not type-check. We offer the whole question as a challenge to others for further work.

7 Instantiating X for GADTs, type classes, and type families

Our general claim is that, the algorithm of Section 5 will infer principal types for an arbitrary underlying constraint domain X, provided:

- the entailment relation of X satisfies the properties of Figure 3,
- a sound and guess-free simplifier is used to solve constraint problems in X.

Kennedy's units of measure is a well-understood and tractable example of just such a domain(Kennedy, 1996). Haskell needs a rather more complicated domain. Indeed, our main purpose was to provide a type inference framework that can accommodate

- multi-parameter type classes
- Generalised Algebraic Data Types (GADTs)
- type families

In this section we describe the entailment relation (Section 7.1) and the simplifier procedure (Section 7.3) for these features. We do not discuss overlapping instances,

Syntactic extensions

τ	$::=$	$\ldots \mid F\ \overline{\tau}$	Type family applications
Q	$::=$	$\ldots \mid \mathtt{D}\ \overline{\tau}$	Type class constraints
$\mathcal{Q}$	$::=$	$Q \mid \mathcal{Q} \wedge \mathcal{Q} \mid \forall \overline{a}\,.\,Q \Rightarrow \mathtt{D}\ \overline{\tau} \mid \forall \overline{a}\,.\,F\ \overline{\xi} \sim \tau$	

Auxiliary syntactic definitions

ζ, ξ	$\in$	$\{\tau \mid \tau$ contains no type families $\}$
$\mathbb{T}$	$::=$	$\mathtt{T}\ \overline{\mathbb{T}} \mid \mathbb{F} \mid \mathbb{T} \rightarrow \mathbb{T} \mid tv \mid \bullet$
$\mathbb{F}$	$::=$	$F\ \overline{\mathbb{T}}$
$\mathbb{D}$	$::=$	$D\ \overline{\mathbb{T}}$

Fig. 17: Syntactic extensions for type classes and type families

implicit parameters, superclasses, or functional dependencies, although our implementation deals with all of these (Section 8). Even so, describing the solver is a fairly challenging task, and this section is a long one – but it is a task that is clearly separable from the rest of the paper.

7.1 The entailment relation

We already have enough syntax to describe GADT equalities, so the required extensions for type classes and type families are given in Figure 17.

The syntax of types is extended with type families of the form $F\ \overline{\tau}$. The syntax of constraints is extended with type class constraints of the form $\mathtt{D}\ \overline{\tau}$. The top-level axioms contain constraints Q as before, and two forms of axiom schemes:

- Class instance axioms, of the form $\forall \overline{a}\,.\,Q \Rightarrow \mathtt{D}\ \overline{\tau}$. Those are brought into $\mathcal{Q}$ by a user instance declaration such as

  ```
  instance Eq a => Eq [a] where ...
  ```

  which gives rise to $\forall a\,.\mathtt{Eq}\ a \Rightarrow \mathtt{Eq}\ [a]$.
- Type family instance declarations, of the form $\forall \overline{a}\,.F\ \overline{\xi} \sim \tau$. Such axiom schemes enter $\mathcal{Q}$ with a `type instance` declaration. For example,

  ```
  type instance F Int = Bool
  type instance F [a] = a
  ```

  gives rise to $F\ \mathtt{Int} \sim \mathtt{Bool} \wedge \forall a\,.F\ [a] \sim a$. GHC enforces that type family instance declarations involve only type families applied to types that contain no type families (type-family-free), and we follow here this restriction (hence $\overline{\xi}$ and not simply $\overline{\tau}$ in type family axiom schemes of Figure 17).

Finally, in the rest of this section we will use $\mathbb{T}$, $\mathbb{F}$, and $\mathbb{D}$ for type, type family, and type class contexts with holes. For example writing $\mathbb{F}[\tau]$ gives a type family application with τ in the hole of $\mathbb{F}$.

Given this syntax of constraints, Figure 18 gives their entailment relation, which

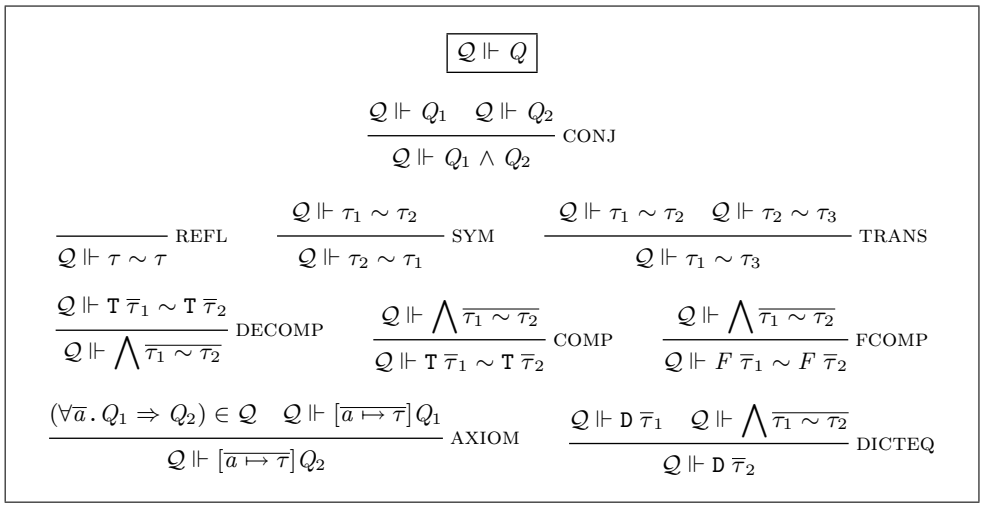

Fig. 18: Concrete entailment

is entirely standard. We have merged the cases for type class instance axioms type family instance axioms in the common rule AXIOM. Notice rule DICTEQ, which allows the rewriting of a type class constraint using a deducible equality.

It is routine induction to confirm that the entailment relation of Figure 18 is well-behaved for type inference purposes.

Lemma 7.1 *The $\Vdash$ relation in Figure 18 satisfies the conditions of Figure 3.*

Proof
Easy induction. □

7.2 Solving equality constraints is tricky

At first it may seem that a constraint solver for type classes, GADTs, and type families is relatively simple. After all, type classes have been with us for twenty years, and we can deal with type families by using the top-level `type instance` declarations as left-to-right rewrite rules. The tricky case comes with local assumptions that involve type families.

Example 7.1 *Consider this contrived example:*

```
type instance F [Int] = Int
type instance G [a]   = Bool

-- Assume  g :: forall b. b -> G b

f :: forall a. (a ~ [F a]) => a -> Bool
f x = g x
```

When type checking `f` we see that, since `(g x)` must return `Bool`, we get the wanted constraint $G\ a \sim \texttt{Bool}$. Using the assumption $a \sim [F\ a]$ we can rewrite the wanted constraint to $G\ [F\ a] \sim \texttt{Bool}$. Aha! Now we can apply the top-level instance for `G` and we are done.

The trick here is that we had to replace a by $[F\ a]$, a process that would go on forever if iterated, so the solver clearly has to be rather careful. Moreover, the assumptions might look like $(F\ a \sim G\ a)$ or even $(F\ (G\ a) \sim G\ a)$, which look nothing like left-to-right rewrite rules, and cannot be used in this way. We discussed these and other issues in an earlier work (Schrijvers *et al.*, 2008a), where we gave a solver for type equalities alone. In the rest of this section we give a solver that is simpler than our earlier one, and incorporates type classes as well.

7.3 The simplifier

We now proceed to the details of a concrete simplifier for the entailment judgement of Figure 18. Our goal is to implement a procedure with the signature:

$$\mathcal{Q}\ ;\ Q_{given}\ ;\ \overline{\alpha}_{tch} \overset{simp}{\mapsto} Q_{wanted} \rightsquigarrow Q_{residual}\ ;\ \theta$$

This procedure will be used to instantiate the solver of Figure 14. Recall that the properties we have postulated for soundness and principality of type inference (Figure 15) give:

$$\mathcal{Q} \wedge Q_{given} \wedge Q_{residual} \Vdash \theta Q_{wanted}$$
$$\mathcal{Q} \wedge Q_{given} \wedge Q_{wanted} \Vdash Q_{residual} \wedge \mathcal{E}_\theta$$

In addition we must have $dom(\theta) \subseteq \overline{\alpha}_{tch}$, $dom(\theta) \# fuv(Q_g, Q_r)$. The way we are going to attack this problem is by discovering a constraint $Q_{residual} \wedge \mathcal{E}_\theta$ that is equivalent to Q_{wanted}, that is, it satisfies:

$$\mathcal{Q} \wedge Q_{given} \Vdash Q_{wanted} \leftrightarrow (Q_{residual} \wedge \mathcal{E}_\theta)$$

Soundness and principality will then follow from this and the observation that $fuv(Q_{given}) \# \overline{\alpha}_{tch}$ at the call sites of the simplifier. Of course, a trivial solution to our problem is to return $\theta = \epsilon$ and $Q_{residual} = Q_{wanted}$, but that would be terrible from a completeness point of view: our golden standard that we aim for is guess-free completeness, described in Section 6.4.

Following previous work (Schrijvers *et al.*, 2008a), we may implement such a simplifier, as the fixpoint of a set of *rewrite rules* that at each step transform our wanted constraint into a simpler, equivalent constraint. Once no more rewriting is possible, we may extract a *substitution* θ for the touchable unification variables from that simplified constraint, and keep the remaining residual constraint, $Q_{residual}$. If the remaining constraint is simply empty then we have managed to *fully solve* our wanted Q_{wanted} by producing a substitution θ. The structure of a simplifier based on this idea appears in Figure 19, and rule SIMPLES implements this strategy.

Rule SIMPLES appeals to the auxiliary judgment $\hookrightarrow$, whose signature is:

$$\mathcal{Q} \mapsto \langle \overline{\alpha}, \varphi, Q_g, Q_w \rangle \hookrightarrow \langle \overline{\alpha}', \varphi', Q_g', Q_w' \rangle_\perp$$

$$\boxed{\mathcal{Q}\,;\, Q_{given}\,;\, \overline{\alpha}_{tch} \overset{simp}{\mapsto} Q_{wanted} \rightsquigarrow Q_{residual}\,;\, \theta}$$

$$\frac{\begin{array}{l}\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \epsilon, Q_g, Q_w\rangle \hookrightarrow^{\star} \langle \overline{\alpha}', \varphi, Q_g', Q_w'\rangle \not\hookrightarrow \\ \quad \varphi Q_w' = \mathcal{E} \wedge Q_r \\ \mathcal{E} = \{\beta \sim \tau \mid ((\beta \sim \tau) \in \varphi Q_w' \text{ or } (\tau \sim \beta) \in \varphi Q_w'), \beta \in \overline{\alpha}', \beta \notin \mathit{fuv}(\tau)\} \\ \theta = [\beta \mapsto \theta\tau \mid (\beta \sim \tau) \in \mathcal{E}] \qquad \overline{\beta} \text{ distinct}\end{array}}{\mathcal{Q}\,;\, Q_g\,;\, \overline{\alpha} \overset{simp}{\mapsto\!\blacktriangleright} Q_w \rightsquigarrow \theta Q_r\,;\, \theta|_{\overline{\alpha}}}\ \text{SIMPLES}$$

$$\boxed{\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi, Q_g, Q_w\rangle \hookrightarrow \langle \overline{\alpha}', \varphi', Q_g', Q_w'\rangle_{\perp}}$$

