Type Checking with Open Type Functions

Tom Schrijvers [*]
K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

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

Manuel Chakravarty
UNSW, Australia
chak@cse.unsw.edu.au

Martin Sulzmann
ITU, Denmark
martin.sulzmann@gmail.com

Abstract

We report on an extension of Haskell with open type-level functions and equality constraints that unifies earlier work on GADTs, functional dependencies, and associated types. The contribution of the paper is that we identify and characterise the key technical challenge of *entailment checking*; and we give a novel, decidable, sound, and complete algorithm to solve it, together with some practically-important variants. Our system is implemented in GHC, and is already in active use.

Keywords Haskell, type checking, type functions, type families

1. Introduction

Dependently-typed languages, such as ATS (Chen and Xi 2005), Cayenne (Augustsson 1998), and Epigram (McBride), enable the programmer to encode complex properties in types. Recently, there has been a push to try to gain much of the expressiveness of dependently-typed languages without actually allowing values as types. Instead, type-level computations are realised through a limited form of function abstraction and pattern matching on types—examples of this approach are functional dependencies in Haskell (Jones 2000), and *generalised abstract data types* (GADTs) (Xi et al. 2003; Peyton Jones et al. 2006), as well as the experimental, Haskell-like Ωmega (Sheard 2006) and Chameleon (Sulzmann et al. 2006) systems.

However, these type systems rapidly become complex; for example, no paper or implementation known to us gives a satisfactory account of the interaction between functional dependencies and GADTs. Our goal in this paper is to unify as much as possible of this research in a backwards-compatible, practical extension to Haskell. Earlier, more foundational work by (Johann and Ghani 2008) and ourselves (Sulzmann et al. 2007a) suggested that a core type system embodying *equality constraints* and *existential types* forms a natural basis for type-level reasoning. In this paper we briefly propose source-language constructs that expose these two features, but our main focus is on the type checking challenges that are thrown up by equality constraints. Our particular contributions are these:

- We briefly introduce *open* type-level functions in Haskell, and show their usefulness (Section 2). Haskell type functions generalise our previous work on *associated types* (Chakravarty et al. 2005), and are independent of type classes (whereas associated types and functional dependencies were tied to classes).
- We identify and characterise the *entailment problem* that arises in type checking, and explain why it is new and tricky (Section 3). A distinguishing feature of our work is that, because our compiler has a typed intermediate language, the type checker is required to produce *evidence* (i.e. a proof term) that can be embodied in the typed intermediate-language form of the program (Section 3.3).
- We give, in detail, an algorithm for entailment, consisting of two parts: completion (Section 4) and solving (Section 5). Although there is much related work (Section 9), the algorithm appears to be novel. We have proved that the algorithm is sound and complete (Section 6).
- An often-neglected point is that the solver must co-operate with the rest of type inference. In particular, the constraints may mention *unification variables*, and the constraint solver may fix values for some of these variables. We tackle this issue in Section 7.
- It turns out that getting crisp termination and completeness properties requires quite draconian restrictions on the type functions — albeit more liberal than those used for functional dependencies. In Section 8 we present and characterise a variant with much greater expressive power, but where, in rare cases, the type checker may reject a program that is, in principle, typeable.

The system has been implemented in the latest version of the Glasgow Haskell Compiler (GHC), and is fully integrated with other aspects of type checking, notably type classes and GADTs. The work is still in progress — the implementation lags the paper slightly — but it is already in active use by others. Related work, of which there is plenty, is discussed in Section 9.

2. Motivation and examples

Type functions, especially in conjunction with GADTs, prove to be useful to implement many of the examples typically put forward in favour of functional dependencies (Jones 2000), Ωmega (Sheard 2006), and even dependently-typed languages. This section dis-

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

cusses open type functions in Haskell and provides running examples for the rest of the paper.

Parametrised collections. Jones (2000) motivating example for functional dependencies are parametrised collections. Using type functions—called *type families* in Haskell—we can define a similar type class of collections:

```
type family Elem c      -- family indexed by c
class Collects c where
  empty  :: c
  insert :: Elem c -> c -> c
  toList :: c -> [Elem c]

type instance Elem BitSet = Char
instance Collects BitSet where ...

type instance Elem [e] = e
instance Eq (Elem c) => Collects [c] c where ...
```

The type function application `Elem c` determines the element type of a collection of type `c`. The `type family` declaration introduces a type function and fixes its arity. Each `type instance` declares an equation of the type function. Just like type classes, new `type instances` can be added for new types at any time; that is, type functions are *open*. If type function and type class declarations and instances go hand in hand, as in this example, it is often convenient to combine the two. This makes a type function into an *associated type* (Chakravarty et al. 2005). Associated types are supported by our implementation, but as far as the type theory is concerned, they are merely syntactic sugar, and we will not consider them any further in this paper.

It is common that the types of value-level functions involving type functions need to constrain the range of a type function application. For example, the following value-level function based on the collection class may only be applied to collections whose element type is `Char`:

```
insx :: (Coll c, Elem c ~ Char) => c -> c
insx c = insert c 'x'
```

The signature of `insx` uses an *equality constraint* `Elem c ~ Char` to restrict the element type. Equality constraints, whose general form is `t1 ~ t2`, can appear anywhere that Haskell admits class constraints. For an equality constraint to be satisfied, the two types must unify modulo the non-syntactic equalities introduced by type family instances — i.e., equations of type functions. As we shall discuss in more detail in Section 3, checking an inferred type that includes equality constraints against a user supplied signature is the central problem in type checking programs with type functions.

Bounded vectors. A standard example demonstrating the utility of dependent types are *bounded vectors;* i.e., inductively defined lists whose type is indexed by the length of the list. By using two empty data types `Z` and `S a` to represent Peano numerals, we can define bounded vectors using a *generalised abstract data type (GADT)* (Xi et al. 2003; Peyton Jones et al. 2006).

```
data Z     -- Peano numerals
data S a   -- at the level of types

data Vec e len where       -- bounded vector
  Nil  :: Vec e Z
  Cons :: e -> Vec e len -> Vec e (S len)
```

The GADT definition ensures that a type `Vec e len` is only inhabited by vectors of length `len`; for example, we have `Cons 'a' (Cons 'b' Nil) :: Vec Char (S (S Z))`.

Instead of dependent types, `Vec` uses a *type-indexed data type*. Consequently, operations on `Vec` changing the length need to be accompanied by appropriate type-level length computations. For example, to define the concatenation of two bounded vectors, we need addition on type-level Peano numerals. We implement this by way of a type function for addition:

```
type family Add n m
type instance Add Z x     = x
type instance Add (S x) y = S (Add x y)

vappend :: Vec e n -> Vec e m -> Vec e (Add n m)
vappend Nil         l   = l
vappend (Cons x xs) ys  = Cons x (vappend xs ys)
```

The result type of `vappend`, `Vec e (Add n m)`, is a GADT indexed by a type function application. A similar combination of GADTs and type functions may be used to implement a wide range of uses of dependent types. During type checking, they lead to the same challenges as equality constraints. This becomes obvious when considering the equality GADT:

```
data EQ a b where
  EQ :: EQ a a
```

A type `EQ t1 t2` requires the same constraint solving capabilities as an equality constraint `t1 ~ t2`.

Type-preserving CPS transformation. The idea of expressing relationships between parameters of GADTs is a very versatile one. As another example of this idea, consider a type-preserving CPS transformation, where we want to statically ensure that the transformation only produces well-typed programs. We start with a GADT of expressions parametrised by their type:

```
data Exp t where
  Pair :: Exp a -> Exp b       -> Exp (a, b)
  Lam :: (Exp s -> Exp t)      -> Exp (s -> t)
  App :: Exp (s -> t) -> Exp s -> Exp t
```

Now, a CPS transformation that gets an expressions, of object type `t`, and the current continuation as an argument has the type

```
cps :: Exp t -> (ValK (Cps t) -> ExpK) -> ExpK
```

The crucial point is that the type of the current continuation `ValK (Cps t) -> ExpK` has an argument type parametrised by the CPS-transformed object type, namely `Cps t`, where `Cps` is the following type function:

```
type family Cps t
type instance Cps (a, b)   = (Cps a, Cps b)
type instance Cps (s -> t) = (Cps s, Cps t -> Z) -> Z
```

This example is from (Guillemette and Monnier 2008) who implemented a type-preserving CPS transformation for a more substantial expression type in two variants: (1) using type functions and (2) using only GADTs and encoding `Cps` using additional GADTs together with explicit type conversion functions. The version using only GADTs has 397 LoC, whereas with type functions it shrinks by a third to 264 LoC.

Monad Transformers. The Monad Transformer Library (MTL)[1] defines a family of elementary monads as well as combinators—the transformers—to compose elementary monads into more complex monads (Liang et al. 1995). This library has proven to be very useful and popular, with many other Haskell packages building on it.

The elementary monads are usually parametrised and the type of these extra parameters is dependent on the concrete monad type. For example, the reader monad defined by the class `MonadReader` has an environment parameter `r` depending on the monad. The MTL, which predates the proposal of type families for Haskell, expresses this situation using functional dependencies; thus,

[1] `http://darcs.haskell.org/packages/mtl/`

```
class (Monad m) => MonadReader r m | m -> r where
  ask   :: m r
  local :: (r -> r) -> m a -> m a
```

A similar situation arises in the `MonadError` class, which provides a common interface for monads `m` that support throwing and catching errors of type `e`:

```
class (Monad m) => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

We can simplify these two-parameter classes into one-parameter classes if we use type functions to project the dependent type (i.e., the environment of a reader monad and the error type of an error monad) from the monad type `m`:

```
type family Env (m :: * -> *)
type family Err (m :: * -> *)

class (Monad m) => MonadReader m where
  ask   :: m (Env m)
  local :: (Env m -> Env m) -> m a -> m a
class (Monad m) => MonadError m where
  throwError :: Err m -> m a
  catchError :: m a -> (Err m -> m a) -> m a
```

As with the example of parametrised collections, we would usually use the syntactic sugar provided by associated types here.

The MTL provides a number of basic instances of these monads; e.g., `Reader r` for `MonadReader`:

```
type instance Env (Reader r) = r

instance MonadReader (Reader r)
```

The type function instances for the elementary monad classes are rather straightforward. However, as soon as we turn to the *transformers*, which add, e.g., reader and error capabilities to other monads, matters get more complicated. For instance, the monad `ErrorT e (Reader r)` adds error capabilities to the `Reader` monad.

