

# Type Classes With More Higher-Order Polymorphism

Matthias Neubauer    Peter Thiemann  
Universität Freiburg  
{neubauer,thiemann}@informatik.uni-freiburg.de

## Abstract

We propose an extension of Haskell's type class system with lambda abstractions in the type language. Type inference for our extension relies on a novel constrained unification procedure called guided higher-order unification. This unification procedure is more general than Haskell's kind-preserving unification but less powerful than full higher-order unification.

The main technical result is the soundness and completeness of the unification rules for the fragment of lambda calculus that we admit on the type level.

## Categories and Subject Descriptors

D.3.2 [Programming Languages]: Language Classifications—*Applicative (functional) languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*Polymorphism*; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*Type Structure*

## General Terms

Languages, Theory

## Keywords

type classes, higher-order unification, type inference, Haskell

## 1 Introduction

Haskell's type system is among the most powerful type systems implemented in a programming language. This is mostly due to Haskell's type class system which supports parametric overloading on type constructors and thus higher-order polymorphism [12]. These features have found many applications in everyday programming problems. Abstract concepts like monads and functors can be

conveniently expressed using type classes of higher-order constructors. To retain a decidable type inference algorithm, Haskell's type language is restricted to a combinator language, *i.e.*, there are only type constructors and variables, but no abstractions. However, there are a number of interesting examples, where the lack of lambda abstraction in the type language is limiting and leads to awkward programming exercises.

Hence, we propose to extend Haskell's type language with a (suitably restricted) notion of lambda abstraction. The main problems here are to remain downwards compatible with Haskell98 and to keep type inference decidable. If we were to add unrestricted lambda abstraction, then type inference would have to rely on unification for simply-typed lambda terms (higher-order unification [11]). There are well-known decidable subclasses of this problem, for instance, pattern unification [20], generalizations thereof [35], and bounded higher-order unification [36]. These subclasses are used for implementing logic-based programming languages ( $\lambda$ Prolog [25], Elf [33] and its descendants) and theorem provers (most notably Isabelle [30]). Unfortunately, the subclasses mentioned above are not really suited for type inference as we will demonstrate in Section 1.5.

Technically, the paper reports the following results.

- We define a restricted simply-typed lambda calculus  $\Lambda_{\text{GHOU}}$  suitable as a type language for extended Haskell.
- We define a Henkin model for  $\Lambda_{\text{GHOU}}$  and prove that every function type in the model is inhabited only by injective functions. This is required for downwards compatibility.
- We define guided higher-order unification and prove the soundness and completeness of the unification rules. Unfortunately, we currently have to rely on a heuristic to guarantee termination of this procedure.

In the next subsections, we exhibit some programming problems that motivate the introduction of lambda abstraction into the type language. Then, we discuss the problems of using existing higher-order unification procedures (which are too non-deterministic), we recall Haskell's approach, and explain our approach informally. Section 2 starts the formal treatment. After introducing some notation, we define a syntax for our language (Section 2.2) and a semantics for  $\Lambda_{\text{GHOU}}$ , the type-level language (Section 2.3), based on sets of injective functions. After stating some properties of  $\Lambda_{\text{GHOU}}$ , we define guided higher-order unification in Section 2.4 and prove soundness and completeness of the rules. We briefly discuss termination issues (Section 2.5) and sketch an extension of Milner's algorithm  $W$  to our system in Section 2.6. Section 3 evaluates our proposal by reviewing the motivating examples and by discussing

some limitations. Finally, we discuss related work (Section 4) and conclude.

We have completed a prototype implementation and have performed some preliminary experiments with it. In particular, all examples presented in this paper have been checked. The implementation builds on Jones's work on Typing Haskell in Haskell [16]. The implementation of higher-order unification is inspired by Nipkow's paper [27].

Throughout, we assume familiarity with Haskell [9]. We always refer to Haskell98 as specified in the standard, not to the many extensions present in Haskell implementations.

## 1.1 Functors

The type class `Functor` expresses the essence of the categorical concept of a functor by specifying the action of the functor on a function of type `a -> b`. If `f` is a functor, then the map function for `f`, called `fmap`, transforms a function of type `a -> b` into a function of type `f a -> f b`. The following type class definition formalizes this.