$$\frac{\mathit{canon}[g]\ (Q_1) = \{\overline{\beta}, \varphi_2, Q_2\}_{\perp}}{\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi_1, Q_g \wedge Q_1, Q_w\rangle \hookrightarrow \langle \overline{\alpha}\overline{\beta}, \varphi_1 \uplus \varphi_2, Q_g \wedge Q_2, Q_w\rangle_{\perp}}\ \text{CANG}$$

$$\frac{\mathit{canon}[w]\ (Q_1) = \{\overline{\beta}, \varphi_2, Q_2\}_{\perp}}{\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi_1, Q_g, Q_w \wedge Q_1\rangle \hookrightarrow \langle \overline{\alpha}\overline{\beta}, \varphi_1 \uplus \varphi_2, Q_g, Q_w \wedge Q_2\rangle_{\perp}}\ \text{CANW}$$

$$\frac{\mathit{interact}[g](Q_1,\ Q_2) = Q_3}{\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi, Q_g \wedge Q_1 \wedge Q_2, Q_w\rangle \hookrightarrow \langle \overline{\alpha}, \varphi, Q_g \wedge Q_3, Q_w\rangle_{\perp}}\ \text{INTG}$$

$$\frac{\mathit{interact}[w](Q_1,\ Q_2) = Q_3}{\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi, Q_g, Q_w \wedge Q_1 \wedge Q_2\rangle \hookrightarrow \langle \overline{\alpha}, \varphi, Q_g, Q_w \wedge Q_3\rangle_{\perp}}\ \text{INTW}$$

$$\frac{(Q)\ \mathit{simplifies}\ (Q_1) = Q_2}{\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi, Q_g \wedge Q, Q_w \wedge Q_1\rangle \hookrightarrow \langle \overline{\alpha}, \varphi, Q_g \wedge Q, Q_w \wedge Q_2\rangle_{\perp}}\ \text{SIMPL}$$

$$\frac{\mathit{topreact}[g](\mathcal{Q},\ Q_1) = \{\epsilon, Q_2\}_{\perp}}{\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi, Q_g \wedge Q_1, Q_w\rangle \hookrightarrow \langle \overline{\alpha}, \varphi, Q_g \wedge Q_2, Q_w\rangle_{\perp}}\ \text{TOPG}$$

$$\frac{\mathit{topreact}[w](\mathcal{Q},\ Q_1) = \{\overline{\beta}, Q_2\}_{\perp}}{\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi, Q_g, Q_w \wedge Q_1\rangle \hookrightarrow \langle \overline{\alpha}\overline{\beta}, \varphi, Q_g, Q_w \wedge Q_2\rangle_{\perp}}\ \text{TOPW}$$

Fig. 19: Main simplifier structure

The purpose of this judgement is to rewrite an input quadruple into an output quadruple. The *inputs* to this judgement are:

- The top-level axioms $\mathcal{Q}$.
- An input quadruple, $\langle \overline{\alpha}, \varphi, Q_g, Q_w\rangle$, which consists of a set of touchable variables, $\overline{\alpha}$, a substitution for some unification variables, φ, a set of *given* constraints Q_g, and a set of *wanted* constraints Q_w. We will refer to the substitution φ as the *flattening substitution* because it undoes the so-called *flattening* operation that we apply to canonicalise constraints; Section 7.4.1 will give the details.

The *output* of this judgement is then a new quadruple, consisting of a new set of

touchable variables, $\overline{\alpha}'$, a new flattening substitution φ', a new set of givens Q'_g as a result of massaging the original givens, and a new set of wanteds Q'_w that record any remaining goals to be shown, that we were not able to deduce. The returned set of touchable variables is always a superset of the input, and will include any new unification variables that have been allocated during simplification (we will see how this may happen in Section 7.4.1, and Section 7.4.4).

Returning to rule SIMPLES, the idea is to repeatedly apply $\hookrightarrow$ (hence $\hookrightarrow^*$) until it no longer applies (hence $\not\hookrightarrow$). Then we massage the results to extract a substitution θ and residual constraint Q_r. The details are best understood after we introduce the constraint rewrite relation, in the next section. We then return to demystify the rest of SIMPLES in Section 7.5.

The $\bot$ symbol in the signature of $\hookrightarrow$ should be understood as syntactic sugar for the possibility that rewriting might *fail*. That is, every rule with a conclusion of the form $\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi, Q_g, Q_w \rangle \hookrightarrow \langle \overline{\alpha}', \varphi', Q'_g, Q'_w \rangle_\bot$ abbreviates *two* rules, one of which simply returns $\bot$ if one of the premises rewrites to $\bot$. For instance, if we encounter the constraint `Int` $\sim$ `Bool` our solver immediately returns $\bot$, instead of keeping the constraint in the quadruple. The possibility of returning $\bot$ amounts to a check for inconsistent constraints. Although such a check is necessarily incomplete (see Section 6.2) we still desire it for three reasons:

- We do not want to quantify over obviously-inconsistent constraints. For example, it would be stupid (although sound) to infer the type (`Int` $\sim$ `Bool`) $\Rightarrow$ `Int` $\rightarrow$ `Bool` for

  ```
  f x = (not x) + 3
  ```

  because `f` could never be called.
- Where possible, we would like to detect unreachable case alternatives, as we discussed in Section 6.2.
- In general, we would like definite errors to be reported as early as possible.

7.4 Rewriting constraints

We turn our attention now to the internals of the judgement

$$\mathcal{Q} \vdash\!\blacktriangleright \langle \overline{\alpha}, \varphi, Q_g, Q_w \rangle \hookrightarrow \langle \overline{\alpha}', \varphi', Q'_g, Q'_w \rangle_\bot$$

given in Figure 19. As we have seen, it transforms quadruples consisting of some touchable variables, a substitution, some given constraints, and some wanted constraints. It does this by appealing to simpler rewrite rules, of four categories:

Canonicalization (Section 7.4.1) is used in rules CANG and CANW. They both call function *canon*, whose signature signature:

$$canon[\ell]\ (Q_1) = \{\overline{\beta}, \varphi_2, Q_2\}_\bot$$

where ℓ is either wanted (w) or given (g). The canonicalisation function transforms a single atomic[7] constraint Q_1 to a simpler form. Constraints that *canon*

[7] An atomic constraint is one that does not involve conjunctions

does not transform are canonical. An example of canonicalization would be to transform $[\alpha] \sim [\texttt{Int}]$ to the simpler form $\alpha \sim \texttt{Int}$. The canonicalization rules may need to create new touchable variables $\overline{\beta}$, or new flattening substitutions φ. Finally notice that those rules can fail returning $\bot$ in which case rule CANG and CANW should also fail returning $\bot$.

Binary interaction (Section 7.4.2) is used in INTG and INTW, which both appeal to the function *interact*:

$$interact[\ell](Q_1,\ Q_2) = Q_3$$

where ℓ can be either given (g) or wanted (w). Interaction combines two atomic constraints (both given or both wanted), producing new wanted or new given constraints, respectively. For example, if we are given two constraints $\alpha \sim \beta$ and $\beta \sim \texttt{Int}$, we would get a new given that $\alpha \sim \texttt{Int}$.

Simplification (Section 7.4.3) is used in rule SIMPL, which invokes

$$(Q)\ \ simplifies\ (Q_1) = Q_2$$

This function uses an atomic given constraint Q to simplify an atomic wanted constraint Q_1, producing a transformed wanted Q_2. It will often be the case that this rule completely discharges the wanted constraint Q_1 producing ϵ. A typical reaction with given $\alpha \sim \texttt{Int}$ and wanted $\alpha \sim \texttt{Int}$ would produce $\texttt{Int} \sim \texttt{Int}$ (which could then be discharged by a canonicalization rule).

Top-level reactions (Section 7.4.4) appear in rules TOPG and TOPW, using function

$$topreact[\ell](\mathcal{Q},\ Q_1) = \{\overline{\beta}, Q_2\}_\bot$$

This function uses the top-level axioms $\mathcal{Q}$ to transform an atomic given or wanted constraint Q_1. For example they may be used to deduce a wanted type class constraint `Eq Int` from an axiom for `Eq Int` introduced by some class instance declaration. We will see that these rules may create new touchable variables.

Our intention is that the rewrite relation induced by these rules is confluent and terminating (under certain conditions on the axiom schemes). Though we do not present a detailed confluence proof, we discuss several design decisions motivated by keeping the algorithm independent of the order that the rules will be applied in the next sections; Section 7.7 discusses termination. For the rest of this section, we will cover a concrete instantiation of those four kinds of rules that give rise to a simplifier for the entailment of Figure 18.

7.4.1 Canonicalization rules

The purpose of canonicalization is to transform a single constraint, given or wanted, to a simpler form. These simple forms that we will be using throughout will be called *canonical constraints*, and are specified in Figure 20 with the judgement

$$\vdash_{can} Q$$

There exist two rules for equality in this figure:

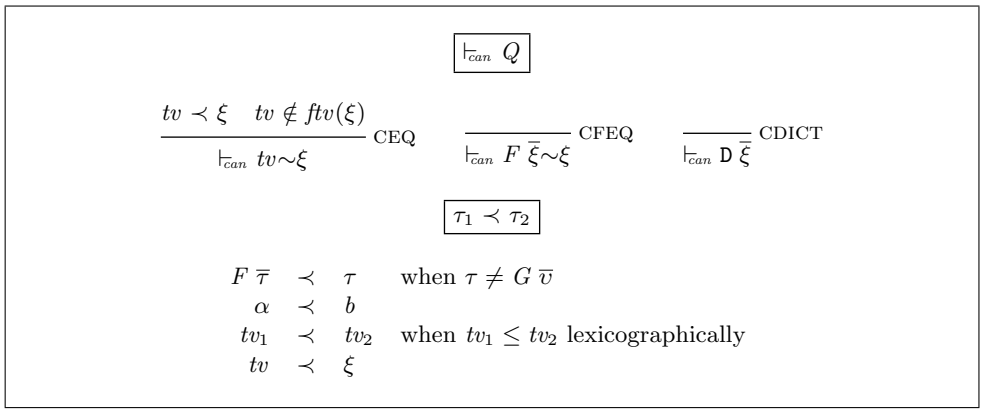

Fig. 20: Auxiliary definitions for canonicalization

$$\boxed{canon[\ell]\ (Q_1) = \{\overline{\beta}, \varphi, Q_2\}_\perp}$$

REFL	$canon[\ell]\ (\tau \sim \tau)$	$=$	$\{\epsilon, \epsilon, \epsilon\}$
TDEC	$canon[\ell]\ (\mathtt{T}\ \overline{\tau}_1 \sim \mathtt{T}\ \overline{\tau}_2)$	$=$	$\{\epsilon, \epsilon, \bigwedge \overline{\tau_1 \sim \tau_2}\}$
FAILDEC	$canon[\ell]\ (\mathtt{T}\ \overline{\tau}_1 \sim \mathtt{S}\ \overline{\tau}_2)$	$=$	$\perp$
OCCCHECK	$canon[\ell]\ (tv \sim \xi)$		
	where $tv \in \xi, \xi \neq tv$	$=$	$\perp$
ORIENT	$canon[\ell]\ (\tau_1 \sim \tau_2)$		
	where $\tau_2 \prec \tau_1$	$=$	$\{\epsilon, \epsilon, \tau_2 \sim \tau_1\}$
DFLATW	$canon[w]\ (\mathbb{D}[G\ \overline{\xi}])$	$=$	$\{\beta, \epsilon, \mathbb{D}[\beta] \wedge (G\ \overline{\xi} \sim \beta)\}$
DFLATG	$canon[g]\ (\mathbb{D}[G\ \overline{\xi}])$	$=$	$\{\epsilon, [\beta \mapsto G\ \overline{\xi}], \mathbb{D}[\beta] \wedge (G\ \overline{\xi} \sim \beta)\}$
FFLATWL	$canon[w]\ (\mathbb{F}[G\ \overline{\xi}] \sim \tau)$	$=$	$\{\beta, \epsilon, (\mathbb{F}[\beta] \sim \tau) \wedge (G\ \overline{\xi} \sim \beta)\}$
FFLATWR	$canon[w]\ (\tau \sim \mathbb{T}[G\ \overline{\xi}])$		
	where $(\tau = F\ \overline{\xi}'$ or $\tau = tv), \beta$ fresh	$=$	$\{\beta, \epsilon, (\tau \sim \mathbb{T}[\beta]) \wedge (G\ \overline{\xi} \sim \beta)\}$
FFLATGL	$canon[g]\ (\mathbb{F}[G\ \overline{\xi}] \sim \tau)$	$=$	$\{\epsilon, [\beta \mapsto G\ \overline{\xi}], (\mathbb{F}[\beta] \sim \tau) \wedge (G\ \overline{\xi} \sim \beta)\}$
FFLATGR	$canon[g]\ (\tau \sim \mathbb{T}[G\ \overline{\xi}])$		
	where $(\tau = F\ \overline{\xi}'$ or $\tau = tv), \beta$ fresh	$=$	$\{\epsilon, [\beta \mapsto G\ \overline{\xi}], (\tau \sim \mathbb{T}[\beta]) \wedge (G\ \overline{\xi} \sim \beta)\}$

Fig. 21: Canonicalization rules

- Rule CEQ asserts that a constraint of the form $tv \sim \xi$ is canonical when when $tv \notin \xi$ – otherwise this must be an occurs check error. We also remind the reader at this point that ξ-types are function free.
- Rule CFEQ asserts that the only constraint that may contain a function symbol should be of the form $F\ \overline{\xi} \sim \xi$. There is no occurs check condition in canonical equalities that involve function symbols. It is perfectly valid to have a constraint of the form $F\ \alpha \sim \alpha$, contrary to, say $\alpha \sim [\alpha]$.

Finally, a canonical type class constraint may also never mention any function symbols and rule CDICT asserts that it is of the form $\mathtt{D}\ \overline{\xi}$.

We now turn to the actual canonicalization rules, in Figure 21. Their purpose is to convert any constraint (given or wanted) to a set of canonical constraints. Rule REFL simply removes a given or wanted reflexive equality. The rest of the rules can be grouped, according to their functionality.

Occurs check Rule OCCCHECK fails in the case where a type variable tv is equal

to a type that may contain the very same variable. Since in rule OCCCHECK the constraint is of the form $tv \sim \xi$, ξ contains no function symbols and hence we are not in danger of raising an erroneous occur check violation for a perfectly valid constraint of the form $a \sim F\ [a]$.

Decomposition rules Rule TDEC *decomposes* an equality between two types with the same head constructor, and rule FAILDEC fails in the case where the head constructors are different.

It is worth noticing that rules FAILDEC (and OCCCHECK) may fail even for *given* constraints. Whereas failure for wanted constraints amount to an unsatisfiable constraint, failure in the given constraint amounts to *inconsistency detection* (see related discussion in Section 6.2). For example, assume the following code:

```
f :: (Bool ~ Char) => Bool -> Char
f x = x && 'c'
```

Since rule FAILDEC applies to both given and wanted constraints, it will result in rejecting `f`.

Orientation We've seen in Figure 20 that canonical equality constraints must have a very particular shape. This means that sometimes equality constraints may need to be oriented to prefer unifiable variables or function applications on the left. This is achieved with rule ORIENT which orients an equality constraint according to the $\prec$ function defined in Figure 20. For example, rule ORIENT will fire for a constraint of the form $a \sim F\ [a]$, to transform it to $F\ [a] \sim a$. Orientation prefers unification variables on the left of equality constraints over skolem variables, but that is just so that the shape of constraints looks more like a substitution, and does not have any deep implications. Finally, for two variables that are both unification variables or skolems, we simply impose an orientation based on the lexicographic ordering of the names of those variables – this has to do with termination and will be explained in Section 7.4.2.

Flattening (and the role of the flattening substitution) We've also seen in Figure 20 that canonical constraints mention function applications only as left-hand sides of equalities. Transforming a constraint to do this is achieved with the *flattening rules* in Figure 21, DFLATW, DFLATG, FFLATWL, FFLATWR, FFLATGL, and FFLATGR. Their behavior is similar: in all cases a fresh variable β is generated and the nested function application is lifted out as an extra equality. For instance, for a constraint:

$$F\ (G\ \texttt{Int}) \sim \texttt{Int}$$

we would get two new constraints:

$$F\ \beta \sim \texttt{Int} \wedge G\ \texttt{Int} \sim \beta$$

Notice though the difference between the *wanted* and *given* cases. If the original constraint is given then we are emitting a new *given* constraint $G\ \overline{\xi} \sim \beta$. But what is the *evidence* that justifies this? This is where the flattening substitutions come into play: we record in the flattening substitution that β *is equal to* $G\ \overline{\xi}$, hence establishing evidence (the identity) that justifies the new given constraint

$G\ \overline{\xi} \sim \beta$. The fresh variable β is carefully *not* recorded as touchable, since we already have a substitution for it.

The role of flattening, often used in completion-based term rewriting (Kapur, 1997), is essential in many dimensions.

- As we will see in Section 7.4.2 and Section 7.4.3 having type family symbols appearing only to the left of equations restricts the number of possible interactions between those equations and other constraints.
- Flat equations involving type functions of the form $F\ \overline{\xi} \sim \xi$ are helpful in ensuring termination of simplification. Recall Example 7.1, and the possibility for non-termination if we simply view the constraint $a \sim [F\ a]$ as a left-to-right substitution. Flattening comes into play here. Instead of viewing $a \sim [F\ a]$ as a left-to-right substitution, we will first *flatten* the equation using introducing a new flatten skolem β to get new givens $a \sim [\beta] \wedge F\ a \sim \beta$ along with the flattening substitution $[\beta \mapsto F\ a]$. In this way we have broken the vicious substitution cycle for variable a. We may now use $a \sim [\beta]$ to rewrite our goal $G\ a \sim \texttt{Int}$ to $G\ [\beta] \sim \texttt{Int}$, which can be readily solved from the top-level axiom for G!

Finally, notice that flattening for *wanted* constraints (rule DFLATW) generates a new wanted goal that $G\ \overline{\xi} \sim \beta$, without binding β in the flattening substitution. Instead, it must record the fresh β variable as touchable – we will bee seeking evidence that $G\ \overline{\xi} \sim \beta$ and that evidence may (but not need to) be found by unifying $\beta \mapsto G\ \overline{\xi}$ later on.

It is very important that we treat the given and wanted case differently. We explain the reasons below:

- Why don't we simply create wanted $G\ \overline{\xi} \sim \beta$ in the given case? Because we can create evidence for $G\ \overline{\xi} \sim \beta$ on the spot, namely by simply setting $[\beta \mapsto G\ \overline{\xi}]$.
- Why don't we create given $G\ \overline{\xi} \sim \beta$ and update the flattening substitution $[\beta \mapsto G\ \overline{\xi}]$ in the wanted case? A superficial reason is that our rules allow canonicalisation of wanteds to wanteds, and of givens to givens, but not canonicalization of wanteds to givens. But the real reason is that $G\ \overline{\xi}$ arises in the context of a wanted constraint, and hence may contain touchable variables. By creating a given $G\ \overline{\xi} \sim \beta$ we are "polluting" our given constraints with touchable variables.

  To see why this is dangerous, consider the original wanted constraints with touchable α:

$$F\ (G\ [\alpha]) \sim \texttt{Int} \tag{19}$$

$$G\ [\alpha] \sim \alpha \tag{20}$$

  along with a given $G\ [\alpha] \sim \gamma$ for an untouchable γ. Suppose that constraint (19) canonicalizes to $F\ \beta \sim \texttt{Int}$ (wanted), and $G\ [\alpha] \sim \beta$ (given), with the flattening substitution being $[\beta \mapsto G\ [\alpha]]$. From these and the given $G\ [\alpha] \sim \gamma$ we can deduce the given constraint $\gamma \sim \beta$. To solve (20), and using the given $G\ [\alpha] \sim \beta$ it suffices to solve $\alpha \sim \beta$. One may think that $\alpha \sim \beta$ can be readily

$$\boxed{\mathit{interact}[\ell](Q_1,\ Q_2) = Q_3}$$

EQSAME	$\mathit{interact}[\ell](tv \sim \xi_1,\ tv \sim \xi_2)$ where $\vdash_{can} tv \sim \xi_1, \vdash_{can} tv \sim \xi_2$	$=$	$(tv \sim \xi_1) \wedge (\xi_1 \sim \xi_2)$
EQDIFF	$\mathit{interact}[\ell](tv_1 \sim \xi_1,\ tv_2 \sim \xi_2)$ where $\vdash_{can} tv_i \sim \xi_i, tv_1 \in \mathit{ftv}(\xi_2)$	$=$	$(tv_1 \sim \xi_1) \wedge (tv_2 \sim [tv_1 \mapsto \xi_1]\tau_2)$
EQFEQ	$\mathit{interact}[\ell](tv \sim \xi_1,\ F\ \overline{\xi} \sim \xi)$ where $\vdash_{can} tv \sim \xi_1, tv \in \mathit{ftv}(\overline{\xi}, \xi)$	$=$	$(tv \sim \xi_1) \wedge (F\ [tv \mapsto \xi_1]\overline{\xi} \sim [tv \mapsto \xi_1]\xi)$
EQDICT	$\mathit{interact}[\ell](tv \sim \xi,\ \mathtt{D}\ \overline{\xi})$ where $\vdash_{can} tv \sim \xi, tv \in \mathit{ftv}(\overline{\xi})$	$=$	$(tv \sim \xi_1) \wedge (\mathtt{D}\ [tv \mapsto \xi]\overline{\xi})$
FEQFEQ	$\mathit{interact}[\ell](F\ \overline{\xi} \sim \xi_1,\ F\ \overline{\xi} \sim \xi_2)$	$=$	$(F\ \overline{\xi} \sim \xi_1) \wedge (\xi_1 \sim \xi_2)$
DDICT	$\mathit{interact}[\ell](\mathtt{D}\ \overline{\xi},\ \mathtt{D}\ \overline{\xi})$	$=$	$\mathtt{D}\ \overline{\xi}$

Fig. 22: Binary interaction rules

solved by setting $[\alpha \mapsto \beta]$, but that does not work because β is already unified to $G\ [\alpha]$ and hence we will get a non-idempotent unifier $[\alpha \mapsto G\ [\alpha]]$! Alas, we could instead have tried to solve $\alpha \sim \gamma$, which surprisingly *is* solvable by setting $[\alpha \mapsto \gamma]$ since we have a given constraint $\gamma \sim \beta$.
To summarize, the presence of touchable variables in the givens makes the algorithm sensitive in the order that the rewrite rules fire.