```
type instance Err (ErrorT e m) = e

instance MonadError (ErrorT e m)
```

And yet, we still need to be able to use the reader functionality of `ErrorT e (Reader r)`. Generally, we can say that every error transformation of a reader monad is also a reader monad. This naturally leads to a recursive definition of `Env`; thus,

```
type instance Env (ErrorT e m) = Env m

instance MonadReader m => MonadReader (ErrorT e m)
```

3. Formulating the problem

We begin with an overview of how to adapt the type checking engine to accommodate type functions. Reconsider `insx` and `insert` from Section 2:

```
insert :: (Coll c) => c -> Elem c -> c

insx :: (Coll c, Elem c ~ Char) => c -> c
insx c = insert c 'x'
```

A standard approach is to reduce type checking to a constraint satisfaction problem. Instantiating the call to `insert` in the body of `insx` gives rise to the class constraint (`Coll` c). The second argument of `insert` has type `Elem c`, which must be equal to the supplied argument, of type `Char`; hence, we have the constraint `Elem` $c \sim$ `Char`. (We use the symbol $\sim$ for type equality to avoid confusion with the mathematical equality symbol $=$.) These constraints can both be trivially discharged, since they both appear in `insx`'s type signature.

Now consider the GADT pattern match in `vappend` of Section 2, where we, for brevity, only consider the first clause of `vappend`:

```
vappend :: Vec e n -> Vec e m -> Vec e (Add n m)
vappend Nil l = l
```

We know that `l` is of type `Vec` e m, and the result of `vappend` should have type `Vec` e (`Add` n m). These two should be the same; so the program text gives rise to the constraint `Vec` $e\,m \sim$ `Vec` e (`Add` $n\,m$) which, by decomposition, holds iff $m \sim$ `Add` $n\ m$ holds. In addition, the pattern match `Nil` makes available the (local) assumption $n \sim$ `Z`, which we can use to reason that our desired constraint is equivalent to $m \sim$ `Add Z` m. Now, we can use the `type instance` for `Add Z` m to show that this constraint does indeed hold.

3.1 The entailment problem

Generalising from these examples, we see that type checking proceeds by generating from the program text (e.g. the body of a function) a set of *wanted equations*, E_w. The task of this paper is to find an algorithm that attempts to satisfy the wanted equations from

- E_t, the *top-level equations*, written directly by the programmer in the form of `type instance` declarations. These constitute the type function theory.
- E_g, the *local given equations*, which arise from:
  (a) programmer-supplied type annotations (e.g. the signature for `insx`).
  (b) the extra equations arising from a GADT pattern match (e.g. $n \sim$ `Z` in the `Nil` case of `vappend`).

We may write the deduction like this:

$$E_t \cup E_g \vdash E_w$$

and call it the *entailment problem*. We will completely ignore type-class constraints in this paper, since they are well studied elsewhere; our focus is on equational constraints. Initially, we will also ignore the fact that the wanted equations E_w will typically involve Damas-Milner-style *unification variables*. The pure checking problem is hard enough, so we defer unification variables to Section 7.

3.2 Syntax

Figure 1 defines the syntax of types. We use the meta-variables s, t, u, v to range over monomorphic types:

$$s, t, u, v \quad ::= \quad a \mid F\ t_1...t_n \mid S\ t_1...t_n$$

We only solve constraints involving monomorphic types, but the overall setting is a polymorphic type system, so a type can mention a quantified type variable a. For the purposes of constraint solving these can be treated simply as constants.

Type constructors come in two forms. *Data type* (DT) constructors, ranged over by S, T, are familiar from ML and Haskell, and include types such as `List`, `Bool`, `Maybe`, and tuples. For example, `Maybe` is declared thus:

```
data Maybe a = Nothing | Just a
```

Data type constructors also include a fixed set of primitive types, such as `Int` and `Float`.

The other kind of type constructor is a *type function* (TF) constructor, ranged over by F, G. They are introduced by a type family declaration giving its kind, thus:

```
type family F a
```

Type function constructors	F, G
Data type (DT) constructors	S, T
Type variables	a, b
Evidence variables	g
Skolem types	α, β
Schema Variables	x, y

Types	s, t, u, v	$::=$	$a \mid F\,t_1 ... t_n \mid S\,t_1 ... t_n$
Function-free types	c, d	$::=$	$a \mid S\,c_1 ... c_n$
Evidence	γ	$::=$	$g\,\bar{t} \mid F\,\bar{\gamma} \mid S\,\bar{\gamma} \mid t$
		$\mid$	$\mathtt{sym}\,\gamma \mid \gamma \circ \gamma \mid \mathtt{decomp}_{S_i}\,\gamma$
Top level equations	e^t	$::=$	$g\,\overline{x} : F\,\overline{c} \sim t$
Local equations	e^g	$::=$	$\gamma : s \sim t$
Wanted equations	e^w	$::=$	$g : s \sim t$
Equation sets	E	$::=$	$E_t \cup E_g$
	E_g	$::=$	$e^g_1 \ldots e^g_n$
	E_t	$::=$	$e^t_1 \ldots e^t_n$
Substitution	θ	$::=$	$[t_1/x_1, ..., t_n/x_n]$

Shorthands:		
	$\bar{t} \equiv t_1 ... t_n$	Sequence
	$\overline{t/x} \equiv [t_1/x_1, ..., t_n/x_n]$	Substitution

Figure 1. Syntax

Hole	$\square$		
Contexts	$\mathbb{S}, \mathbb{T}$	$::=$	$\square \mid a \mid F\,\mathbb{T}_1 ... \mathbb{T}_n \mid S\,\mathbb{T}_1 ... \mathbb{T}_n$
DT contexts	$\mathbb{C}$	$::=$	$S\,\mathbb{C}'_1 ... \mathbb{C}'_n$
DT' contexts	$\mathbb{C}'$	$::=$	$\square \mid t \mid S\,\mathbb{C}'_1 ... \mathbb{C}'_n$
Function contexts	$\mathbb{F}$	$::=$	$F\,\mathbb{T}_1 ... \mathbb{T}_n$

Figure 2. Type Contexts

Haskell supports higher-kinded types, but that is a distraction here so we assume that type constructors always appear saturated. For example, (`Maybe Int`) is a type, but `Maybe` is not.

Our other notational conventions are these:

- We use c, d to range over function-free types; that is, ones that mention no type-function constructors.
- We write $t \in s$ to denote that t is a sub-term of s, and $t \subset s$ to denote that t is a proper sub-term of s, i.e. $t \in s \wedge t \not\equiv s$.
- We write $\mathbb{T}$ to denote a *type context*, i.e. a type with holes ($\square$) in it. Figure 2 defines various kinds of type contexts, which we will introduce later as we need them. Holes are filled in as follows:
    - We write $\mathbb{T}[s]$ to denote $\mathbb{T}[s/\square]$, where $\mathbb{T}$ has a single hole. For instance, if $\mathbb{T} = \mathtt{Maybe}\ \square$, then $\mathbb{T}[\mathtt{Int}] = \mathtt{Maybe\ Int}$.
    - We write $\mathbb{T}\{s\}$ to denote $\mathbb{T}[s/\square]$, where $\mathbb{T}$ has any number of holes. For instance, if $\mathbb{T} = \mathtt{Either}\ \square\ \square$, then $\mathbb{T}[\mathtt{Int}] = \mathtt{Either\ Int\ Int}$.

3.3 Evidence

Our system is designed to fit into GHC, a compiler for Haskell that has a typed intermediate language. Type inference elaborates the program into an explicitly-typed calculus whose terms completely express their typing derivation (Sulzmann et al. 2007a). In particular, when discharging an equality constraint in E_w we must produce an *evidence term* encoding the proof tree, rather than simply declaring "yes the constraints are deducible". This evidence takes the form of a term, ranged over by γ, with the syntax (Figure 1):

$$\begin{array}{rcl} \gamma & ::= & g\,\bar{t} \mid F\,\bar{\gamma} \mid S\,\bar{\gamma} \mid t \\ & \mid & \mathtt{sym}\,\gamma \mid \gamma \circ \gamma \mid \mathtt{decomp}_{S_i}\,\gamma \end{array}$$

$$\text{(VarT)}\quad \frac{g\,\overline{x} : s_1 \sim s_2 \in E}{E \vdash g\,\bar{t} : [\overline{t/x}]s_1 \sim [\overline{t/x}]s_2}$$

$$\text{(VarL)}\quad \frac{\gamma : s_1 \sim s_2 \in E}{E \vdash \gamma : s_1 \sim s_2}$$

$$\text{(Refl)}\quad E \vdash t : t \sim t \qquad \text{(Sym)}\quad \frac{E \vdash \gamma : s \sim t}{E \vdash \mathtt{sym}\,\gamma : t \sim s}$$

$$\text{(Trans)}\quad \frac{E \vdash \gamma_1 : t_1 \sim t_2 \quad E \vdash \gamma_2 : t_2 \sim t_3}{E \vdash \gamma_1 \circ \gamma_2 : t_1 \sim t_3}$$

$$\text{(CompF)}\quad \frac{E \vdash \gamma_i : s_i \sim t_i \quad i = 1, ..., n}{E \vdash F\,\gamma_1 ... \gamma_2\ : F\,s_1 ... s_n \sim F\,t_1 ... t_n}$$

$$\text{(CompT)}\quad \frac{E \vdash \gamma_i : s_i \sim t_i \quad i = 1, ..., n}{E \vdash T\,\gamma_1 ... \gamma_2 : T\,s_1 ... s_n \sim T\,t_1 ... t_n}$$

$$\text{(DecompT)}\quad \frac{E \vdash \gamma : T\,s_1 ... s_n \sim T\,t_1 ... t_n}{E \vdash \mathtt{decomp}_{T_i}\,\gamma : s_i \sim t_i} \quad (i \in 1..n)$$

Figure 3. Type Equation Proof System

We write $\gamma : s \sim t$ to mean "γ is evidence that $s \sim t$ holds". Just as a term in System F encodes its own typing derivation, an evidence term γ encodes the proof tree for a type equality. Concretely, Figure 3 gives the rules of deduction that may be used in the proof $E_t \cup E_g \vdash \gamma : s \sim t$, which can also be regarded as a type system for well-typed evidence terms (Sulzmann et al. 2007a). The entailment problem sketched in Section 3.1 can now be expressed more precisely.

DEFINITION 1 (Entailment Problem). *Given a set of top-level equations E_t, local equations E_g, and a set of wanted equality constraints $g_1 : s_1 \sim t_1, \ldots, g_n : s_n \sim t_n$, find evidence γ_i for each $s_i \sim t_i$, such that*

$$E_t \cup E_g \vdash \gamma_i : s_i \sim t_i$$

or report failure if no such γ_i exist for $i = 1, \ldots, n$.

That is, we want to find evidence (a proof) for each equality $s_i \sim t_i$, using the assumptions in the top-level and local equations.

Figure 3 gives the valid entailment judgements. Rule (VarT) and (VarL) extract top-level and local evidence respectively from the set of assumptions E. Referring to Figure 1, we use an explicit notation for top-level equations:

$$e^t \quad ::= \quad g\,\overline{x} : F\,\overline{c} \sim t$$

where g is the name of the evidence constant that witnesses the equation. For example, the `type instance` declarations for `Add` in Section 2 are rendered into the syntax of Figure 1 like this:

$$\begin{array}{l} g_1\ x : \mathtt{Add}\ x\ \mathtt{Z} \sim x \\ g_2\ x\ y : \mathtt{Add}\ x\ (\mathtt{S}\ y) \sim \mathtt{S}\ (\mathtt{Add}\ x\ y) \end{array}$$

Notice that the left hand side of each equation must take the form of a type function F applied to *function-free* types $\overline{c}$. This is analogous to the value-level requirement that the patterns in a Haskell function definition may mention data constructors only. Furthermore, we require[2] $\mathit{ftv}(\overline{c}) = \overline{x} \supseteq \mathit{ftv}(s)$; that is, the left hand side must bind all the parameters $\overline{x}$, and only they may be used in the right hand side. Unlike the value level, however, the same variable may appear more than once in $\overline{c}$; for example $g\,x : F\,x\,x \sim \mathtt{Int}$ is a valid top-level equation.

[2] $\mathit{ftv}(t)$ is the free type variables of t.

If θ is the substitution $\overline{[t/x]}$, then the evidence term $(g\ \overline{t})$ is a proof that $F\ \theta\overline{c} \sim \theta s$. See Rule (VarT). So, for example, g_1 Z is a proof of Add Z Z $\sim$ Z.

The top-level equations E_t are parametric over the variables $\overline{x}$, and are required to form a confluent, terminating rewrite system. In contrast, the local equations E_g are not parametric, and express the equality of two *arbitrary* types:

$$e^g ::= \gamma : s \sim t$$

Rules (CompF) and (CompT) express the idea that "if $s \sim t$, then certainly $F\ s \sim F\ t$", and similarly for data type constructors T. Rule (DecompT) expresses the inverse: "if $T\ s \sim T\ t$ then $s \sim t$". But this claim only holds for data type constructors! For example, if Maybe $s \sim$ Maybe t holds then surely $s \sim t$ holds too — but if Add $s_1\ s_2 \sim$ Add $t_1\ t_2$ then we do *not* know that $s_1 \sim t_1$. (In more familiar notation, $x_1 + x_2 = y_1 + y_2$ does not imply $x_1 = y_1$.) In short, data types are *injective*, but type families are not. That is why there is only one decomposition rule in Figure 3.

A type t can also be treated as an evidence term: it provides the trivial evidence that $t \sim t$, using rule (Refl). Finally, Rules (Sym) and (Trans) express the symmetry and transitivity of equality.

3.4 Solving entailment is tricky

We borrow ideas from the term rewriting community to solve the entailment problem. At first it seems quite easy. The top-level equations constitute an equational theory, so the natural approach is to view them as a rewrite system (from left to right). It is reasonable to expect the programmer to write `type instance` declarations that are confluent and terminating. Now, let us suppose that E_g is empty. Then it is easy to deduce whether a wanted equation $s \sim t$ is implied by E_t. Simply normalise s and t, using E_t as a rewrite system, and compare the normal forms: $s \sim t$ holds iff the normal forms of s and t are identical.

The entailment problem is also fairly easy when E_t is empty: simply compute the *congruence closure* — see (Bachmair and Tiwari 2000) and the references therein — of E_g, and see if $s \sim t$ is included in it. An alternative formulation is to find the *completion* of E_g. This completion, E'_g, is now a terminating, confluent rewrite system which we can use to normalise s and t as before, and check for identical normal forms. The two formulations are equivalent (Kapur 1997), but we prefer the latter because it uses the same rewriting infrastructure (both intellectual and implementation) as E_t.

We assume here that completion should not change E_t for two reasons. First, completion of non-ground equations (such as $E_t \cup E_g$) is undecidable (Novikov 1955). ((Beckert 1994) gives a completion algorithm for non-ground equations, but it may diverge.) Second, there may be a great many top-level equations in scope, only a few of which will be relevant to solving a particular entailment problem, so applying completion to them seems excessive.

So we formulate the following sub-problem: transform the local equations E_g into a form E'_g, in such a way that $E_t \cup E'_g$ forms a confluent and terminating rewrite system. This completion problem is new, in two ways:

- Completion of *ground* equations is decidable in polynomial time (David A. Plaisted and Andrea Sattler-Klein 1996), which seems promising because the local equations E_g are indeed ground. However, completing ground equations E_g *in the presence of fixed, top-level, non-ground equations* E_t appears to be a new problem, not covered in the literature. The extension is definitely non-trivial.
- The transformation must be *evidence-preserving*; that is, each equation in E'_g must come with evidence for its veracity, expressed in terms of the evidence provided by E_t and E_g.

Example 1. This examples illustrates the first point. Consider these equations:

$$\begin{array}{lcl} E_t & = & \{g : F\ Bool \sim F\ (G\ Int)\} \\ E_g & = & \{\gamma : G\ Int \sim Bool\} \end{array}$$

Considered separately, both sets are separately strongly normalising. Nevertheless, the combination of both rules is non-terminating. That is, strong normalisation is not compositional. □

4. Step 1: Completion

The object of completion is to transform E_g such that it forms a strongly normalising rewrite system with E_t. We treat completion as a rewrite system thus:

$$E_t\ \vdash\ E_g, \Sigma \Longrightarrow E'_g, \Sigma'$$

Here, Σ is a substitution mapping skolem constants, α, to types. Skolems are explained in Section 4.5, and may safely be ignored until then. Solving starts with Σ empty, and proceeds step by step no further rewrite rule applies. At this point $E_t \cup E'_g$ is a strongly normalising rewrite system, and each equation in E'_g is of form e'_g:

$$e'_g ::= \gamma : F\ \overline{t} \sim s\ \mid\ \gamma : a \sim t\ \mid\ \gamma : \alpha \sim t$$

where the left-hand side does not occur in the right-hand side.

We define the following set of rewrite combinators:

- $r_1 \circ r_2$: first perform rewrite r_1 and then r_2
- $r_1|r_2$: apply rewrite rule r_1 or r_2 (non-deterministic choice)
- **fix**(r) : exhaustively apply r

The overall completion algorithm is defined thus:

fix(**Triv**|**Swap**|**Skolem**|**Decomp**|**Top**|**Fail**|**Subst**)

The individual rewrite steps are discussed below. Each rewrite rule transforms the (E_g, Σ) pair, although to avoid clutter when writing rules we omit the parts that do not change.

4.1 Orienting equations

The **Swap** rules orient equations so that they are more useful as left-to-right rewrite rules. In general, we want either a type function $F\ \overline{s}$ or type variable a on the left-hand-side. If that isn't the case, but there is one on the right-hand-side, then we swap the sides of the equation. If we cannot have either, then a skolem α is preferred (Section 4.5). The case where both sides have a data type constructor is dealt with by the **Decomp** rule (Section 4.2).

Swap ::= **FunSwap** | **VarSwap** | **AlphaSwap**

$$\begin{array}{lrcl} (\textbf{FunSwap}) & \gamma : t \sim F\ \overline{s} & \Longrightarrow & \mathtt{sym}\ \gamma : F\ \overline{s} \sim t \\ & \multicolumn{3}{l}{\text{where either } t \text{ matches } T\ \overline{t} \text{ or } \alpha, \text{ or } t \in \overline{s}} \\ (\textbf{VarSwap}) & \gamma : t \sim a & \Longrightarrow & \mathtt{sym}\ \gamma : a \sim t \\ & \multicolumn{3}{l}{\text{where } t \text{ matches } T\ \overline{t} \text{ or } \alpha} \\ (\textbf{AlphaSwap}) & \gamma : T\ \overline{t} \sim \alpha & \Longrightarrow & \mathtt{sym}\ \gamma : \alpha \sim T\ \overline{t} \end{array}$$

In general, if there is a function application on both sides, such as $F\ a \sim G\ b$, the orientation does not matter and no swap rule applies. But **FunSwap** *does* apply in one case even when both sides are a function call: if the call on the left is a proper subterm of the call on the right, we swap. For example

$$F\ a \sim G\ (F\ a)\ \Longrightarrow\ G\ (F\ a) \sim F\ a$$

Re-orienting such rules eliminates an obvious source of divergence, when the equations are treated as left-to-right rewrites. That in turn lead to fewer uses of the loop-cutting **Skolem** rule (Section 4.5).

We also allow ourselves to discard trivial equations:

$$(\textbf{Triv})\quad \gamma : s \sim s\quad \Longrightarrow\quad \langle \textit{nothing} \rangle$$

4.2 Eliminating Injective Type Constructors

The **Decomp** rule decomposes equalities on DT constructors.

(Decomp)

$$\gamma : T\ t_1\ \cdots\ t_n \sim T\ s_1\ \cdots\ s_n \Longrightarrow \begin{array}{c} \texttt{decomp}_{T_1}\ \gamma : t_1 \sim s_1 \\ \cdots \\ \texttt{decomp}_{T_n}\ \gamma : t_n \sim s_n \end{array}$$

where T is an injective type constructor.

4.3 Applying Top-Level Equations

We apply the top-level equations as rewrite rules to both the left- and right-hand sides of the local equations, adjusting evidence as we do so. Suppose $g\ \bar{x} : F\ \overline{c} \sim s \in E_t$; then these rules apply:

$$\textbf{Top} ::= \textbf{TopL} \mid \textbf{TopR}$$

$$\begin{array}{llll} (\textbf{TopL}) & \gamma : \mathbb{T}[F\ \overline{c}[\overline{t}/\overline{x}]] \sim v & \Rightarrow & \texttt{sym}\ \mathbb{T}[g\ \overline{t}] \circ \gamma : \mathbb{T}[s[\overline{t}/\overline{x}]] \sim v \\ (\textbf{TopR}) & \gamma : u \sim \mathbb{T}[F\ \overline{c}[\overline{t}/\overline{x}]] & \Rightarrow & \gamma \circ \mathbb{T}[g\ \overline{t}] : u \sim \mathbb{T}[s[\overline{t}/\overline{x}]] \end{array}$$

4.4 Substituting Local Equations

Similarly, we can apply one local equation in all the others, again adjusting evidence as we do so. Suppose the current set equations is $\{\gamma_g : t \sim s\} \uplus E$, where $\uplus$ is disjoint union. Then the following rules apply:

$$\textbf{Subst} ::= \textbf{fix}(\textbf{Top}) \circ \textbf{fix}(\textbf{SubstL} \mid \textbf{SubstR})$$

$$\begin{array}{llll} (\textbf{SubstL}) & \gamma : \mathbb{T}[t] \sim v & \Longrightarrow & \texttt{sym}\ \mathbb{T}[\gamma_g] \circ \gamma : \mathbb{T}[s] \sim v \\ (\textbf{SubstR}) & \gamma : u \sim \mathbb{T}[t] & \Longrightarrow & \gamma \circ \mathbb{T}[\gamma_g] : u \sim \mathbb{T}[s] \\ & \multicolumn{3}{l}{\text{where } t \not\in s \text{ and } t \text{ not of form } T\ \overline{t}} \end{array}$$

Why the two fixpoints? The **fix**(**Top**) makes sure that all the top-level equations have been applied exhaustively first.

Example 2. Suppose $g : F\ [Int] \sim F\ Int \in E_t$, and E_g is as follows:

$$\gamma_2 : F\ Int \sim F\ [Int],\ \gamma_3 : F\ [Int] \sim Bool$$

If we substitute the top-level equation in the second of the local equations, the latter becomes

$$\texttt{sym}\ g \circ \gamma_3 : F\ Int \sim Bool$$

If we subsequently substitute the second equation in this new equation, we get the original equation, which leads to non-termination. However, if we instead apply the top-level equation exhaustively, we get:

$$\gamma_2 \circ g : F\ Int \sim F\ Int,\ \texttt{sym}\ g \circ \gamma_3 : F\ Int \sim Bool$$

Now the second equation is trivial, and can't be used for substitution. □

The second fixpoint ensures that each substitution is performed performed exhaustively on every other equation to which it is applicable.

Example 3. Consider

$$E_g = \gamma_1 : a \sim b,\ \gamma_2 : b \sim c,\ \gamma_3 : c \sim b$$

If we substitute for b in the first but *not* the third equation, we get

$$\gamma_1 \circ \gamma_2 : a \sim c,\ \gamma_2 : b \sim c,\ \gamma_3 : c \sim b$$

Now substitute for c in the first but *not* the second equation to get

$$\gamma_1 \circ \gamma_2 \circ \gamma_3 : a \sim b,\ \gamma_2 : b \sim c,\ \gamma_3 : c \sim b$$

This process can be repeated indefinitely. However, if we substitute b in all equations, we get

$$\gamma_1 \circ \gamma_2 : a \sim c,\ \gamma_2 : b \sim c,\ \gamma_3 \circ \gamma_2 : c \sim c$$

which precludes repeated substitution of b. □

4.5 The Skolem rule

The hard part about completion is dealing with divergence. Consider this program fragment:

```
type instance F [Int] = Int

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

At first this looks odd: how can $a \sim \texttt{[F }a\texttt{]}$ hold? If we had used a data type T instead of a type function `F`, only infinite types[3] would satisfy the constraint, so `f` could never be called. But with type functions the constraint makes perfect sense; we can call `f` at type `[Int]`, say, by supplying a proof that `[Int]` $\sim$ `[F [Int]]`, which certainly holds.

But now consider type checking the definition of `f` itself. Suppose we also have the top-level assumption

```
type instance H [x] = Int
```

and when type-checking `f`'s `<rhs>` we find that we must solve the constraint `H` $a \sim$ `Int`. It's easy! Just use the local assumption $a \sim \texttt{[F }a\texttt{]}$ left-to-right, to expose the fact that a is really a list; then H's top-level rule lets us simplify the left hand side to `Int`, and we are done.

But *we must not use* $a \sim \texttt{[F }a\texttt{]}$ *repeatedly as a left-to-right rewrite rule*, because doing so would clearly diverge — that is why (**Subst**) has the side condition that $t \not\in s$. Yet we must use it once, else we cannot find the proof we desire. Resolving this dilemma is precisely the cleverness of congruence closure. The latter is invariably explained with diagrams and pointers, but we prefer a symbolic presentation because it is easier to formalise and fits our rewriting framework. Following (Kapur 1997), we invent a fresh *skolem constant*, α, to stand for $(F\ a)$. Now replace the equation $\gamma : a \sim \texttt{[F }a\texttt{]}$ by the two equations

$$\begin{array}{rcl} \gamma & : & a \sim [\alpha] \\ F\ \gamma & : & \alpha \sim F\ [\alpha] \end{array}$$

(The second equation will then be flipped around by **FunSwap**, but that is a separate matter.) Once this is done we can freely substitute for a, thereby potentially exposing information about a at its usage sites.

In general, the rule is this:

$$\begin{array}{llll} (\textbf{Skolem}) & \gamma : t \sim \mathbb{C}[\mathbb{F}[t]] & \Longrightarrow & \begin{array}{l} \gamma : t \sim \mathbb{C}[\alpha] \\ \mathbb{F}[\gamma] : \alpha \sim \mathbb{F}[\mathbb{C}[\alpha]] \\ \alpha := \mathbb{F}[t] \end{array} \\ & \text{where} & \multicolumn{2}{l}{\alpha \text{ is a fresh skolem}} \\ & & \multicolumn{2}{l}{t \text{ is not of form } T\ \bar{s}} \end{array}$$

The notation $\alpha := \mathbb{F}[t]$ denotes the extension of the skolem substitution Σ with $[\mathbb{F}[t]/\alpha]$. The substitution Σ is simply accumulated until the entire entailment algorithm is is finished (i.e. both completion and solving); then it can be applied to the solution to eliminate the skolems from the result.

If the side condition that t is not of form $T\ \bar{a}$ does not hold then either **Decomp** or **DecompFail** will apply, so **Skolem** should not.

4.6 Inconsistency Detection

The local equations can be inconsistent. For example, suppose the user writes this function:

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

This function is perfectly well typed: if `f` is supplied with evidence that `Bool` and `Char` are the same time, then it can use that evidence to demonstrate the well-typedness of the body. Of

[3] E.g. the recursive type $\mu a.[a]$ is a solution.

course, we know that no such evidence can be produced — the equation `Int` $\sim$ `Char` is *inconsistent* — and hence `f` can never be called. Rejecting such definitions is *desirable* for error reporting, but *not essential* for soundness. Similar situations arise when pattern-matching a GADT, where some branches of the pattern match may be unreachable because their equality constraints are inconsistent.

Inconsistent constraints are not always detectable. For example:

```
f :: (Add a (S Z) ~ a) => a -> a
```

Since we know that $a \neq a + 1$, the constraint is inconsistent. In general we can encode arbitrary theorems of arithmetic, so it is clear that we cannot *guarantee* to detect inconsistency.

Nevertheless, where inconsistency is obvious, it is desirable to report it early, and we provide two rules to do so:

$$\textbf{Fail} ::= \textbf{DecompFail} \mid \textbf{OccursCheck}$$

The first check concerns non-matching DT constructors.

(DecompFail) Signal inconsistency:

$$\gamma : T_1\, t_1\, \cdots\, t_n \sim T_2\, s_1\, \cdots\, s_m \quad \Longrightarrow \quad \textbf{raise}\ \textit{error}$$

where $T_1 \neq T_2$ are injective type constructors.

The second check is known as the *occurs-check* in the unification-based Hindley-Milner type inference algorithm. If the left hand side of an equation appears in its own right-hand side, under a data-type constructor, the equation is unsatisfiable, so we fail:

$$(\textbf{OccursCheck}) \quad \gamma : s \sim \mathbb{C}[s] \quad \Longrightarrow \quad \textbf{raise}\ \textit{error}$$

Notice that the occurs check does not fire if the occurrence is under a type function. For example, $a \sim F\ [a]$ is not necessarily unsatisfiable, because it is possible that there is a top-level equation $g\,x : F\ [x] \sim Int$.

5. Step 2: Solving

In the solving phase, we attempt to discharge the wanted equations E_w with the help of the top-level equations E_t and the completed local equations E'_g. In the absence of unification variables (which we defer to Section 7) solving is straightforward. We take each wanted constraint separately, normalise it by applying the rewrite system $E_t \cup E'_g$, and check for syntactic equality.

The only complication is maintaining evidence. Solving must *construct* evidence for each equation in E_w using evidence *provided* by E_t and E'_g. It is convenient to do this by treating E_w as a set of constraints of form

$$\begin{array}{rcl} e^w & ::= & g : s \sim t \\ E_w & ::= & e^w_1, \ldots, e^w_n \end{array}$$

In a wanted constraint, the evidence g is an evidence *variable*. Now we can treat solving as a rewrite system thus:

$$E_t \cup E'_g \ \vdash \ E_w, \Theta \Longrightarrow E'_w, \Theta'$$

Here, Θ is a substitution mapping coercion variables to coercion terms. Solving starts with Θ empty, and proceeds step by step until E'_w is empty; then Θ' gives the proof of E_w.

The solver is just the fixpoint of the rules:

$$\begin{array}{rcl} \textbf{Solve} & = & \textbf{fix}(\textbf{TrivS} \mid \textbf{TopS} \mid \textbf{LocalS}) \\ \textbf{TopS} & = & \textbf{TopSL} \mid \textbf{TopSR} \\ \textbf{LocalS} & = & \textbf{LocalSL} \mid \textbf{LocalSR} \end{array}$$

The simplest rewrite rule removes a solved constraint:

$$(\textbf{TrivS}) \quad g : s \sim s \quad \Longrightarrow \quad g := s$$

Here, the notation $g := s$ means that the ever-growing substitution Θ is extended by a binding for g. In this rule we merely need the identity evidence, which is represented by the type s itself.

Next, we can apply a top-level equation anywhere on either side of a wanted constraint. The rules are similar to **TopL** and **TopR** except for the inverted evidence construction. Suppose $g\ \bar{x} : F\ \overline{c} \sim s \in E_t$; then

$$\begin{array}{lccl} (\textbf{TopSL}) & g : \mathbb{T}[F\ \overline{c}[\overline{t}/\overline{x}]] \sim v & \Longrightarrow & \begin{array}{l} g' : \mathbb{T}[s[\overline{t}/\overline{x}]] \sim v \\ g := \texttt{sym}\ \mathbb{T}[g\ \overline{t}] \circ \gamma \end{array} \\ (\textbf{TopSR}) & g : u \sim \mathbb{T}[F\ \overline{c}[\overline{t}/\overline{x}]] & \Longrightarrow & \begin{array}{l} g' : u \sim \mathbb{T}[s[\overline{t}/\overline{x}]] \\ g := \gamma \circ \mathbb{T}[g\ \overline{t}] \end{array} \end{array}$$

Similarly, we can apply the local equations on both sides. Suppose $\gamma : t \sim s \in E'_g$; then

$$\begin{array}{lccl} (\textbf{LocalSL}) & g : \mathbb{T}[t] \sim v & \Longrightarrow & \begin{array}{l} g' : \mathbb{T}[s] \sim v \\ g := \mathbb{T}[\gamma] \circ g' \end{array} \\ (\textbf{LocalSR}) & g : u \sim \mathbb{T}[t] & \Longrightarrow & \begin{array}{l} g' : u \sim \mathbb{T}[t]t \\ g := g' \circ \texttt{sym}\ \mathbb{T}[\gamma] \end{array} \end{array}$$

In effect, the full evidence for the original wanted equations is constructed gradually during the solving process. At the end, when no wanted equations remain, the evidence has been constructed in its entirety.

6. Properties

In this section, we consider various vital properties of our type checking algorithm related to soundness, completeness and termination. First we consider the programmer-supplied top-level equations (Section 6.1), then we show that the completion algorithm is sound, decidable, and complete (Section 6.2), and finally that the solving algorithm also enjoys these properties (Section 6.3).

The interested reader may find proofs of the main theorems in an Appendix at

`http://www.cs.kuleuven.be/~toms/icfp2008/`

6.1 Strong Normalisation of the Top-Level Equations

As we have mentioned, the minimum requirement on the top-level equations is that they be strongly normalising – that is, confluent and terminating. However, we have tried, and failed, to find a completion algorithm that is sound, decidable, and complete, assuming *only* that the top-level equations are strongly normalising. Instead, we require that they satisfy somewhat more restrictive conditions.