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Many one-parameter type constructors are in fact functors, for example, the type constructor `[]` that maps a type `a` to the type `[a]` of lists with elements of type `a` or the type constructor `Maybe` that maps a type `a` to the type `Maybe a` with elements `Nothing` and `Just x`, for some `x` of type `a`. The following instance declarations illustrate the map functions for the `[]` and `Maybe` functors.

```
instance Functor [] where
  fmap h [] = []
  fmap h (x:xs) = h x : g xs
```

```
instance Functor Maybe where
  fmap h Nothing = Nothing
  fmap h (Just x) = Just (h x)
```

The concept of a functor has much in common with the usual notion of a map between sets, only at a higher level. One commonality is the fact that

- there is an identity functor that maps each type to itself and
- the composition of two functors is again a functor.

Unfortunately, these concepts are not readily available in Haskell. In both cases, additional data structures have to be introduced that provide names for the identity functor and for the composition operator. Here is a suitable definition for the identity functor.

```
data Id x = Id x

instance Functor Id where
  fmap h (Id x) = Id (h x)
```

The composition operator requires another special feature of Haskell. Type variables may not just range over types, but also over functions on types, functions on functions on types, and so on. Hence, the following definitions make sense.

```
data Comp f g x = Comp (f (g x))

instance (Functor f, Functor g) => Functor (Comp f g) where
  fmap h (Comp fgx) = Comp (fmap (fmap h) fgx)
```

While it is possible to write functions with composable functors and functor identify using these definitions, it seems rather awkward to

be forced to provide names for something as simple as the identity function or function composition.

Life would be much easier for the poor programmer if we could define similar instances of functor, but without having to introduce the data types `Id` and `Comp`<sup>1</sup>. If we were to make a wish, then we would rather write the following code:

```
instance Functor (λx. x) where
  fmap h x = h x

instance (Functor f, Functor g) =>
  Functor (λx. f (g x)) where
  fmap h fgx = fmap (fmap h) fgx
```

That is, instead of using a named identity function and a named composition operator, we employ anonymous functions, *i.e.*, lambda expressions in the type language.

## 1.2 Contravariant Functors

Once we have accepted the use of lambdas in the type language, we can readily identify further applications. For example, the function type constructor `->` is a functor in its second (result) parameter and this fact is easily expressed as a standard instance declaration (where `->`) is the prefix notation for the infix operator `->`):

```
instance Functor ((->) a) where
  fmap h g = h . g
```

However, it is well-known that `->` is a *contravariant functor* (where the action of `f` on a mapping `a -> b` exchanges domain and range) in its first argument. Of course, this concept can be captured in a type class, as well.

```
class OpFunctor g where
  opmap :: (a -> b) -> (f b -> f a)
```

Unfortunately, in Haskell it is not possible to make the function type constructor `->` an instance of `OpFunctor` because it is not possible to abstract over the first parameter of `->`. Given lambda expressions in the type language, this turns out to be easy.

```
instance OpFunctor (λx. x -> a) where
  opmap h g = g . h
```

## 1.3 A Real-World Example: WASH/CGI

Before the reader is tempted to rate this motivation as an academic exercise, here is a real-world example, taken from our work on WASH/CGI [37, 39], an embedded domain-specific language for server-side Web scripting. In this context, we need to perform a task which is known as “kind lifting” in the generic programming community. Roughly spoken, the essence of lifting is to pull type parameters out from contexts.

Among other features, WASH/CGI provides a means to construct typed interactions as follows. The programmer constructs an HTML form by using the combinator `inputField` to construct typed input fields. Another combinator, `submit`, specifies the action taken when the form is submitted (via the submit button). Here is an example:

```
demo = run
  (ask (standardPage "Demo"
    (makeForm $
      do text "Enter a number "
```

<sup>1</sup>Besides being awkward they also impose a performance penalty.

```
f1 <- inputField (fieldSIZE 10)
submit f1 showIt (fieldVALUE "Echo"))))
```

```
showIt :: InputField Int VALID -> CGI ()
showIt f1 =
  let n1 = value f1 in
  htell $ standardPage "Input Echo" (text (show n1))
```

The important point here is that the evaluation of `inputField (fieldSIZE 10)` creates a *handle*, `f1`, to an input field of type `InputField Int INVALID`. The `submit` combinator, used at type

```
submit :: InputField Int INVALID ->
  (InputField Int VALID -> CGI ()) ->
  HTMLField ()
```

validates the input field (it checks that the input field is present and contains a number) and passes it to the function, `showIt`, that processes the form once it is submitted.