7.4.2 Binary interaction rules

Binary interaction rules transform two *canonical* constraints that are either both given or both wanted to a new given or wanted constraint, respectively. Figure 22 gives the details.

Rule EQSAME reacts two equalities with the same variable on the left-hand side, producing a new equality equating the right-hand sides.

Rule EQDIFF reacts canonical equalities but where one of the left-hand side variables appears in the right-hand-side type of the other equality. This rule is the reason for requiring a lexicographic ordering between variables in the syntax of the canonical constraints. Consider the example:

$$\alpha \sim \beta \wedge \beta \sim \gamma \wedge \gamma \sim \beta$$

where all variables are touchables. After one step of rewriting using EQDIFF we may substitute β in the first constraint and end up with:

$$\alpha \sim \gamma \wedge \beta \sim \gamma \wedge \gamma \sim \beta$$

Now, we may substitute again in the first constraint variable γ to get

$$\alpha \sim \beta \wedge \beta \sim \gamma \wedge \gamma \sim \beta$$

which is *the original constraint* we started with. We see that there is a danger for non-termination. Imposing a lexicographic ordering of variables and reacting only canonical ones would mean that it would be impossible to have both $\beta \sim \gamma$ *and* $\gamma \sim \beta$ able to react with another constraint using EQDIFF.

The returned constraint from an EQDIFF interaction could possibly violate an

occurs check condition. Such a violation will be detected by a canonicalization reaction OCCCHECK later on. For instance:

$$\alpha \sim [\beta] \wedge \beta \sim [\alpha]$$

may react to $\alpha \sim [[\alpha]] \wedge \beta \sim [\alpha]$. Rule EQDIFF does not apply to this new constraint, since the equality $\alpha \sim [[\alpha]]$ is not canonical. Canonicalization will detect and report the occurs check violation.

Rule EQFEQ reacts a canonical equality with a function equality, and rule EQDICT reacts a canonical equality with a type class constraint. In both cases the equality is used to rewrite the other constraint. For instance, `Eq` $\alpha \wedge \alpha \sim$ `Int` rewrites to `Eq Int`. Rule EQFEQ rewrites a type family equality using an equality. Finally, rule FEQFEQ reacts two canonical function equalities, which have the *same* left-hand sides.

Rule DDICT deals with interacting two identical type class constraints. A situation with two identical class constraints may well happen, both in the given and in the wanted case. Consider the code below:

```
data T a where
  K :: Eq a => a -> T a

f :: Eq a => T a -> a -> Bool
f (K y) x = (x == y)
```

In the body of `f`, we see that `Eq a` is available from the type signature, but *also* from the pattern matching against `K`. Moreover, the code `(x == y)` gives rise to a wanted constraint `Eq a`. We could resolve the wanted constraint from *one of the two givens*, but which one? We can think about this problem in terms of evidence: Since each class constraint is associated with a runtime dictionary equipped with operations, such a situation amounts to the presence of two dictionaries for the same type. The semantics of Haskell requires dictionaries of the same type to be contextually equivalent and hence it does not matter which of the two we will choose to solve the wanted constraint `Eq a`. Hence, rule DDICT drops one of two identical *given* constraints. On the other hand, if we have two identical wanted type class constraints in our goal we may simply try to solve one of them – if we have evidence for one, this will immediately be evidence for the other as well.

Finally, observe that the ability to rewrite *both* given and wanted constraints is essential:

Reacting givens Assume that we have two given constraints $F\ a \sim$ `Int` $\wedge\ F\ a \sim a$ and a wanted constraint F `Int` $\sim a$. How can we possibly solve the wanted constraints? By first reacting the givens, to get $a \sim$ `Int`. Then we may rewrite the givens again to get F `Int` $\sim$ `Int` as given, which can then be used to solve the wanted goal, in a way that will be described in the next section.

Reacting wanteds Assume that we have two wanted equations $F\ \gamma \sim \gamma \wedge F\ \gamma \sim$ `Int`. If we react them we may get that $\gamma \sim$ `Int` $\wedge\ F\ \gamma \sim$ `Int`, which can react again to get F `Int` $\sim$ `Int`, which in turn may be solvable by a top-level axiom for F.

$$\boxed{(Q_1)\ \textit{simplifies}\ (Q_2) = Q_3}$$

SEQSAME	$(tv \sim \xi_1)$ *simplifies* $(tv \sim \xi_2)$		
	where $\vdash_{can} tv \sim \xi_i$	$=$	$\xi_1 \sim \xi_2$
SEQDIFF	$(tv_1 \sim \xi_1)$ *simplifies* $(tv_2 \sim \xi_2)$		
	where $\vdash_{can} tv \sim \xi_i, tv_1 \in \xi_2$	$=$	$tv_2 \sim [tv_1 \mapsto \xi_1]\tau_2$
SEQFEQ	$(tv \sim \xi_1)$ *simplifies* $(F\ \overline{\xi} \sim \xi)$		
	where $\vdash_{can} tv \sim \xi_1, tv \in ftv(\overline{\xi})$	$=$	$F\ [tv \mapsto \xi_1]\overline{\xi} \sim \xi$
SEQDICT	$(tv \sim \xi)$ *simplifies* $(\mathtt{D}\ \overline{\xi})$		
	where $\vdash_{can} tv \sim \xi, tv \in \overline{\xi}$	$=$	$\mathtt{D}\ [tv \mapsto \xi]\overline{\xi}$
SFEQFEQ	$(F\ \overline{\xi} \sim \xi_1)$ *simplifies* $(F\ \overline{\xi} \sim \xi_2)$	$=$	$\xi_1 \sim \xi_2$
SDDICTG	$(\mathtt{D}\ \overline{\xi})$ *simplifies* $(\mathtt{D}\ \overline{\xi})$	$=$	ϵ

Fig. 23: Simplification rules

7.4.3 Simplification rules

The simplification rules resemble the binary interaction rules; they are however directional: one of the constraints is always a wanted and the other a given. The simplification rules are given in Figure 23.

The reader can confirm that these rules are simple variants of the binary interaction rules we have already seen previously in Figure 22. However some rules are *missing*. For example, there is no rule with a type family equality on the left (as a given) and an ordinary equality on the right (as a wanted):

$$(F\ \overline{\xi} \sim \xi)\ \textit{simplifies}\ (tv \sim \xi_1) = \ldots$$

If there were such a rule, what could be in place of ...? It is tempting to produce a new wanted constraint $(tv \sim \xi_1) \wedge (F\ [tv \mapsto \xi_1]\overline{\xi} \sim [tv \mapsto \xi_1]\xi)$. This is certainly sound but produces an entirely new goal, which – after all – is already deducible if we can solve our original goal $tv \sim \xi_1$. Polluting our constraints with such useless goals seems dangerous for termination, and leads to larger constraints to quantify over when we are inferring types for expressions. On the other hand, we certainly cannot produce a new *given* constraint $(F\ [tv \mapsto \xi_1]\overline{\xi} \sim [tv \mapsto \xi_1]\xi)$ since the evidence of that constraint will rely on the wanted evidence for $tv \sim \xi_1$.

More generally, allowing given constraints to contain evidence from wanted constraints can easily make the algorithm sensitive to the order in which the rewrite rules fire. Consider the given constraint $F\ a \sim \mathtt{Int}$ and wanted constraint $F\ a \sim \mathtt{Int}$. If, instead of deciding to simplify the wanted using the given, we chose to simplify the given using the wanted, we'd get a new given $\mathtt{Int} \sim \mathtt{Int}$ and a remaining wanted goal of $F\ a \sim \mathtt{Int}$ which would now be unsolvable.

7.4.4 Top-level reaction rules

Finally, we reach the top-level reaction rules. These rules apply top-level axioms for type families or type classes and are given in Figure 24. In the case of a wanted type class constraint (rule DINSTW) we produce new wanted goals using the instance

$$\boxed{\mathit{topreact}[\ell](\mathcal{Q},\ Q_1) = \{\overline{\beta}, Q_2\}_\perp}$$

$$\frac{\begin{array}{c}\forall \overline{a}\,.\,Q \Rightarrow \mathtt{D}\ \overline{\xi}_0 \in \mathcal{Q} \\ \overline{b} = \mathit{ftv}(\overline{\xi}_0) \quad \overline{c} = \overline{a} - \overline{b} \quad \overline{\gamma}\ \text{fresh} \quad \theta = [\overline{b \mapsto \xi_b}, \overline{c \mapsto \gamma}] \quad \theta\overline{\xi}_0 = \overline{\xi}\end{array}}{\mathit{topreact}[w](\mathcal{Q},\ \mathtt{D}\ \overline{\xi}) = \{\overline{\gamma}, \theta Q\}}\ \textsc{dinstw}$$

$$\frac{\forall \overline{a}\,.\,Q \Rightarrow \mathtt{D}\ \overline{\xi}_0 \in \mathcal{Q} \quad \theta = [\overline{a \mapsto \xi_a}] \quad \theta\overline{\xi}_0 = \overline{\xi}}{\mathit{topreact}[g](\mathcal{Q},\ \mathtt{D}\ \overline{\xi}) = \perp}\ \textsc{dinstg}$$

$$\frac{\begin{array}{c}\forall \overline{a}\,.\,F\ \overline{\xi}_0 \sim \xi_0 \in \mathcal{Q} \\ \overline{b} = \mathit{ftv}(\overline{\xi}_0) \quad \overline{c} = \overline{a} - \overline{b} \quad \overline{\gamma}\ \text{fresh} \quad \theta = [\overline{b \mapsto \tau_a}, \overline{c \mapsto \gamma}] \quad \theta\overline{\xi}_0 = \overline{\xi} \\ \text{if } (\ell = w) \text{ then } \overline{\delta} = \overline{\gamma} \text{ else } \overline{\delta} = \epsilon\end{array}}{\mathit{topreact}[\ell](\mathcal{Q},\ F\ \overline{\xi} \sim \xi) = \{\overline{\delta}, \theta\xi_0 \sim \xi\}}\ \textsc{finst}$$

Fig. 24: Top-level reaction rules

declaration. Suppose that we have an axiom scheme $\forall a\,.\,\mathtt{Eq}\ a \Rightarrow \mathtt{Eq}\ [a]$ in $\mathcal{Q}$. This may react with a wanted `Eq [Bool]` as follows:

$$\mathit{topreact}[w](\mathcal{Q},\ \mathtt{Eq}\ [\mathtt{Bool}]) = \{\epsilon, \mathtt{Eq}\ \mathtt{Bool}\}$$

We get a new wanted constraint `Eq Bool`. Notice that new unification variables ($\overline{\gamma}$) may be introduced. These variables should be considered touchables for simplification purposes.

In the case of a given type class constraint (rule DINSTG), we disallow application of matching top-level axioms, so we simply return $\perp$. The reason behind this reaction, which is reminiscent of an *overlapping instance* check, is to keep the simplifier strategy simple. Consider the following situation:

```
class D a where
   d :: a -> Bool
instance C a => D [a] where ...

f :: forall a. D [a] => [a] -> Bool
f x = d x
```

The resulting wanted constraint `D [a]` in the body of `f` could be discharged either by the local given constraint `D [a]` from the signature, or by using the top-level axiom scheme $\forall a\,.\,\mathtt{C}\ a \Rightarrow \mathtt{D}\ [a]$. However, since there is no instance for `C a` available, accidentally preferring the top-level axiom scheme would mean that `f` would be rejected. Hence, the solving algorithm is non-deterministic.