Our first stab at these conditions is inspired by the *Terminating Weak Coverage Condition* of (Sulzmann et al. 2007b):

DEFINITION 2 (Strong Termination Condition). *The top-level equations E_t satisfy the Strong Termination Condition iff*

1. *The equations have non-overlapping left-hand sides.*
2. *For each equation $g\,\overline{x} : F\ \overline{c} \sim t \in E_t$, either t contains no type functions, or t is of form $G\ \overline{d}$ (where the $\overline{d}$ are function free), and*
   (a) *the sum of the number of DT constructors and schema variables in the right-hand side is smaller than the similar sum in the left-hand side, and*
   (b) *the right-hand side has no more occurrences of any schema variable than the left-hand side.*

These conditions are rather restrictive — for example, they exclude most of the examples in Section 2 — and we will consider more relaxed conditions in Section 8. However, they certainly ensure that the top-level equations are strongly normalising and, as we shall see in the following sub-sections, are enough to guarantee good properties for completion and solving.

THEOREM 1 (Strong Normalisation 1). *Any set of top-level equations E_t that satisfies the Strong Termination Condition is strongly normalising.*

Proof. (Sketch) We show termination by defining a term norm $|t|$ with a well-founded ordering, and showing that this norm decreases with each rewrite. Since there are no critical pairs (non-joinable or otherwise) because of the non-overlap condition and the rewrite rules terminate, we have confluence. □

6.2 Completion Properties

Assuming the strong termination condition, the completion process produces an environment that has the same proof strength as the original one. We prove this in two steps: soundness and completeness.

Firstly, any equation provable in the completed environment can also be proven in the original environment, with exactly the same evidence.

LEMMA 1 (Soundness). *Let E'_g with skolem substitution Σ' be the completion of E_g with respect to E_t. Then, for any γ, s, t that don't contain skolems*

$$E_t \cup E'_g \vdash \gamma : s \sim t \quad \textit{implies} \quad E_t \cup E_g \vdash \Sigma'(\gamma) : s \sim t$$

Proof. The proof is straightforward: we can easily show that the property is preserved by each rewrite step in the completion algorithm. □

Secondly, any equation provable in the original environment can also be proven in the completed environment. Note that we do not demand that the same evidence is given. In general, this is not possible. Take for instance the **Triv** rewrite step, which forgets certain given evidence. However, the particular choice of evidence is not relevant. Any evidence will do, as long as it is sound.

LEMMA 2 (Completeness). *Let E'_g be the completion of E_g with respect to E_t. Then, for any γ, s, t,*

$$E_t \cup E_g \vdash \gamma : s \sim t \quad \textit{implies} \quad \exists \gamma' : E_t \cup \vdash \gamma' : s \sim t$$

Proof. The proof is similar as for the soundness property. □

Hence, we may conclude that the original and complete environment have the same proof strength.

THEOREM 2 (Equivalence). *Let E'_g with skolem substitution Σ' be the completion of E_g with respect to E_t. Then, for any s, t that don't contain skolems,*

$$\exists \gamma : E_t \cup E_g \vdash \gamma : s \sim t \quad \textit{iff} \quad \exists \gamma' : E_t \cup E'_g \vdash \gamma' : s \sim t$$

Proof. The theorem follows from Lemma 1 and Lemma 2. □

The purpose of the completion algorithm is to yield an environment that is strongly normalising. In other words, when read from left-to-right as rewrite rules by the **TopS** and **LocalS** steps, the completed environment is *confluent* and *terminating*.

THEOREM 3 (Strong Normalisation 2). *Let E'_g be the completion of E_g with respect to E_t. Then we have that $E_t \cup E'_g$ is a strongly normalising rewrite system, when the equations are read as left-to-right rewrite rules.*

Proof. (summary) The proof strategy for termination is similar to that of Theorem 1. We show confluence by establishing the joinability of all critical pairs. □

Finally, we want the completion process to terminate.

THEOREM 4 (Termination). *Completion terminates.*

Proof. (summary) The proof consists of defining a suitable norm, with a well-founded order, that decreases with each step. □

6.3 Solving Properties

Once we have obtained the completed environment, we use it to solve the wanted equations. Solving should correspond to proving in the proof system of Figure 3.

THEOREM 5 (Soundness). *Assume that $E_t \cup E'_g$ is a completed environment. Then, for any E_w, Θ,*

$$\textit{if } E_t \cup E'_g \vdash E_w, \emptyset \Longrightarrow \emptyset, \Theta \quad \textit{then} \quad E_t \cup E'_g \vdash \Theta(E_w)$$

Proof. The proof is straightforward: we can easily show that the property is preserved by each rewrite step in the solving algorithm. □

The solving process should terminate and be complete.

THEOREM 6 (Termination). *Solving terminates.*

Proof. Termination trivially follows from the fact that $E_t \cup E'_g$ is strongly normalising (Theorem 3). □

Moreover, the solving algorithm always finds a proof, if there is one.

THEOREM 7 (Completeness). *Let E'_g be the completion of E_g with respect to E_t. Then for all E_w,*

$$\textit{if } E_t \cup E_g \vdash E_w \quad \textit{then} \quad \exists \Theta : E_t \cup E'_g \vdash E_w, \emptyset \Longrightarrow \emptyset, \Theta$$

Proof. Completeness also follows from the fact that $E_t \cup E'_g$ is strongly normalising. It means that all equivalent terms have the same normal form. So $t \sim s$ holds iff the normal forms of t and s are identical. □

7. First extension: unification variables

Thus far we have assumed that the wanted constraints are fully known, so that the entailment check is the only problem. In reality, though, a type inference system must take account of *unification variables*. For example, consider this program fragment

```
insx :: forall c. (Coll c, Elem c ~ Char) => c -> c
isBig :: BitSet -> Bool

f c = ...(insx c)...(isBig c)...
```

As discussed in Section 3, we may regard type inference as a two-step process: first generate constraints from the program text, and then solve them (Simonet and Pottier 2007; Sulzmann et al. 2006). In the case of `f`, the lambda-bound variable `c` will be given a unification variable δ_c as its type. The call to `insx` will be instantiated with a unification variable δ_a, and give rise to the constraints $(\texttt{Coll}\,\delta_a, \texttt{Coll}\,\delta_a \sim \texttt{Char})$. Since `c` is passed both to `insx` and to `isBig` we also generate constraints $(\delta_a \sim \delta_c$ and $\delta_c \sim \texttt{BitSet}$ respectively. The context of the call to `insx` may cause δ to be unified with, say `BitSet`. Only at this point can we discharge the constraint $(\texttt{Elem}\,\delta \sim \texttt{Char})$.

Unification variables arise only in the wanted constraints E_w. The top level equations E_t are written directly by the programmer. The local equations E_g that arise from a user type signature are also fully known. Lastly, those that arise from GADT matching are also free of unification variables, or else type inference would be intractable — see (Sulzmann et al. 2008; Peyton Jones et al. 2006).

7.1 Solving with unification variables

Unification variables complicate the solving algorithm described in Section 5. In general, we can no longer consider the wanted equations one at a time, but must consider their interaction. We have already seen one example: suppose $E_w = (\delta \sim \texttt{BitSet}, \texttt{Elem}\,\delta \sim \texttt{Char})$.

In terms of rewrites, we must now also carry around a substitution ϕ of unification variables and Σ of skolems (see **SkolemS**):

$$E_t \cup E'_g \vdash E_w, \Theta, \phi, \Sigma \Longrightarrow E'_w, \Theta', \phi', \Sigma'$$

Usually the initial substitutions are empty. If the equations hold, then at then end E'_w and Σ' are empty.

7.2 The new rules

The extension adds five new rules, four of which are simple variants of previously discussed rules. The new one is the unification rule, which fires when we learn what type a unification variable stands for:

$$\begin{array}{lccl}(\textbf{Unify}) & E_w, \Theta, \phi, \Sigma & \Longrightarrow & \Sigma(E_w)[t/\delta], \Sigma(\Theta), \phi \cup \{\delta := t\}, \emptyset \\ & \multicolumn{3}{l}{\text{where } g : \delta \sim t \in E_w \text{ or } g : t \sim \delta \in E_w \text{, and } \delta \notin \Sigma(t)}\end{array}$$

Note that the unification step eagerly applies the substitution $[t/\delta]$ to the entire set of wanted constraints, so that δ is completely eliminated. Each application of **Unify** will produce a wanted constraint $g : t \sim t$, but that is eliminated by **TrivS**. **Unify** also eliminates any skolems introduced by **SkolemS**, by applying Σ and continuing with the empty skolem substitution; the reason for this is explained when we discuss **SkolemS**.

Next, like any unifier, we must decompose data type applications to expose possible uses of **Unify**, remembering to maintain evidence:

$$(\textbf{DecompS}) \quad g : T\ t_1\ \cdots\ t_n \sim T\ s_1\ \cdots\ s_n \Rightarrow \begin{array}{c} g_1 : t_1 \sim s_1 \\ \cdots \\ g_n : t_n \sim s_n \\ g := T\ g_1 \cdots g_n \end{array}$$

The **SubstS** step propagates information from one equation to another to uncover further hidden unifications. For example, the unification $\delta \sim \texttt{Int}$ is uncovered by substituting the wanted equation $F\ \delta \sim \texttt{[}\delta\texttt{]}$ into the other wanted equation $F\ \delta \sim \texttt{[Int]}$.

$$\textbf{SubstS} \quad ::= \textbf{SubstSL} \mid \textbf{SubstSR}$$

Suppose the current set equations is $E \uplus \{g_1 : \mathbb{F}[\delta] \sim s\}$, where $\uplus$ is disjoint union, $\mathbb{F}[\delta] \notin s$. These rules are applied exhaustively to E.

$$(\textbf{SubstSL}) \quad g_2 : \mathbb{T}[\mathbb{F}[\delta]] \sim v \quad \Longrightarrow \quad \begin{array}{l} g_3 : \mathbb{T}[s] \sim v \\ g_2 := \texttt{sym}\ \mathbb{T}[g_1] \circ g_3 \end{array}$$

$$(\textbf{SubstSR}) \quad g_2 : u \sim \mathbb{T}[\mathbb{F}[\delta]] \quad \Longrightarrow \quad \begin{array}{l} g_3 : u \sim \mathbb{T}[s] \\ g_2 := g_3 \circ \texttt{sym}\ \mathbb{T}[g_1] \end{array}$$

where g_3 is a fresh evidence variable. Note that this rule is similar to **Subst** (Section 4.4), but requires that the substitution involves a unification variable δ. The intuition behind this restriction is that solving is stuck because of lack of information. All information must be supplied by $E_t \cup E'_g$ except information on unification variables. Hence, substitution of wanted equations in other wanted equations is only justifiable for unification variables. Moreover, the substitution of types without unification variables easily leads to nonterminating interaction with the given local equations.

Example 4. Assume we have $E'_g = \{F\ [Int] \sim F\ (G\ Int)\}$ and $E_w = \{G\ Int \sim [Int], H\ (F\ [Int]) \sim Bool\}$. If we substitute the local given equation in the second wanted equation we get $H\ (F\ (G\ Int)) \sim Bool$. Now we can substitute the first wanted equation. This results in $H\ (F\ [Int]) \sim Bool$, which is the original wanted equation. Now the process can repeat itself. □

Such nontermination is avoided with **SubstS** because the local given equations do not contain unification variables. For exactly the same reason as in the **Subst** rules, we must be sure to apply the top-level equations exhaustively before using **SubstS**.