Since there are different kinds of input fields (pushbuttons, selections boxes, etc.), the combinator `submit` is overloaded as follows:

```
submit :: InputHandle h =>
  h INVALID -> (h VALID -> CGI ()) -> HTMLField ()
```

where the class `InputHandle` has instances for the input fields listed above. For example

```
instance InputHandle (InputField a) where ...
```

In the class constraint `InputHandle h`, the variable `h` ranges over type constructors of kind `* -> *`. This is also evident in the instance declaration where `InputField` is only applied to one type argument (`a`) instead of two, as it is declared.

Now consider the case where we want to pass a pair of handles to the processing function:

```
submit (f1, f2) actionOnSubmit ...
```

Unfortunately, this results in a strange type error message with Haskell98. As a workaround, the programmer<sup>2</sup> must invent a “suitable” lifted datatype and must make it an instance of the `InputHandle` class:

```
data F2 a b x = F2 (a x) (b x)
instance (InputHandle a, InputHandle b) =>
  InputHandle (F2 a b) where ...
```

The reason for having to invent `F2` is as follows. To apply `submit` to `(f1, f2)` it is necessary to unify the argument type of `submit` with the type of `(f1, f2)`:

```
submit :: InputHandle h =>
  h INVALID -> (h VALID -> CGI ()) -> HTMLField ()
(f1, f2) :: (InputField Int INVALID, InputField Char INVALID)
```

Hence, the type checker must solve the equation

```
h INVALID
=?= (InputField Int INVALID, InputField Char INVALID)
```

This equation cannot be solved by first-order unification (which is the basis of Haskell’s type inference). In this situation there are (at least) two remedies. Either, we perform kind lifting explicitly by introducing the `F2` type shown above. Or we might consider using higher-order unification in the type checker. In fact, higher-order unification can proceed by substituting  $h \mapsto \lambda x. (h1\ x, h2\ x)$ , with `h1` and `h2` fresh type variables, which is called an *imitation step*

<sup>2</sup>... or rather the library designer.

in higher-order unification. Applying this substitution and simplifying  $(\lambda x. (h1\ x, h2\ x))\ INVALID$  leads to

```
(h1 INVALID, h2 INVALID)
=?= (InputField Int INVALID, InputField Char INVALID)
```

This can be decomposed to

```
h1 INVALID =?= InputField Int INVALID
h2 INVALID =?= InputField Char INVALID
```

Again by imitation, we substitute  $h1 \mapsto \lambda x. InputField (h11\ x)\ (h12\ x)$ , again with `h11` and `h12` fresh type variables, in the first equation and obtain

```
InputField (h11 INVALID) (h12 INVALID)
=?= InputField Int INVALID
```

We conclude by two further standard steps in higher-order unification: imitation  $h11 \mapsto \lambda x. Int$  and projection  $h12 \mapsto \lambda x. x$ , which completely reduce the equation. The second equation is reduced analogously.

Analogous problems arise when we try to pass no input handles to the `actionOnSubmit` or a list of input handles:

```
submit () actionOnSubmit'
submit [f1, f2] actionOnSubmit''
```

They may be addressed in an analogous manner as for pairs: either by introducing explicit lifting types `F0` and `FL` (as it is done in the current version of WASH/CGI) or by using higher-order unification in the type checker.

We feel that the introduction of explicit lifting types, as forced by Haskell, is particularly inappropriate for an embedded domain-specific language like WASH/CGI. Experts can program around this problem by introducing specialized wrapper data types like `F2` above and enclosing input fields into suitable wrappers. However, we argue that this is inconvenient, leads to obscure programs, and is hard to explain to non-expert users, which are the intended customers of WASH/CGI. Using our proposed extension, there is much less to explain because it is possible to directly use Haskell’s standard data structures, like pairs and lists.