We may attempt to improve the situation by preferring reaction with local given constraints over reaction with top-level axioms. Yet, this does not resolve the issue altogether. Consider this example:

```
instance P x  => Q [x] where ...
instance x ~ y => R [x] y
```

```
-- Assume  wob :: forall a b. (Q [b], R b a) => a -> Int

g :: forall a. Q [a] => [a] -> Int
g x = wob x
```

From `g` we get the implication constraint $\mathtt{Q}\ [a] \supset (\mathtt{Q}\ [\beta] \wedge \mathtt{R}\ \beta\ [a])$. At this point, we can only react with one of the two top-level axioms. If we choose to react ($\mathtt{Q}\ [\beta]$) with the first one, we end up with ($\mathtt{P}\ \beta$), which we have no way of discharging. If, in contrast, we react with the second top-level axiom, we get $\mathtt{Q}\ [\beta] \wedge \beta \sim a$. After substituting the equality in the type class constraint (with rule EQDICT), we obtain the wanted constraint $\mathtt{Q}\ [a]$ that is readily discharged with the local given. Hence, even if we defer applying top-level axioms as long as possible, the behavior of the solving algorithm remains non-deterministic. This might be fixable by refraining from applying DINSTW if a local given *could* match with the wanted constraint (perhaps after instantiating unification variables), but it is all getting rather complicated.

Our compromise, in favor of simplicity and determinism, is to reject all situations where a given constraint overlaps with a top-level type class axiom scheme, and that is what DINSTG says.

For type family equations, independently of whether they are given or wanted, we may rewrite them by looking for a top-level type family instance that matches (rule FINST). This is possible because evidence construction under type family instance reductions works in both directions (Sulzmann *et al.*, 2007a). For instance, if we are given an axiom $\forall a\,.\,F\ [a] \sim a$ in $\mathcal{Q}$ and a wanted constraint $F\ [\mathtt{Bool}] \sim \beta$ then we may have the reaction:

$$\mathit{topreact}[w](\mathcal{Q},\ F\ [\mathtt{Bool}] \sim \beta) = \{\epsilon, \mathtt{Bool} \sim \beta\}$$

As in the case for wanted class constraints, rule FINST must return new touchable variables in the wanted case ($\overline{\delta}$).

7.5 The rule SIMPLES

We now return to the heart of the simplifier, rule SIMPLES, which we repeat below:

$$\frac{\begin{array}{ll}(1) & \mathcal{Q} \vdash\!\!\blacktriangleright \langle \overline{\alpha}, \epsilon, Q_g, Q_w\rangle \hookrightarrow^{\star} \langle \overline{\alpha}', \varphi, Q_g', Q_w'\rangle \not\hookrightarrow \\ (2) & \varphi Q_w' = \mathcal{E} \wedge Q_r \\ (3) & \mathcal{E} = \{\beta \sim \tau \mid ((\beta \sim \tau) \in \varphi Q_w' \text{ or } (\tau \sim \beta) \in \varphi Q_w'), \beta \in \overline{\alpha}', \beta \notin \mathit{fuv}(\tau)\} \\ (4) & \theta = [\beta \mapsto \theta\tau \mid (\beta \sim \tau) \in \mathcal{E}] \quad \overline{\beta} \text{ distinct}\end{array}}{\mathcal{Q}\ ;\ Q_g\ ;\ \overline{\alpha} \vdash\!\!\blacktriangleright^{\mathit{simp}} Q_w \rightsquigarrow \theta Q_r\ ;\ \theta|_{\overline{\alpha}}}\ \text{SIMPLES}$$

Rule SIMPLES first rewrites a constraint as much as possible with condition (1), using the rewrite relation of the previous section. The output quadruple $\langle \overline{\alpha}', \varphi, Q_g', Q_w'\rangle$ contains an extended set of touchables $\overline{\alpha}'$, and a flattening substitution φ. Condition (2) applies the flattening substitution to the residual constraint Q_w', to obtain $\varphi Q_w'$. The flattening substitution cannot mention any touchable unification variables in its domain or range, as the canonicalization rules reveal. The constraint $\varphi Q_w'$ is then

syntactically split into an equational part $\mathcal{E}$ in conjunction with a constraint Q_r, so that $\mathcal{E}$ satisfies certain properties. The properties that $\mathcal{E}$ must satisfy are given with conditions (3) and (4). The constraint $\mathcal{E}$ must contain constraints of the form $(\beta \sim \tau)$, drawn from $\varphi Q'_w$, such that β is a touchable variable not in the free unification variables of τ. We then require that there exists an idempotent substitution induced by $\mathcal{E}$, which we write with the "lazy" notation: $\theta = [\beta \mapsto \theta\tau \mid (\beta \sim \tau) \in \mathcal{E}]$. Once the substitution θ is extracted, we return the appropriate restriction of θ to the original touchables $\overline{\alpha}$ along with θQ_r as the residual constraint, since soundness requires that $dom(\theta) \subseteq \overline{\alpha}$ and that $dom(\theta) \# fuv(\theta Q_r)$.

To see an example of the operations in rule SIMPLES, suppose that, after the flattening substitution has been applied, our constraint $\varphi Q'_w$ looks like:

$$F\ \texttt{Int}\ \beta \wedge F\ \texttt{Int}\ \gamma \wedge \delta \sim \gamma$$

where β, γ, δ are all touchable variables. Observe that this constraint is indeed normal with respect to the rewrite rules. We may then take $\mathcal{E} = (\beta \sim F\ \texttt{Int}) \wedge (\gamma \sim F\ \texttt{Int}) \wedge (\delta \sim \gamma)$ from which we can extract the idempotent substitution $\theta = [\beta \mapsto F\ \texttt{Int}, \gamma \mapsto F\ \texttt{Int}, \delta \mapsto F\ \texttt{Int}]$. The residual constraint in this case is just the empty constraint ϵ.

The equational constraint $\mathcal{E}$ need not contain *all* equations that involve touchable unification variables. Consider the following constraint $\varphi Q'_w$:

$$F\ \texttt{Int} \sim \beta \wedge F\ \texttt{Char} \sim \beta$$

where β is touchable. Once again, this constraint is normal with respect to the rewrite relation. We can pick $\mathcal{E} = (\beta \sim F\ \texttt{Char})$ (but not $\mathcal{E} = (\beta \sim F\ \texttt{Char}) \wedge (\beta \sim F\ \texttt{Int})$, since $\overline{\beta}$ have to be distinct), and $\theta = [\beta \mapsto F\ \texttt{Char}]$. In this case the residual constraint will be $F\ \texttt{Char} \sim F\ \texttt{Int}$, which we may quantify over, provided we are operating in inference mode.

7.6 Soundness and principality

We now show that rewriting preserves constraint equivalence, as expected.

Lemma 7.2 *If* $\mathcal{Q} \mapsto \langle \overline{\alpha}, \varphi, Q_g, Q_w \rangle \hookrightarrow \langle \overline{\alpha}', \varphi', Q'_g, Q'_w \rangle_{\perp}$ *and* $\overline{\alpha} \# fuv(Q_g)$ *and* $dom(\varphi) \subseteq \overline{\alpha}$ *then:*

1. $\mathcal{Q} \Vdash \varphi Q_g \leftrightarrow \varphi' Q'_g$, *and*
2. $\mathcal{Q} \wedge \varphi' Q'_g \Vdash \varphi Q_w \leftrightarrow \varphi' Q'_w$

Moreover $\overline{\alpha}' \supseteq \overline{\alpha}$, $dom(\varphi') \subseteq \overline{\alpha}'$, *and* $\overline{\alpha}' \# fuv(Q'_g, Q_g)$.

Proof
This can be done with a simple case analysis on the rewrite rules. □

From this it is a small step to conclude that the simplifier satisfies the conditions for soundness and principality.

Theorem 7.1 *The simplifier of Figure 19 satisfies the conditions of Figure 15, when called with touchables disjoint from the given constraint.*

What about our "gold standard" of guess-free completeness under consistent assumptions, described in Section 6.4? Because of rule DINSTG our simplifier does not meet that definition. For example, the entailment relation can be used to deduce that

$$(\forall a\,.\,\mathtt{C}\ [a]) \wedge \mathtt{C}\ [\mathtt{Int}] \Vdash \mathtt{C}\ [\mathtt{Int}]$$

but our simplifier will fail due to rule DINSTG. Related to this is the issue of overlapping top-level instances, which our algorithm also does not detect. Consider for example

$$(\forall a\,.\,\mathtt{C}\ [a]) \wedge (\forall a\,.\,\mathtt{E}\ a \Rightarrow \mathtt{C}\ [a]) \Vdash \mathtt{C}\ [\mathtt{Int}]$$

The simplifier would non-deterministically attempt to use one of the two top-level axioms, but only one of two would work, as there is no way to discharge `E Int`.

Excluding such overlapping definitions in the specification is possible though somewhat heavy. We could require that no given class constraint, or no instance of top-level axiom, *matches* an instance of another top-level axiom, but the following example demonstrates that the correct condition is trickier than that naïve approach. Consider the top-level axiom schemes

$$\mathcal{Q} = (\forall a\,.\,\mathtt{D}\ [a]) \wedge (\forall a\,.\,F\ [a] \sim [a])$$

and a local given constraint $Q = \mathtt{D}\ (F\ [a])$. Notice that no instance of top-level axiom exactly matches Q but, rather, there exist an instance of a top-level axiom scheme that can be *rewritten* to Q using the top-level axioms.

Hence we give the following revised definition of consistency (Definition 6.1).

Definition 7.1 (Revised non-overlapping consistency) *A constraint $\mathcal{Q} \wedge Q$ is non-overlapping consistent iff*

- *It satisfies Definition 6.1, and*
- *For all ground substitutions θ_g, such that $\mathcal{Q} \wedge \theta_g Q$ satisfies Definition 6.1, we have:*
  1. *If $\mathtt{D}\ \overline{\tau} \in \theta_g Q$ then for all axiom schemes $\forall \overline{a}\,.\,Q \Rightarrow \mathtt{D}\ \overline{\xi} \in \mathcal{Q}$ there is no substitution for $\overline{a}$, φ, such that $\mathcal{Q} \wedge \theta_g Q \Vdash \varphi(\overline{\xi}) \sim \overline{\tau}$, and*
  2. *If $(\forall \overline{a}\,.\,Q \Rightarrow \mathtt{D}\ \overline{\xi}) \in \mathcal{Q}$ and $\mathtt{D}\ \overline{\tau}$ is a ground instantiation of this axiom then for all* other *axiom schemes $(\forall \overline{a}'\,.\,Q' \Rightarrow \mathtt{D}\ \overline{\xi}') \in \mathcal{Q}$ there is no substitution φ, such that $\mathcal{Q} \wedge \theta_g Q \Vdash \varphi(\overline{\xi}') \sim \overline{\tau}$.*

We conjecture that our simplifier is complete for the notion of consistency of Definition 7.1, but have not formally carried out the proof.

Conjecture 7.1 *The simplifier of Figure 19 is guess-free complete (Definition 6.2) for the notion of consistency given in Definition 7.1.*

7.7 Termination

The $\hookrightarrow$ judgement always terminates in the absence of top-level axioms $\mathcal{Q}$, but it is obvious that top-level axioms threaten termination. After all, if we permitted the programmer to write

```
type instance F a = F a
```

then we could hardly expect the type inference engine to terminate. So the question becomes: what restrictions on the top-level axioms suffice to guaranteed termination?