Because equations are not always oriented properly, we need the **FunSwapS** rule, which only differs from **FunSwap** in its treatment of evidence.

$$\begin{array}{lccl}(\textbf{FunSwapS}) & g_1 : t \sim F\ \overline{s} & \Longrightarrow & \begin{array}{l} g_2 : F\ \overline{s} \sim t \\ g_1 := \texttt{sym}\ g_2 \end{array} \\ & \multicolumn{3}{r}{\text{where either } t \text{ matches } T\ \overline{t} \text{ or } \alpha \text{, or } t \in F\ \overline{s}}\end{array}$$

Finally, we also need **SkolemS**, the counterpart of **Skolem**, to allow safe substitution of wanted equations where the left-hand side occurs in the right-hand side. Note that **SkolemS** is restricted to left-hand sides that also match the **SubstS** rule.

$$\begin{array}{lccl}(\textbf{SkolemS}) & g_1 : \mathbb{F}_1[\delta] \sim \mathbb{C}[\mathbb{F}_2[\mathbb{F}_1[\delta]]] & \Rightarrow & \begin{array}{l} g_1 : \mathbb{F}_1[\delta] \sim \mathbb{C}[\alpha] \\ g_2 : \alpha \sim \mathbb{F}_2[\mathbb{C}[\alpha]] \\ \alpha := \mathbb{F}_2[\mathbb{F}_1[\delta]] \end{array} \\ & \multicolumn{3}{r}{\text{where} \quad \alpha \ \text{ is a fresh skolem}}\end{array}$$

We need to take care if **Unify** binds a unification variable that is mentioned inside a skolem binding $\alpha := \mathbb{F}_2[\mathbb{F}_1[\delta]]$. When δ is bound, we might be able to simplify $\mathbb{F}_2[\mathbb{F}_1[\delta]]$, but the skolem will prevent that information from being "seen" by other constraints. To avoid this the **Unify** rule brutally undoes all skolemisation by substitution; the **SkolemS** will re-introduce any skolems that are necessary. (In practice, we could be less brutal by substituting only skolems whose bindings involved the unified variable.)

7.3 The new solving algorithm

The new solving algorithm, **SolveU**, is an extension of the previous one **Solve** (Section 5):

$$\begin{array}{rcl} \textbf{SolveU} & = & \textbf{fix}(\textbf{Solve} \circ \textbf{Extension}) \\ \textbf{Extension} & = & \textbf{DecompS}|\textbf{Unify}|\textbf{FunSwapS}|\textbf{SkolemS}|\textbf{SubstS} \end{array}$$

Of the ordering imposed by **SolveU**, only the fact that **fix**(**TopS**|**LocalS**) comes before **SkolemS** is essential. An implementation is free to otherwise order the rules as it pleases. The essential ordering is necessary for completeness' sake; it prevents **SkolemS** from hiding a redex.

The following example shows most of the rewrite steps in action.

Example 5. For simplicity we omit the evidence. Let $E_t = \{H\ [x] \sim [Int]\}$ and $E_g = \{F\ Int \sim [Int], G\ [Int] \sim Int\}$. The wanted equations are:

$$F\ \delta \sim [G\ (F\ \delta)],\ H\ (F\ \delta) \sim [\delta]$$

We can't substitute the first equation in the second with **SubstS** because $F\ \delta$ also appears in the right-hand side. We have to apply **SkolemS** first.

$$F\ \delta \sim [\alpha],\ \alpha \sim G\ (F\ [\alpha]),\ H\ (F\ \delta) \sim [\delta]$$

where $\alpha := G\ (F\ \delta)$. Now we can do the **SubstS** substitution.

$$F\ \delta \sim [\alpha],\ \alpha \sim G\ (F\ [\alpha]),\ H\ [\alpha] \sim [\delta]$$

This enables us to apply the toplevel equation with **TopS**.

$$F\ \delta \sim [\alpha],\ \alpha \sim G\ (F\ [\alpha]),\ [Int] \sim [\delta]$$

After **DecompS**, we apply **Unify** on $Int{\sim}\delta$. This unification makes us substitute α again with $G\ (F\ Int)$.

$$F\ Int \sim [G\ (F\ Int)],\ G\ (F\ Int) \sim G\ (F\ [G\ (F\ Int)])$$

The resulting wanted equations can be solved as before, with **LocalS** and **TrivS**. □

7.4 Stable Principal Solutions

If E'_w and Σ' are empty, then the resulting pair of substitutions (Θ', ϕ') should make the wanted equations hold. If that is the case, we call (Θ', ϕ') a *solution*.

DEFINITION 3 (Solution). *The pair* (Θ, ϕ) *is a* solution *of the wanted equations* E_w *with respect to local equations* $E_t \cup E_w$ *iff*[4]

- $dom(\phi) \subseteq uvars(E_w)$,
- *ϕ is an idempotent substitution, i.e. its domain does not appear in its range, and*
- *the equations hold under the substitution*

$$E_t \cup E_g\ \vdash\ \phi(\Theta(E_w))$$

[4] $uvars(t)$ is the unification variables in t.

However, we don't want just any solution. No, we want the most general or *principal* solution. Some wanted equations do not have a unique principal solution. These *ambiguous* equations should be rejected.

Example 6. For an example of wanted equations without a principal solution, consider $E_t \cup E_g = \{F\ Int \sim Char, F\ Bool \sim Char\}$ and $E_w = \{F\ \delta \sim Char\}$. There are two distinct substitutions that make E_w hold: $\phi_1 = [Int/\delta]$ and $\phi_2 = [Bool/\delta]$. □

Because our type functions are open, and we expect that new toplevel equations are added frequently, we want our principal solutions to be *stable*. A principal solution is stable when it remains principal when the toplevel equations are extend. When we have a stable principal solution, we don't have to run the type checker again and again with every new toplevel equation. Stability also means that the type checker can avoid speculative function evaluation, as in the narrowing evaluation strategy. Because of this we can still guarantee termination and completeness, as we'll see shortly in Section 7.5.

Not all principal solutions are stable.

Example 7. Consider $E_t \cup E_g = \{F\ Int \sim Char\}$ and $E_w = \{F\ \delta \sim Char\}$. The principal solution of E_w is $\phi_1 = [Int/\delta]$. However, if we extend $E_t \cup E_g$ with $F\ Bool \sim Char$, then we get a second solution $\phi_2 = [Bool/\delta]$, and ϕ_1 is no longer principal. Hence, it is not stable. □

DEFINITION 4 (Stable Principal Solution). *Solution* (Θ, ϕ) *is a* stable principal solution *of* E_w *with respect to* $E_t \cup E_g$ *iff it subsumes all solutions* ψ *of* E_w *with respect to* $E_t \cup E_g \cup E$*, with* E *an arbitrary extension of the local equations. Subsumption means that there is an auxiliary substitution* σ *such that* σ *composed with* ϕ *is equivalent to* ψ *modulo the equational theory*

$$\forall \delta \in uvars(E_w).\exists \gamma, \sigma : E_t \cup E_g \cup E \vdash \gamma : \sigma \circ \phi(\delta) \sim \psi(\delta)$$

7.5 Properties

As unification variables do not arise in top-level or local equations, we only have to revisit the properties of the solving phase.

It is straightforward to see that the soundness and termination properties of solving still hold. For soundness, the unambiguity and stability properties are enforced because all unifications are entailed by the wanted equations themselves. No search or other form of guessing takes place. For termination, the number of unifications is bounded by the number of unification variables in the initial set of wanted equations.

At the moment we conjecture that our algorithm is complete with respect to unification variables.

CONJECTURE 1 (Completeness wrt. Unification Variables). *The solving algorithm constructs a stable principal solution for any wanted equations* E_w*, if there is one.*

We leave the proof as future work.

7.6 Retrospective

Solving in Section 5 was pretty simple. Adding unification variables has made it much more complicated. Indeed, it will not have escaped the reader that the rules for solving in this section now look extremely similar to the rules for completion (Section 4), apart from the different treatment of evidence. This similarity is directly exploited by our implementation.

In future work we will go further, and explore the direct combination of completion and solving into a single algorithm with a single set of rules, rather than the current two-phase approach.

8. Second extension: relaxed top-level conditions

The Strong Termination Condition (Definition 2) greatly restricts the expressive power of type functions. For example, even the `Add` example is not covered. In this section we explore a much more relaxed condition, with more expressive power.

DEFINITION 5 (Relaxed Condition). *The set of equations* E_t *is* strongly well behaved *iff equations in* E_t *are of the form* $g\,\overline{x} : F\,\overline{c} \sim s$ *and*

1. *No two left-hand sides overlap.*
2. *For each subterm* $G\,\overline{t} \in s$
   (a) *there is no subterm* $H\,\overline{u} \in G\,\overline{t}$*,*
   (b) *the sum of the number of DT constructors and schema variables is smaller than the similar number in* $\overline{c}$*, and*
   (c) *there are not more occurrences of any schema variable* x *than in* $\overline{c}$*.*

All of the examples in Section 2 satisfy this definition. Here are some smaller examples to illustrate:

```
-- These ones are ok
type instance F x      = x
type instance F [Bool] = F Char
type instance F (a,b)  = (F a,F b)
type instance F x      = (x,x)

-- These ones are not ok
type instance F [Char] = F (F Char)  -- Nested fn
type instance F [x]    = F [x]       -- Too many DT
type instance F Bool   = F [Char]    --   constructors
```

THEOREM 8. *The Relaxed Condition ensures that the equations* E_t *are strongly normalising.*

Proof. Condition (1) ensures confluence. Conditions (2) ensure termination. Define the level-mapping of a ground function call as the sum of DT constructors and function applications in arguments of that term. Let the norm of a term be the multiset of level-mappings of all its function applications. Let the order of norms be the usual multiset order: the greatest multiset is the one with the greatest cardinality of the greatest element; if the cardinality is the same, we look at the second greatest element and so on. The Relaxed Condition ensures that the norm of the term decreases under rewriting. □

Unfortunately, greater expressiveness comes at a cost. Under the Relaxed Condition we must choose between two alternatives:

(a) **either** guaranteed completeness and termination of solving, but (potential) non-termination of completion,

(b) **or** guaranteed termination of completion and solving, but (potential) incompleteness of solving.

8.1 Completeness without Termination

To see (a), assume the Relaxed Condition, and the completion algorithm of Section 4. Then completion may diverge.

Example 8. Consider the completion of $E_g = \{a \sim [F\ a]\}$ with respect to $E_t = \{F\ [x] \sim [F\ x]\}$. We apply the **Skolem** step to get $E'_g = \{a \sim [\alpha_1], F\ [\alpha_1] \sim \alpha_1\}$. Now we can apply **Top** on the second equation to get $[F\ \alpha_1] \sim \alpha_1$. After **AlphaSwap** we end up with a variant of the initial equation, and the process can repeat itself indefinitely. Hence, completion does not terminate. □

However, if completion *does* terminate, all the properties of solving remain valid: soundness, completeness and termination.