For equality axiom schemes, here are sufficient (albeit quite restrictive) conditions identified in previous work (Schrijvers *et al.*, 2008a).[8]

Definition 7.2 (Termination Conditions for Equality Axiom Schemes) *An equality axiom scheme* $\forall \bar{a}\ .\ F\ \bar{\xi} \sim \tau$ *satisfies the termination conditions iff* τ *is either of the form* ξ*, or it is of the form* $G\ \bar{\xi}'$ *such that*

1. *the sum of the number of datatype constructors and schema variables in the right-hand side is smaller than the similar sum in the left-hand side, and*
2. *the right-hand side has no more occurrences of any schema variable than the left-hand side.*

These conditions are quite restrictive. For example, they rule out the following declaration:

```
type instance F [x] = [F x]
```

The declaration appears perfectly reasonable but, if permitted, it can cause the algorithm to diverge. Consider this type signature:

```
f :: (F [a] ~ a) => ...
```

This signature will give rise to a given constraint $[F\ \ a] \sim a$. Canonicalization reorients the constraint and flattens it to $a \sim [\beta] \wedge F\ \ a \sim \beta$. Binary interaction of the first with the second constraint yields $a \sim [\beta] \wedge F\ [\beta] \sim \beta$. Note that the latter constraint is a variant of the original one, and the process starts again from scratch; it does not terminate.

Follow-up work (Schrijvers *et al.*, 2008a) introduces more relaxed conditions, which do not always guarantee termination, but have a completeness trade-off.[9] As the intricacies of termination are not the central topic of this paper, we refer the reader to previous work for more details. We are not aware of any related work that detects non-termination under more relaxed conditions without compromising completeness.

We apply similar restrictions on type class schemes, also based on previous work (Sulzmann *et al.*, 2007b).

Definition 7.3 (Termination Conditions for Type Class Axiom Schemes) *An equality axiom scheme* $(\forall \bar{a}\,.\,Q \Rightarrow \mathtt{D}\ \bar{\xi})$ *satisfies the termination conditions iff* Q *is a conjunction of type class constraints* `D'` $\bar{\xi}'$ *such that*

1. *the sum of the number of datatype constructors and schema variables in* $\bar{\xi}'$ *is smaller than the similar sum in* $\bar{\xi}$*, and*

[8] where it is called the *Strong Termination Condition*.
[9] Completeness, in the sense of how many constraints can be fully discharged by the solver.

2. $\overline{\xi}'$ *has no more occurrences of any schema variable than* $\overline{\xi}$.

Note that equality constraints are not allowed to occur in type class scheme contexts; nor are open type families. So, while equality schemes may affect both equality and type class constraints, type class schemes only affect type class constraints.

We conjecture that termination is preserved with somewhat more liberal conditions that do allow type equality constraints in type class scheme contexts.

Definition 7.4 (Conjectured Termination Conditions) *An equality axiom scheme* $(\forall \overline{a}\,.\,Q \Rightarrow \texttt{D}\ \overline{\xi})$ *satisfies the termination conditions iff* Q *is either a conjunction of type class constraints* `D'` $\overline{\xi}'$ *such that*

1. *the sum of the number of datatype constructors and schema variables in* $\overline{\xi}'$ *is smaller than the similar sum in* $\overline{\xi}$*, and*
2. $\overline{\xi}'$ *has no more occurrences of any schema variable than* $\overline{\xi}$.

or it is an equality constraint whose type variables also occur in $\overline{\xi}$ *and it either has the form* $F\ \overline{\xi}' \sim \tau$ *satisfying the Termination Conditions for Equality Axiom Schemes or the form* $\xi_1 \sim \xi_2$.

The restriction on the equality constraints is essential to avoid a scenario similar to the problematic one above:

```
instance (F [x] ~ [F x], F [x] ~ x, C x) => C [x]

f :: (C [b]) => ...
```

The first type equality constraint in the type class instance context clearly does not satisfy the above condition. Consider the constraint `C [b]` in the type signature of `f`. The instance scheme reduces it to $F\ [b] \sim [F\ b] \wedge F\ [b] \sim b \wedge \texttt{C}\ b$. Canonicalization reorients and flattens the first constraint to $F\ [b] \sim [\beta] \wedge F\ b \sim \beta$. Binary interaction of the first of the flattened constraint with the other one yields, after canonicalisation, $b \sim [\beta]$. Interaction with the remaining type class constraints yields `C` $[\beta]$, which is a variant of the original type class constraints, leading to divergence.

Proving (or disproving) that Definition 7.4 rules out all cases of non-termination is an important challenge for future work.

In conclusion, we can see to *guarantee* termination we are forced to impose quite restrictive conditions on top-level axioms. In practice, it may be more attractive to drop these conditions (perhaps selectively), and instead accept non-termination, much as we do for executable programs themselves.

8 Implementation

We have fully implemented OutsideIn(X) in a released version of the Glasgow Haskell Compiler (GHC, release 7.0). GHC supports many type system extensions beyond those described here, including: scoped type variables, kind signatures, unboxed types, type-class defaults, higher rank types, impredicative polymorphism,

associated types, overlapping instances, functional dependencies, and implicit parameters. Happily, the interaction between these features and the OutsideIn(X) constraint solver is minimal, with the notable exception of functional dependencies and implicit parameters. Even they were accommodated without much trouble, although the details are beyond the scope of this paper.

8.1 Evidence

When solving type-class constraints, GHC's type checker transforms the program by adding extra "evidence parameters" to overloaded functions, and extra "evidence arguments" to applications of such functions. For example, consider

```
square :: Num a => a -> a
square x = x * x
```

then the type checker will add an implicit parameter to `square`, and an implicit argument to the call of `(*)`, so that the definition is transformed to this:

```
square :: Num a -> a -> a
square d x = (*) d x x
```

Here `d` is a tuple of the methods of the `Num` class.

GHC extends this evidence idea to *all* constraints. When solving a constraint, it generates an evidence term that encodes the proof, and decorates the program with this evidence term, a process known as *elaboration*. This decoration is not *ad hoc*: GHC translates the original implicitly-typed Haskell program into an *explicitly-typed* program in a well-defined core language FC_2 (Sulzmann *et al.*, 2007a; Weirich *et al.*, 2010), an extension of System F. Typechecking FC_2 is easy and fast, involving none of the complexities of this paper. In a perfect world there would be no point in type-checking the intermediate program, but in practice many, many compiler bugs are caught by such a check.

8.2 Brief sketch of the implementation

GHC's implementation is heavily influenced by the need to generate evidence, but the code still directly reflects the structure of the solver described in this paper:

- A single pass generates constraints, just as described in Section 5.4. The type checker deals with a very large source language: the syntax tree has dozens of data types and hundreds of constructors. As a result, the constraint generator has many lines of code, but it is mostly very simple. Moreover it required very little alteration when we switched to OutsideIn(X). For example, the bidirectional propagation of type information needed to support higher-rank inference remains untouched (Peyton Jones *et al.*, 2007).
- The main practical refinement to the constraint generator is that it performs on-the-fly unification using side effects, just like a conventional Damas-Milner type inference algorithm (Peyton Jones *et al.*, 2007). In the vastly common

Lines of:	Code	Comments
Constraint generation		
`TcAnnotations.lhs`	31	19
`TcArrows.lhs`	199	149
`TcBinds.lhs`	550	651
`TcClassDcl.lhs`	342	281
`TcDefaults.lhs`	61	41
`TcDeriv.lhs`	704	817
`TcExpr.lhs`	667	742
`TcForeign.lhs`	212	143
`TcGenDeriv.lhs`	1,075	922
`TcHsType.lhs`	539	507
`TcInstDcls.lhs`	453	748
`TcMatches.lhs`	389	277
`TcPat.lhs`	474	589
`TcRnDriver.lhs`	952	695
`TcRules.lhs`	61	69
`TcSplice.lhs`	649	662
`TcTyClsDecls.lhs`	872	748
`TcTyDecls.lhs`	102	225
TOTAL	**8,332**	**8,285**
Constraint solver		
`TcSimplify.lhs`	562	632
`TcInteract.lhs`	771	1,166
`TcUnify.lhs`	591	776
`TcCanonical.lhs`	433	505
TOTAL	**2,357**	**3,079**
Infrastructure (type definitions, monads, error reporting)		
`Inst.lhs`	269	259
`FamInst.lhs`	105	108
`TcEnv.lhs`	392	346
`TcErrors.lhs`	615	307
`TcHsSyn.lhs`	737	370
`TcType.lhs`	784	697
`TcMType.lhs`	825	805
`TcSMonad.lhs`	640	414
`TcRnTypes.lhs`	432	536
`TcRnMonad.lhs`	721	452
TOTAL	**5,520**	**4,294**
GRAND TOTAL	**16,209**	**15,658**

Fig. 25: GHC's type checker

case, this unifier can solve the equality constraint on the spot, but if it has any difficulty – for example, if a type family is involved or if it is asked to unify a variable not belonging in the current set of touchable variables – it

bales out by adding a new, unsolved, equality constraint to the accumulating set of constraints.

- A single module, `TcSimplify`, solves implication constraints, using the generic algorithm described in Section 5.5.
- It in turn needs to solve flat constraints, so here the solver has to know the specifics of the constraints (Section 7). The first step is to canonicalise each constraint, as described in Section 7.4.1, implemented in `TcCanonical`. The structure of canonical constraints is so useful for the rest of the solver that GHC defines a separate data type for them, used only internally in the solver.
- Then the canonical constraints are solved using the pairwise interaction rules of Section 7.4.2. The interaction solver, implemented by `TcInteract`, works by maintaining an *inert set* of (canonical) constraints that have no pairwise interactions, and a *work set* of un-processed (canonical) constraints. We repeatedly take a constraint from the work set, interact it with each member of the inert set, and add any "reaction products" back to the work set.

Figure 25 summarizes the line count for the type checker. The constraint-generation code is voluminous but simple. The constraint solver is much smaller (albeit still substantial), and is insensitive to new source-language constructs. The higher ratio of comments to code in the constraint solver reflects its relative subtlety.

9 Related work

There has been a significant volume of related work on constraint-based type systems and type inference for advanced type system features, a fragment of which we discuss in the rest of this Section.

9.1 Constraint-based type inference

There is a very long line of work in Hindley-Milner derivatives, parameterized over various constraint domains (Jones, 1992; Sulzmann, 2000; Odersky *et al.*, 1999; Sulzmann *et al.*, 1999). Rémy and Pottier (2005) give a comprehensive account of type inference for HM(X) (Hindley-Milner, parameterized over the constraint domain X). To our knowledge, our presentation is the first one that deals with local assumptions introduced by type signatures and data constructors, and where those local assumptions may include type equalities. At the same time, a drawback of our system is that it does not handle local let-generalization, an essential ingredient of HM(X).

Simonet and Pottier (2007) study type inference for GADTs, where local GADT type equalities may be introduced as a result of pattern matching. They propose a solution that does generalization over local `let`-bound definitions, by abstracting over the full generated constraint. We have seen that this approach has practical disadvantages, though theoretically appealing and technically straightforward. Interestingly, since ML is call by value the constraints arising from a `let`-bound definition have to be satisfiable by *some* substitution, since the expression will be

evaluated independently of whether it will be called or not. By contrast, in Haskell we may postpone the satisfiability check of the generated constraints all the way to the call sites of a definition. In the case of our previous work on type inference for GADTs (Schrijvers *et al.*, 2009) such a satisfiability check happens implicitly since at local `let`-bound definitions, the constraint generation procedure calls the solver to discharge the generated constraints by means of substitutions.