8.2 Termination without Completeness

The non-termination of the completion algorithm is caused by equations of form $\alpha{\sim}\mathbb{C}[\mathbb{F}[\alpha]]$ in E'_g; we call them *loopy equations*. Termination can be recovered in a simple but brutal manner: (i) during completion, do not apply **Skolem** to a loopy equation; and (ii) at the end of completion, discard all loopy equations. This change leads to alternative (b). Both completion and solving are guaranteed to terminate, but since we have thrown away some of the provided evidence, solving may no longer be complete. For example, suppose

$$\begin{array}{lcl} E_t & = & g_1\, x : F\,[x] \sim Int,\ g_2\, x : F\ (T\ x) \sim [F\ x] \\ E_g & = & g_3 : a \sim T\ (F\ a) \end{array}$$

Then it is possible to show $E_t \cup E_g \vdash \gamma : F\ a \sim \texttt{[Int]}$ where

$$\gamma = F\ g_3 \circ g_2\ (F\ a) \circ [F\ (F\ g_3 \circ g_2\ (F\ a)) \circ g_1\ (F\ (F\ a))]$$

The completion algorithm on E_g yields first $\{a \sim T\ \beta, \beta \sim [F\ \beta]\}$, and then, by discarding the loopy equation, to $\{a \sim T\ \beta\}$. Now if we attempt to solve the wanted equation, we first substitute a for $T\ \beta$, obtaining $T\ \beta \sim T\ [Int]$. After decomposing, we get stuck at $\beta \sim [Int]$. In other words, we are unable to discharge a wanted equation, even though there is a valid proof.

It is bad for a compiler to mysteriously loop, so we prefer alternative (b), especially since it enjoys the following property:

THEOREM 9 (Partial Completeness). *If completion does not discard any loopy equations, solving is complete.*

Even in the presence of loopy equations can we identify certain equations that are definitely inconsistent: these are the equations subject to **Fail**. Hence, by including **Fail** in the solving algorithm, we further reduce the class of undecidable wanted equations.

Another possible refinement of the approach would be to apply **Skolem** up to a nesting depth of k (by equipping each skolem with a level number), before discarding a level-k loopy equation. This would provide a knob for programmers to turn to say "work harder to typecheck this program".

8.3 Summary

The Relaxed Condition lacks the crisp completeness result that language designers crave, but in practice the extra expressiveness is very significant, and we believe that the loss of completeness (arising in alternative (b)) is rarely felt. The loss of completeness manifests in the following way: if the compiler fails to prove an equality, *and* it has discarded a loopy equation, it may report that it cannot be certain whether or not the wanted equality is justified. We believe these cases are rare because a loopy equation must have started in a form at least as complicated as this:

$$G\, t \sim T\, (F\, (G\, t))$$

Programmers are unlikely to write such equations in a type signature!

Does the special treatment of loopy equations permit even more liberal conditions on top-level equations, such as requiring only strong normalisation? No, it does not. Consider Example 1 in Section 3.4: both completion and solving diverge on the term $F\ [Int]$, without loopy equations ever arising.

9. Related Work

Languages with type function support. Existing languages with type functions differ on various accounts from our type functions. They support closed type functions (e.g. ATS (Chen and Xi 2005), Cayenne (Augustsson 1998), Epigram (McBride), Omega (Sheard 2004)) whereas we consider open, extensible type functions. Dependently typed languages generally impose fewer restrictions on the type functions the user can write. Type checking for such languages is therefore undecidable in general (Augustsson 1998) or requires some form of user interaction (Chapman et al. 2005) to recover decidability. Automatic type checking for closed type functions typically relies on narrowing to solve equations This leads to a very different type checking approach (Gasbichler et al. 2002; Sheard 2006). We are not aware of any formal decidability and completeness results which match the results stated in this paper. The alternative is to give up on automatic type checking and demand that the programmer has to construct the proofs himself (e.g. LH (Licata and Harper 2005)).

More closely related to our work is the Chameleon system described in (Sulzmann et al. 2006). Chameleon makes use of the Constraint Handling Rules (CHR) (Frühwirth 1998) formalism for the specification of type class and type improvement relations. CHR is a committed-choice language consisting of constraint rewrite rules. Via CHR rewrite rules we can model open type functions. The CHR type inference strategies (Stuckey and Sulzmann 2005) is different from the type function setting in that it mixes completion and solving. We keep both steps separate mainly because evidence generation is then fairly straightforward. The issue of evidence generation has not been fully addressed for CHR yet.

Functional dependencies. Functional dependencies (Jones 2000) provide the programmer with a relational notation for specifying type-level computations. Functional dependencies and type functions have similar expressive power, and it is mainly a matter of taste whether to write type-level programs in functional or relational style. For example, Neubauer et al. (Neubauer et al. 2001) and Diatchki (Diatchki 2007) propose a functional notation for type classes with a functional dependencies which is essentially syntactical sugar for the conventional relational notation of type classes.

This correspondence enables a fruitful and mutually beneficial transfer of results. For example, the Strong Termination Condition (Definition 2) is inspired by the Terminating Weak Coverage Condition for functional dependencies (Sulzmann et al. 2007b). In the other direction, our work naturally integrates type functions and GADTs, and the generation of evidence is reasonably straightforward. Neither of these issues has been studied for functional dependencies, but our results provide some strong clues how to achieve similar results for functional dependencies. Similarly, the relaxed conditions in Section 8 improves on existing type checking results for functional dependencies, but again might be transferable to that setting. We refer to (Schrijvers et al. 2007; Schrijvers and Sulzmann 2008) for more details on the connection and the translation schemes between functional dependencies and type functions.

Completion and congruence closure. For entailment checking we critically rely on completion. There are close connections between completion and building the congruence closure. Indeed, our Skolem rule can be found in similar form in (Kapur 1997) and our rule application strategies are close to the ones proposed in (Bachmair and Tiwari 2000). However, we are much more parsimonious than Kapur in our use of skolemisation, aiming to use it only where it is truly necessary.

Solving is considered in (Tiwari et al. 2000) but the assumptions differ from ours. We only allow the unification of variables in wanted equations whereas the approach in (Tiwari et al. 2000) also unifies variables in local equations. None of the above works consider the treatment of (universally quantified) top-level equations. This topic is studied in (Beckert 1994). Again, we consider a more specialised setting where we only complete local equations and leave top-level equations unchanged. We also need to construct evidence to justify wanted equations. The closest work we are aware of (Nieuwenhuis and Oliveras 2005) employs a different notion of proof (evidence).

10. Conclusion & Future Work

We presented a type checking algorithm for open type functions and equational constraints. Our implementation is available in the GHC HEAD branch, and is documented at `http://haskell.org/haskellwiki/GHC/Type_families`. It provides opportunities for many extensions that we plan to study, such as overlapping toplevel equations, closed functions, and a unified type checking algorithm for equational and class constraints, to name just three.

References

Lennart Augustsson. Cayenne - a language with dependent types. In *Proc. of ICFP'98*, pages 239–250. ACM Press, 1998.

Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In *Proc. of CADE'00*, volume 1831 of *LNCS*, pages 64–78. Springer-Verlag, 2000.

Bernhard Beckert. A completion-based method for mixed universal and rigid e-unification. In *Proc. of CADE'94*, pages 678–692. Springer-Verlag, 1994.

Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proc. of ICFP '05*, pages 241–253, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-064-7. doi: http://doi.acm.org/10.1145/1086365.1086397.

James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: a standalone typechecker for ETT. In *The Sixth Symposium on Trends in Functional Programming*, pages 79–94. Intellect, 2005.

Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proc. of ICFP'05*, pages 66–77. ACM Press, 2005.

David A. Plaisted and Andrea Sattler-Klein. Proof Lengths for Equational Completion. *Information and Computation*, 125(2): 154–170, 1996.

Iavor S. Diatchki. *High-level abstractions for low-level programming*. PhD thesis, OGI School of Science & Engineering, May 2007.

Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.

Martin Gasbichler, Matthias Neubauer, Michael Sperber, and Peter Thiemann. Functional logic overloading. In *Proc. POPL'02*, pages 233–244. ACM Press, 2002.

Louis-Julien Guillemette and Stefan Monnier. One vote for type families in haskell! In *The Ninth Symposium on Trends in Functional Programming*, 2008. Forthcoming.

Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In *Proc ACM Conference on Principles of Programming Languages (POPL'08)*, pages 297–308. ACM, 2008.

Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

Deepak Kapur. Shostak's congruence closure as completion. In *Proc. of RTA '97*, pages 23–37, London, UK, 1997. Springer-Verlag.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proc. of POPL '95*, pages 333–343, New York, NY, USA, 1995. ACM.

Daniel R. Licata and Robert Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.

Conor McBride. Epigram: A dependently typed functional programming language. http://www.dur.ac.uk/CARG/epigram/.

Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A functional notation for functional dependencies. In *Proceedings of the 2001 Haskell Workshop*, 2001.

Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Proc. of RTA'05*, volume 3467 of *LNCS*, pages 453–468. Springer-Verlag, 2005.

Pyotr S. Novikov. On the algorithmic unsolvability of the word problem in group theory. In *the Steklov Institute of Mathematics 44*, pages 1–143, 1955. (Russian).

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proc. of ICFP'06*, pages 50–61. ACM Press, 2006.

Tom Schrijvers and Martin Sulzmann. Restoring confluence of functional dependencies via type families. In *The Ninth Symposium on Trends in Functional Programming*, 2008. Forthcoming.

Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel Chakravarty. Towards open type functions for Haskell. In O. Chitil, editor, *Proceedings of the 19th International Symposium on Implemantation and Application of Functional Languages*, pages 233–251, 2007.

Tim Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM Press, 2004.

Tim Sheard. Type-level computation using narrowing in Omega. In *Proceedings of the Programming Languages meets Program Verification (PLPV 2006)*, volume 174 of *Electronic Notes in Computer Science*, pages 105–128, 2006.

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

Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

Martin Sulzmann, Jeremy Wazny, and Peter J.Stuckey. A framework for extended algebraic data types. In *Proc. of FLOPS'06*, volume 3945 of *LNCS*, pages 47–64. Springer-Verlag, 2006.

Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, 2007a.

Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, 2007b.

Martin Sulzmann, Tom Schrijvers, and P. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Report CW 507, Department of Computer Science, K.U.Leuven, Leuven, Belgium, January 2008.

Ashish Tiwari, Leo Bachmair, and Harald Rueß. Rigid -unification revisited. In *Proc. of CADE'00*, volume 1831 of *LNCS*. Springer-Verlag, 2000.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proc. of POPL '03*, pages 224–235, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-628-5. doi: http://doi.acm.org/10.1145/604131.604150.

A. Proof of Theorem 1

Proof. We show termination by defining a term norm $|t|$ with a well-founded ordering, and showing that this norm decreases with each rewrite. Firstly, we define our norm $|t|$ to be the multiset S of tuples (l, s), one for each function sub-term $F\ \overline{s} \in t$, where

- l is the *level* of the function sub-term: the number of function terms $G\ \overline{t} \in t$ that contain the function sub-term, and
- s is the *size* of the function sub-term: the number of DT constructors and type variables in the function sub-term not contained in a proper function sub-term $F'\ \overline{s}'$ of the function sub-term.