The pioneering work of Mark Jones on qualified types (Jones, 1992) is closely related to our approach, except for the fact that we additionally have to deal with type equalities and local assumptions.

9.2 The special case of GADTs

In the special case of GADTs, there has been a flurry of papers on inference algorithms to support them in a practical programming language. 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 (Cheney & Hinze, 2003) and (Simonet & Pottier, 2007).

On the other hand, type inference for unannotated (or partially annotated) GADT programs turns out to be extremely hard. The difficulty lies in the fact that GADT pattern matches bring into scope local type assumptions. Following the standard route of reducing type inference to constraint solving, GADTs require implication constraints to capture the inference problem precisely (Sulzmann *et al.*, 2008). Unification is no longer sufficient to solve such constraints. We require more complicated solving methods such as constraint abduction (Maher, 2005) and E-unification (Gallier *et al.*, 1992). It is fairly straightforward to construct examples which show that no principal solutions (and therefore no principal types) exist.

How do previous inference approaches tackle these problems? Apart from the aforementioned work of Simonet and Pottier (2007), Sulzmann *et al.* (2008) go the other direction, by keeping constraints (in types) simple, and instead apply a very powerful abductive solving mechanism, inspired by Maher's work (Maher, 2005). Compared to our simplifier principality conditions, Figure 15, Sulzmann *et al.* propose a more powerful form of constraint abduction and introduce the weaker fully maximally general condition where $Q_{given} \wedge Q_{wanted}$ and $Q_{given} \wedge Q_{residual}$ are equivalent. Simplified principal solutions $Q_{residual} \wedge \mathcal{E}_\theta$ are fully maximally general but the other direction does not hold in general. For example, consider $Q_{given} = (\alpha \sim \texttt{Bool})$ and $Q_{given} = (b \sim \texttt{Bool})$. Then, $Q_{residual} = (\alpha \sim b)$ is fully maximally general but violates the simplifier principality conditions.

9.2.1 Practical compromises for GADT type inference

Since tractable type inference for completely unannotated GADT programs is impossible, it becomes acceptable to demand a certain amount of user-provided type information. We know of three well-documented approaches:

Régis-Gianas and Pottier (2006) 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. Régis-Gianas and Pottier present two different shape analysis procedures, the W and Z systems. The W system has similar expressiveness and need for annotation as in (Peyton Jones *et al.*, 2006). The Z 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 implication constraint. In terms of expressiveness, the vast majority of programs typeable by our system are typeable in Z but we conjecture that there exist programs typeable in our system not typeable in Z, because unification of simple (global) constraints may be able to figure more out about the types of expressions than the preprocessing shape analysis of Z. On the other hand, Z lacks a declarative specification that does not force the programmer to understand the intricacies of shape propagation.

Peyton Jones *et al.* (2006) 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. Because rigidity analysis is more aggressive in our system we type many more programs than in (Peyton Jones *et al.*, 2006), including the carefully-chosen Example 7.2 from (Pottier & Régis-Gianas, 2006). 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 (Peyton Jones *et al.*, 2006). Arguably, though, this program *should* be rejected, because there are several incomparable types for `funny_id` (in the unrestricted system of Figure 10), 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[10]. 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`.

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

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.

Moreover, in both approaches the programmer is required to understand an entirely new concept (shape or rigidity respectively), with somewhat complex and *ad hoc* rules (e.g. Figure 6 of (Pottier & Régis-Gianas, 2006)). Nor is the implementation straightforward; e.g., GHC's implementation of (Peyton Jones *et al.*, 2006) is known to be flawed in a non-trivial way.

Lin and Sheard (2010a) recently presented *point-wise* GADTs, a type system for GADTs where unification in GADT pattern matching is replaced by a unidirectional matching procedure between the scrutinee type and the data constructor type. The authors claim that limiting the power of unification in GADT pattern matching increases predictability, and gives rise to more intuitive behavior and error messages.

9.3 The special case of multi-parameter type classes

Sulzmann *et al.* (2006a) describe an implication solver for multi-parameter type classes. In the multi-parameter type class setting, local constraints may only include type classes and not type equations. The implication solver described in (Sulzmann *et al.*, 2006a) is more powerful and uses a form of abduction to infer the missing constraints to solve implication constraints. The consequence is that Sulzmann *et al.* require stronger conditions imposed on type classes to guarantee that their solver yields principal solutions if successful.

9.4 Solving equalities involving type families

Our solver is based on ideas from *completion* and *congruence closure* in term-rewriting systems (Kapur, 1997; Bachmair & Tiwari, 2000; Beckert, 1994; Nieuwenhuis & Oliveras, 2005), and is an improvement and simplification of previous work by Schrijvers *et al.* (Schrijvers *et al.*, 2008a). That work presented a completion-based solver, where the top-level set of axioms was transformed to a strongly normalizing and confluent rewrite system, along with the current given equations. Our algorithm streamlines the completion, achieved by flattening, decomposing, and orienting, in the actual solving procedure and hence it provides a more uniform approach to the problem. Moreover the details of flattening and canonical constraints slightly differ between the two papers. Yet another difference is that the simplifier in this paper is aware of the touchable variables, which significantly simplifies the previous treatment of unification variables. Finally, this paper gives the complete story of how the simplifier for type families plugs-in in a general-purpose constraint-based type system with local assumptions, required for the OutsideIn(X) approach.

Finally, closely related to this work is the Chameleon system described in (Sulzmann *et al.*, 2006b). Chameleon makes use of the Constraint Handling Rules (CHR)

formalism (Frühwirth, 1998) for the specification of type class relations potentially involving functional dependencies. CHR is a committed-choice language consisting of constraint rewrite rules. Via CHR rewrite rules we can model open type functions. As in the solver of this paper (but not in (Schrijvers *et al.*, 2008a)), the CHR type inference strategies (Stuckey & Sulzmann, 2005) mixes completion and solving. On the other hand, whereas our solver is designed with evidence generation in mind, the issue of evidence generation has not been fully addressed for CHR yet.

9.5 Let generalization for units of measure and type families

Kennedy's system of units of measure (Kennedy, 1996) was briefly sketched in Section 1, and demonstrates related problems with generalizing `let`-bound definitions. Lacking qualified types, Kennedy adopted **NoQual** (Section 4.2.4) but then, quite unexpectedly, discovered a type inference completeness problem:

```
div :: forall u1 u2.
         num (u1*u2) -> num u1 -> num u2

main x = let f = div x
         in (f this, f that)
```

In the program above, `div` is a typed division function. Let us assume that `x` gets type `num u` in the environment, for some unknown `u`. From type checking the body of `f` we get the constraint $\mathtt{u} \sim \mathtt{u1} * \mathtt{u2}$ for some unknown instantiations of `div`. If the unifier naïvely substitutes `u` away for $\mathtt{u1} * \mathtt{u2}$, those variables become bound in the environment, and hence we are not allowed to apply `f` polymorphically in the body of the definition.

Kennedy found a technical fix, by exploiting the fact that units of measure happen to form an Abelian group, and adapting an algebraic normalization procedure to types. For example, the normal form type for `f` above is:

```
forall u. num u -> num (u/u1)
```

His technique is ingenious, but leads to a significant complexity burden in the inference algorithm. More seriously, it does not generalise because it relies on special algebraic properties of units of measure. Naturally, his solution fails for arbitrary type functions.

Kennedy's problematic situations are encodable in our case through the use of type families. Even worse, in the case of type families case **NoQual** is a no-option for one extra reason: the order of solving the constraints may affect typeability. Consider:

```
type instance F Int b = b
let f x = (let h y = e1 in 42, x + 42)
```

Assume that `x` is assigned a unification variable α and `y` is assigned β respectively, and assume that `e1` yields the constraint $F\ \alpha\ \beta \sim \mathtt{Int}$. If we first attempt to solve this constraint to generalize `h`, we will simply fail, as we can't quantify over it. On

the other hand, if we first solve the constraint from `x + 42`, we may learn that $\alpha \mapsto$ `Int`, which can then be used to rewrite the constraint from `e1` to F `Int` $\beta \sim$ `Int`. This in turn is solvable by using the top-level axiom, and `h` is perfectly well-typed! Since **NoQual** is a no-option and quantifying freely means we have to defer unifications, we see that type families even further necessitate our abandoning of local `let` generalization.

An interesting possibility would be to allow programmers to extend the constraint canonicalization rules with domain-specific algebraic normalization procedures, but we have not carried out this experiment.

9.6 Ambiguity

There has been work on addressing the ambiguity problem, described earlier in Section 6, by imposing conditions on the types or the typing derivations of a constraint-based type system (Nipkow & Prehofer, 1995). Jones (1992) identified a type of the form $\forall \overline{a} \,.\, Q \Rightarrow \tau$ as unambiguous, iff all quantified variables $\overline{a}$ appear in τ. His proposal has been to reject those programs whose more general type is ambiguous, and seems to work for qualified types, in the absence of local constraints, type signatures, and type families.

In the presence of more complex constraints and local type signatures, this definition is no longer sufficient. Stuckey and Sulzmann (2005) employ a more elaborate ambiguity condition than Jones.

Definition 9.1 (Stuckey-Sulzmann unambiguous types) *A type $\forall \overline{a} \,.\, Q \Rightarrow \tau$ is unambiguous in $\mathcal{Q}$, iff for some fresh set of variables $\overline{b}$ in bijection with $\overline{a}$ we have that*

$$\mathcal{Q} \wedge Q \wedge ([\overline{a \mapsto b}]Q) \wedge (\tau \sim [\overline{a \mapsto b}]\tau) \Vdash \overline{a \sim b}$$

In other words, the equality between two instantiated types implies equality of instantiations.

For example, consider the type: $\forall ab \;.\; F\ a \sim b \Rightarrow$ `Int` $\rightarrow a$ and a renaming $[a \mapsto a_1, b \mapsto b_1]$. Then we must show that

$$F\ a \sim b \wedge F\ a_1 \sim b_1 \wedge a \sim a_1 \Vdash (a \sim a_1) \wedge (b \sim b_1)$$

One important reason for extending Jones' definition is that there even exist entirely constraint-free types that are ambiguous. Take for example $\forall a \;.\; F\ a \rightarrow$ `Int`. Assuming a renaming $[a \mapsto a_1]$, it does not follow that:

$$F\ a_1 \sim F\ a \Vdash a_1 \sim a$$

as type functions need not be injective. In practical terms, this means that we can never apply a function with that type to a value of type, say, F `Int`. To get completeness by avoiding ambiguity, Stuckey and Sulzmann require that *every subexpression* in their type system must have a principal type which is unambiguous. Such a condition is rather heavyweight as it involves a quantification over all possible types of every subexpression in the program; but it's the only condition we know of in the

literature that effectively eliminates ambiguity. Adapting it to our setting is not entirely straightforward as many subexpressions, due to local assumptions, *do not have* principal quantified types and we would probably have to employ, on top of those conditions, an Outside-In flavored type system (Schrijvers *et al.*, 2009). It remains therefore open whether it is possible to restrict the type system specification of Figure 10 so that we obtain sound and complete type inference.

Finally, ambiguity seems also related to the discussion about type functions and type classes being defined under an "open" or "closed" world assumption (Sulzmann, 2000). For example, if the set of axioms was considered fixed (closed world) one could consider more elaborate search-based strategies that would more effectively detect ambiguity.

9.7 Is the emphasis on principal types well-justified?

Although from a software engineering viewpoint principal types in the natural type system are extremely useful, they may be less desirable from a program correctness viewpoint – an issue that we have seen mentioned in (Lin & Sheard, 2010b)). For instance, consider the program below.

```
data R a where
  RInt  :: Int  -> R Int
  RBool :: Bool -> R Bool
  RChar :: Char -> R Char

flop1 (RInt x) = x
```

Function `flop1` can be assigned two types: $\forall a\,.\,\texttt{R}\ a \rightarrow a$ or $\forall a\,.\,\texttt{R}\ a \rightarrow \texttt{Int}$ neither of which is an instance of the other; however, there is a third type that is arguably more desirable, and that type is $\texttt{R Int} \rightarrow \texttt{Int}$. The reason that this type is more desirable for `flop` is because it rules out applications of `flop` to values of type `R Bool` or `R Char`, which would result in runtime errors.

Though this is valid for `flop1`, what happens in the following variation?

```
flop2 (RInt x)  = x
flop2 (RBool x) = x
```

Via the same reasoning, the most desirable type for `flop2` would be a type like $\texttt{R}\ a \rightarrow a$ but where a must be *constrained* to be either `Int` or `Bool`. Unfortunately, ordinary polymorphic types are too weak to express this restriction and we can only get $\forall a\,.\,\texttt{R}\ a \rightarrow a$ for `flop2`, which does *not* rule the application of `flop` to values of type `R Char`. In conclusion, giving up on some natural principal types in favor of more specialized types that eliminate more pattern match errors at runtime is appealing but does not quite work unless we consider a more expressive syntax of types. Furthermore it is far from obvious how to *specify* these typings in a high-level declarative specification.

10 Future Work

We have already implemented a variation of the algorithm and the solver described in this paper and we plan to integrate our implementation in GHC.

Our ambitious plan is to eventually support extensibility of a type inference system that supports local assumptions with new forms of constraints and interactions with each other. The implication constraint infrastructure sets the ground for this but there remain many open problems to address in future work such as how to combine multiple constraint domains, how to specify language extensions and have existing library code use them, and how to ensure type safety.

Acknowledgements We would like to thank the anonymous JFP reviewers for their insightful comments and Brent Yorgey for carefully proofreading a draft of this article. Many thanks to the Haskell Wiki users who provided us with useful feedback to the online version of the paper at

`http://www.haskell.org/haskellwiki/Simonpj/Talk:OutsideIn`

References

Bachmair, Leo, & Tiwari, Ashish. (2000). Abstract congruence closure and specializations. *Pages 64–78 of: Proceedings of CADE'00*. LNCS, vol. 1831. Springer-Verlag.

Beckert, Bernhard. (1994). A completion-based method for mixed universal and rigid E-unification. *Pages 678–692 of: Proceedings of CADE'94*. Springer-Verlag.

Chakravarty, M., Keller, Gabriele, & Peyton Jones, S. (2005a). Associated type synonyms. *Pages 241–253 of: Proceedings of ICFP'05*. New York, NY, USA: ACM Press.

Chakravarty, M., Keller, Gabriele, Peyton Jones, S., & Marlow, S. (2005b). Associated types with class. *Pages 1–13 of: Proceedings of POPL'05*. ACM Press.

Cheney, J., & Hinze, R. (2003). *First-class phantom types*. TR 1901. Cornell University. `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901`.

Damas, Luis, & Milner, Robin. (1982). Principal type-schemes for functional programs. *Pages 207–212 of: Proceedings of POPL'82*. New York, NY, USA: ACM Press.

Faxén, Karl-Filip. (2003). Haskell and principal types. *Pages 88–97 of: Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. New York, NY, USA: ACM Press.

Frühwirth, Thom. (1998). Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, **37**(1–3), 95–138.

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

Hall, C. V., Hammond, K., Peyton Jones, S. L., & Wadler, P. L. (1996). Type classes in Haskell. *Acm TOPLAS*, **18**(2), 109–138.

Heeren, Bastiaan, Leijen, Daan, & van IJzendoorn, Arjan. (2003). Helium, for learning Haskell. *Pages 62 – 71 of: ACM SIGPLAN 2003 Haskell Workshop*. New York: ACM Press.

Jones, M. P. 1992 (Sept.). *Qualified types: Theory and practice*. D.phil. thesis, Oxford University.

Jones, Mark P. 1993 (September). *Coherence for qualified types.* Research Report YALEU/DCS/RR-989. Yale University, New Haven, Connecticut, USA.

Jones, Mark P. (2000). Type classes with functional dependencies. *Proceedings of ESOP 2000.* Lecture Notes in Computer Science, no. 1782. Springer-Verlag.

Kapur, Deepak. (1997). Shostak's congruence closure as completion. *Pages 23–37 of: Proceedings of RTA'97.* London, UK: Springer-Verlag.

Kennedy, AJ. 1996 (Sept.). *Type inference and equational theories.* LIX RR/96/09. Ecole Polytechnique.

Kiselyov, Oleg, Peyton Jones, Simon, & Shan, Chung-chieh. (2010). Fun with type functions. Roscoe, Bill (ed), *Tony Hoare 75th birthday Festschrift.* Springer History of Computing.

Lämmel, Ralf, & Peyton Jones, Simon. (2003). Scrap your boilerplate: A practical design pattern for generic programming. *Pages 26–37 of: Proceedings of TLDI 2003.* ACM Press.

Lämmel, Ralf, & Peyton Jones, Simon. (2005). Scrap your boilerplate with class: extensible generic functions. *Pages 204–215 of: Proceedings of ICFP'05.* New York, NY, USA: ACM Press.

Läufer, K., & Odersky, M. (1994). Polymorphic type inference and abstract data types. *Acm Transactions on Programming Languages and Systems*, **16**(5), 1411–1430.

Lin, Chuan-kai, & Sheard, Tim. (2010a). Pointwise generalized algebraic data types. *Pages 51–62 of: Proceedings of TLDI '10.* New York, NY, USA: ACM.

Lin, Chuan-kai, & Sheard, Tim. 2010b (July). *Three techniques for GADT type inference.* draft.

Maher, M. (2005). Herbrand constraint abduction. *Pages 397–406 of: Proceedings of LICS'05.* IEEE Computer Society.

Milner, Robin. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**(3), 348–375.

Nieuwenhuis, Robert, & Oliveras, Albert. (2005). Proof-producing congruence closure. *Pages 453–468 of: Proceedings of RTA'05.* LNCS, vol. 3467. Springer-Verlag.

Nipkow, Tobias, & Prehofer, Christian. (1995). Type reconstruction for type classes. *Journal of Functional Programming*, **5**(2), 201–224.

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

Peyton Jones, S., Washburn, G., & Weirich, S. 2004 (July). *Wobbly types: type inference for generalised algebraic data types.* Tech. rept. MS-CIS-05-26. University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389.

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Washburn, G. (2006). Simple unification-based type inference for GADTs. *Pages 50–61 of: Proceedings of ICFP'06.* ACM Press.

Peyton Jones, S., Vytiniotis, D., Weirich, S., & Shields, M. (2007). Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, **17**(Jan.), 1–82.

Pottier, F., & Régis-Gianas, Y. (2006). Stratified type inference for generalized algebraic data types. *Pages 232–244 of: Proceedings of POPL'06.* ACM Press.

Pottier, F., & Rémy, D. (2005). The essence of ML type inference. *Chap. 10, pages 389–489 of:* Pierce, Benjamin C. (ed), *Advanced topics in types and programming languages.* MIT Press.

Schrijvers, T., Sulzmann, M., Peyton Jones, S., & Chakravarty, M. (2007). Towards open type functions for Haskell. *Pages 233–251 of:* Chitil, O. (ed), *Proceedings of IFL'09.*

Schrijvers, Tom, Peyton Jones, Simon, Chakravarty, Manuel, & Sulzmann, Martin.

(2008a). Type checking with open type functions. *Pages 51–62 of: Proceedings of ICFP'08*. New York, NY, USA: ACM Press.

Schrijvers, Tom, Guillemette, Louis-Julien, & Monnier, Stefan. (2008b). Type invariants for haskell. *Pages 39–48 of: Proceedings of PLPV'09*. PLPV '09. New York, NY, USA: ACM.

Schrijvers, Tom, Peyton Jones, Simon, Sulzmann, Martin, & Vytiniotis, Dimitrios. (2009). Complete and decidable type inference for GADTs. *Proceedings of ICFP'09*. ACM Press.

Simonet, V., & Pottier, F. (2007). A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, **29**(1).

Skalka, Christian, & Pottier, François. (2002). Syntactic type soundness for HM(X). *Electronic Notes in Theoretical Computer Science*, **75**.

Stuckey, P. J., & Sulzmann, M. (2005). A theory of overloading. *ACM Transactions on Programming Languages and Systems*, **27**(6), 1–54.

Sulzmann, M. 2000 (May). *A general framework for Hindley/Milner type systems with constraints*. Ph.D. thesis, Yale University, Department of Computer Science.

Sulzmann, M., Müller, M., & Zenger, C. (1999). *Hindley/Milner style type systems in constraint form*. Research Report ACRC-99-009. University of South Australia, School of Computer and Information Science.

Sulzmann, M., Schrijvers, T., & Stuckey, P. J. (2006a). Principal type inference for GHC-style multi-parameter type classes. *Pages 26–43 of: Proceedings of APLAS'06*. LNCS, vol. 4279. Springer-Verlag.

Sulzmann, M., Chakravarty, M., Peyton Jones, S., & Donnelly, K. (2007a). System F with type equality coercions. *Proceedings of TLDI'07*. ACM Press.

Sulzmann, M., Duck, Gregory J., Peyton Jones, S., & Stuckey, P. J. (2007b). Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, **17**(1), 83–129.

Sulzmann, M., Schrijvers, T., & Stuckey, P. 2008 (Jan.). *Type inference for GADTs via Herbrand constraint abduction*. Report CW 507. Department of Computer Science, K.U. Leuven, Leuven, Belgium.

Sulzmann, Martin, Wazny, Jeremy, & Stuckey, Peter. (2006b). A framework for extended algebraic data types. *Pages 47–64 of: Proceedings of FLOPS'06*. LNCS, vol. 3945. Springer-Verlag.

Tofte, M. (1990). Type inference for polymorphic references. *Information and computation*, **89**(1).

Vytiniotis, Dimitrios, Peyton Jones, Simon, & Schrijvers, Tom. (2010). Let should not be generalized. *Pages 39–50 of: Proceedings of TLDI'10*. New York, NY, USA: ACM.

Wadler, Philip, & Blott, Stephen. (1989). How to make ad-hoc polymorphism less ad hoc. *Proceedings of POPL'89*. ACM Press.

Weirich, Stephanie, Vytiniotis, Dimitrios, Peyton Jones, Simon, & Zdancewic, Steve. (2010). Generative type abstraction and type-level computation. *Proceedings of POPL'10*. ACM Press.

Wright, Andrew. (1995). Simple imperative polymorphism. *Lisp and Symbolic Computation*, **8**, 343–355.

Xi, Hongwei, Chen, Chiyan, & Chen, Gang. (2003). Guarded recursive datatype constructors. *Pages 224–235 of: Proceedings POPL'03*. ACM Press.