The ordering over these multisets is defined as:

$$\begin{array}{l} S_1 < S_2 \triangleq \\ \max_{(l_1,s_1)\in S_1} l_1 > \max_{(l_2,s_2)\in S_2} l_2 \\ \vee \begin{array}{l} \exists l.l = \max_{(l_1,s_1)\in S_1} l_1 \wedge l = \max_{(l_2,s_2)\in S_2} l_2 \\ \quad \wedge \sum_{(l,s_1)\in S_1} s_1 > \sum_{(l,s_2)\in S_2} s_2 \end{array} \end{array}$$

Clearly, this order is well-founded. Now consider the right-hand side of each top-level equation:

$G\ \overline{s}$ One function sub-term at level l is replaced by another function sub-term at level l. However, its size is strict smaller than than the original function sub-term (condition 1). Moreover, some function sub-terms below it may have been eliminated, while none has been added (condition 2).

x **or** $T\ \overline{s}$ One function sub-term at level l is eliminated. All (if any) the function sub-terms below it decrease in level. The size of its parent, at level $l - 1$ is increased by some number.

In summary, each rewrite decreases the norm.

As there are no critical pairs (non-joinable or otherwise) because of the non-overlap condition and the rewrite rules terminate, we have confluence. □

B. Proof of Theorem 3

Proof. We consider the two properties separately.

Termination. The environment $E_t \cup E_g'$ consists of three types of equations:

1. The first type, of the form $a \sim t \in E_g'$ and $\alpha \sim t$, we call *variable substitutions*. For simplicity of presentation, we only treat the type variable case in the rest of the proof. The extensions to skolems is trivial.
2. The second type, of the form $F\ \overline{t} \sim s \in E_g'$, we call *given function applications*.
3. The remaining type, the equations in E_t, we call *top-level function applications*.

First, we show that the number of variable substitutions is bounded. Every variable substitution equation of the form $a \sim t$ substitutes a type variable a everywhere, and preserves only one occurrences, that in $a \sim t$. No other rewrite step is able to increase the number of occurrences after this. As no new type variable a are generated, the number of variable substitutions is hence bounded by the number of variable substitution equations.

As the number of variable substitutions is bounded, we only have to consider infinite sequences of function applications. Such an infinite sequence cannot exist of top-level function applications alone, for it is a precondition that the top-level equations are terminating. An infinite sequence can also not exist of given function applications alone. Because of completion, no given function application applies only to the result (right-hand side) of another given function application. If that were the case, then the **Subst** rule would apply and E_g' would not be completed. Hence, each given function application must consume part of the original type term at the start of the sequence. As the original type term is finite, there cannot be an infinite sequence of given function applications.

As a consequence, any infinite sequence must consist of alternating finite sub-sequences of given and top-level function applications. We show that this is not possible by (1) defining a norm, the *available functions norm* $|t|_f$ for type terms t, with a well-founded ordering, and (2) showing that this norm decreases with each function application.

We define $|t|_f$ as in the proof of Theorem 1, but now only count the function sub-terms $F\ \overline{t}$ that potentially match the left-hand side of a rewrite rule during the rewriting process. For the initial term t we define $|t|_f$ to be all function sub-terms. For a bounded type term, this initial norm is bounded. Now we show that this norm strictly decreases with each given function application and does not increase with each top-level application.

The given function application eliminates one or more sub-terms of the form $F\ \overline{t}$, and while it may add in its stead more sub-terms of that form, none matches the left-hand side of another function application. If it would, the **Top** and **Subst** rules of the completion process would apply, and that cannot be the case. Hence, given function applications strictly decrease the norm.

A top-level function application eliminates one or more sub-terms of the form $F\ \overline{t}$: one matches the entire left-hand side and the others match the schema variables $\overline{x}$. Because of the conditions on the top-level equations, those that match the schema variables are either not duplicated or at a lower level. The function sub-term may replaced by another one, but then of strictly smaller size.

Confluence. For a set of terminating rewrite rules to be non-confluent, there must be a *non-joinable critical pair*. A critical pair consists of two rewrite rules (equations) with overlapping head (left-hand side) and distinct body (right-hand side).

Let us consider the three possibilities for a critical pair $e_1, e_2 \in E_t \cup E_g'$.

$e_1, e_2 \in E_t$ There is no such critical pair because we require that the left-hand sides of top-level equations are non-overlapping.

$e_1, e_2 \in E_g'$ As the given equations are ground, the left-hand sides may only overlap if they are identical. However, if the are identical, then the **Subst** rule would apply, and $E_t \cup E_g'$ would not be a completed environment. Hence, there is no such critical pair.

$e_1 = F\ \overline{c} \sim s \in E_t, e_2 = u \sim v \in E_g'$ For the left-hand sides to overlap, there must be an instance $e_1[\overline{t/x}]$ of e_1 such that either $F\ \overline{c}[\overline{t/x}] \in u$ or $u \in F\ \overline{c}[\overline{t/x}]$. The former is not possible, because it would mean that the **Top** rule applies, and $E_t \cup E_g'$ is not a completed environment.

For the latter we must realise that u is of the form a or $G\ \overline{s}$, because of completion. If we take the restrictions on e_1's left-hand side into account, it must follow from $u \in F\ \overline{c}[\overline{t/x}]$ that $u \in \overline{t}$. Now we can easily show the joinability of the critical pair: the join is obtained by applying first one rule and then the other.

$$\begin{tikzcd}
& F\ \overline{c}[\overline{t/x}] \arrow[dl, "e_1"'] \arrow[dr, "e_2"] & \\
s[\overline{t/x}] \arrow[dr, "e_2"'] & & F\ \overline{c}[\overline{t'/x}] \arrow[dl, "e_1"] \\
& s[\overline{t'/x}] &
\end{tikzcd}$$

□

C. Proof of Theorem 4

Proof. Our proof establishes termination properties for a few individual steps, and then shows termination for the remaining steps of the algorithm.

We consider two individual steps.

- Firstly, we observe that the **Fail** step immediately terminates the type checking process.
- Secondly, the number of *variable substitutions* is bounded. With a variable substitution we denote a **Subst** step that substitutes an equation of the form $a \sim t$ in all other equations. Observe that the number of type variables in $E_g \cup E_w$ is bounded, and E_t does not contain type variables. Each variable substitution eliminates a type variable a from all equations but the left-hand side of $a \sim t$. No further step increases the number of occurrences of a, and no further step creates fresh type variables. In conclusion, there is only a bounded number of variable substitutions.

Hence, we should only concern ourselves with the remaining rewrite rules for non-termination. We do this by defining a *completion norm* $|E|_c$ for wanted equations with a well-founded order, and by showing that this norm decreases with each step of the algorithm.

First, we define the completion norm $|E|_c$ as the tuple $\langle n_1, n'_1, n_2, n_3, n_4\rangle$ with lexicographic ordering, where

- n_1 is the number of *available functions* defined as in the previous termination proof,
- n'_1 is a multiset defined below,
- n_2 is the multiset of sub-term sizes (one size for each sub-term in E) with the multiset order,
- n_3 is the number of equations in E, and
- n_4 is the number of *ill-oriented* equations in E, i.e. those equations to which **Swap** applies.

The multiset n'_1 is a multiset of tuples (l, u, d), one for each available function subterm $F\ \overline{s}$, where

- l is the *level* of the function subterm: the number of function terms $G\ \overline{t}$ that contain the function subterm,
- u is the *up-size* of the function subterm: the number of DT constructors between the function term and its surrounding parent function term or the term root,
- d is the *down-size* of the function subterm: the number of DT constructors and type variables in the function subterm not contained in a proper function subterm $F'\ \overline{s}'$ of the function subterm.

Define $\max_l(S) \triangleq \max_{(l,u,d)\in S} l$, $c_l(l,S) \triangleq \sum_{(l,u,d)\in S} 1$, $c_u(l,S) \triangleq \sum_{(l,u,d)\in S} u$ and $c_d(l,S) \triangleq \sum_{(l,u,d)\in S} d$. The ordering over these multisets is defined as:

$$
\begin{array}{l}
S_1 < S_2 \triangleq \\
\max_l(S_1) < \max_l(S_2) \\
\vee \ \bigwedge \left\{ \begin{array}{c} \max_l(S_1) = \max_l(S_2) \\ \exists l. \bigwedge \left\{ \begin{array}{c} c_l(l,S_1) < c_l(l,S_2) \\ \forall l' > l. c_l(l',S_1) = c_l(l',S_2) \end{array} \right\} \end{array} \right\} \\
\vee \ \bigwedge \left\{ \begin{array}{c} \max_l(S_1) = \max_l(S_2) \\ \forall l. c_l(l,S_1) = c_l(l,S_2) \\ \exists l. \bigwedge \left\{ \begin{array}{c} c_u(l,S_1) < c_u(l,S_2) \\ \forall l' < l. c_u(l',S_1) = c_u(l',S_2) \end{array} \right\} \end{array} \right\} \\
\vee \ \bigwedge \left\{ \begin{array}{c} \max_l(S_1) = \max_l(S_2) \\ \forall l. c_l(l,S_1) = c_l(l,S_2) \\ \forall l. c_u(l,S_1) = c_u(l,S_2) \\ \exists l. \bigwedge \left\{ \begin{array}{c} c_d(l,S_1) < c_d(l,S_2) \\ \forall l' > l. c_d(l',S_1) = c_d(l',S_2) \end{array} \right\} \end{array} \right\}
\end{array}
$$

Now let us consider the impact of the various steps on this norm.

- **Swap** decreases n_4 and does not affect the other components.
- **Triv** definitely decreases n_2 and n_3, and possibly decreases n_1, n'_1 and n_4.
- **Decomp** decreases n_2, but possibly increases n_3 and n_4. It does not affect n_1 and possibly decreases n'_1.
- **Skolem** decreases n'_1, doesn't increase n_1 and may affect the other components. To see the decrease in n'_1, note that the up-size of the $\mathbb{F}[t]$ is > 0 and it becomes $= 0$ in $\mathbb{F}[[C]]$ after the skolem step. There is also down-size that increases, but the up-size has priority.
- **Subst** decreases n_1 and n'_1, but may increase n_2 and n_4. To see that it decreases n_1 we can follow similar reasoning as in the earlier termination proof. It cannot be that a top-level equation applies to the right-hand side of a substituted equation because **fix**(**Top**) comes before **Subst**.
- **Top** decreases n'_1 because of the Strong Termination Condition. It possibly decreases n_1, but not necessarily.

In conclusion, the completion algorithm must terminate. □

D. Termination Proof of Scenario (b)'s Completion in Section 8

Proof. (sketch) The termination argument of completion hinges on the fact that only a bounded number of ground type terms are generated and considered. However, the **Skolem** step generated new constants, the skolems, which may lead to non-termination. In Appendix C we have been able to find a termination order for the Strong Termination Condition. Under the Relaxed Condition, this order does not guarantee termination anymore. Our solution is to modify the **Skolem** rule: we do not generate skolems for loopy equations. This provides an alternate way to bound the number of skolems generated. The remainder of the proof of Appendix C remains valid. Hence, only a finite number of ground terms are generated, of which only a bounded number are skolemised. □