## 1.4 What is Higher-Order Unification Anyway?

In the preceding subsections we have already hinted at using higher-order unification [11] in the type checker. To be able to discuss the pros and cons of using higher-order unification for type checking purposes, we give a short and rather informal explanation what higher-order unification is in this subsection.

The comparison to the better known notion of first-order unification might be helpful for some readers: in first-order unification, the underlying formal languages are first-order terms over a given signature of function symbols and a set of first-order variables. A unification problem is a pair of terms, and its solution is a substitution that makes both terms syntactically equal.

In contrast to that, the language considered for higher-order unification is the simply-typed  $\lambda$ -calculus with constants. Again, a unification problem is a pair of terms, and a solution of it is a substitution of the (free) variables such that the two resulting terms both have the same normal form. The difference to first-order unification is clear cut: substituted functions do not compose only from given function symbols, but can consist of arbitrarily complex “anonymous”  $\lambda$ -expressions. Since  $\lambda$ -terms are usually compared with respect to

an equational theory, we likewise search for solutions that result in “equal” terms—that is, for terms with the same normal form.

Although higher-order unification is undecidable—Hilbert’s tenth problem can be reduced to it— [4], there exist an algorithm, known as Huet’s algorithm [11], that is a sound and complete semi-decision procedure for higher-order unification problems. In the following we will briefly sketch the algorithm formulated as a set of transformation rules working on finite sets of term equations.

For each step an applicable rule is chosen to transform the actual set of equations  $E$  into a new one. We silently assume that terms occurring in equations are always in long  $\beta\eta$ -normal form—that is, a term is first reduced to its normal form using the usual  $\beta$ -reduction rules, and then  $\eta$ -expanded to exactly reflect its type structure. The decision whether a rule is applicable to an equation is based on the nature of the head symbols of both terms. We call a term *rigid* if its head symbol is a constant or a bound (universal) variable, and we call it *flexible* if its head symbol is a free variable.

In case one of the equations  $E$  is a *flex-flex pair*, the *decomposition* rule applies:

$$\Lambda x_1 \dots x_m . c \ t_1 \dots t_n \ =?= \ \Lambda x_1 \dots x_m . c \ t_1' \dots t_n'$$

decomposes into  $n$  different equations—each pair of arguments creates a new equation—

$$\Lambda x_1 \dots x_m . t_1 \ =?= \ \Lambda x_1 \dots x_m . t_1', \dots$$

whereas bound variables as head symbol act similar to constants. In case the head symbols differ, the procedure fails immediately.

There are two rules that cover *flex-rigid pairs*. To match the rigid side, any substitution for the flex variable must eventually result in a term with the same head symbol as on the rigid side. There are two possibilities where this symbol may come from: first, the substitution for the free variable  $f$  may introduce the symbol itself—this is called *imitation*. The equations  $E$

$$\Lambda x_1 \dots x_k . f \ t_1 \dots t_n \ =?= \ \Lambda x_1 \dots x_k . c \ t_1' \dots t_m', \ E'$$

are substituted by the preliminary solution for  $f$

$$f \mapsto \Lambda x_1 \dots x_n . c \ (g_1 \ x_1 \dots x_n) \dots (g_m \ x_1 \dots x_n),$$

where  $g_1, \dots, g_m$  are fresh variables, that are substituted later on. The other possibility is, that the needed symbol comes from one of the arguments of  $f$ . The *projection* rule handles this case—the applied substitution projects one of the arguments of  $f$  to the head position

$$f \mapsto \Lambda x_1 \dots x_n . x_i \ (g_1 \ x_1 \dots x_n) \dots (g_j \ x_1 \dots x_n)$$

whereas the number of arguments of  $x_i$  is determined by its type. Clearly, unless the kind of  $f$  is  $*$  there is a non-deterministic choice for each *flex-rigid pair* as to which rule to apply.

Provided that there exists at least one constant  $c$  of kind  $*$ , we know that a *flex-flex pair* always has at least one solution: an equation

$$\Lambda x_1 \dots x_k . f \ t_1 \dots t_n \ =?= \ \Lambda x_1 \dots x_k . g \ t_1' \dots t_m'$$

can be solved by substituting

$$\begin{aligned} f &\mapsto \Lambda x_1 \dots x_n . c, \\ g &\mapsto \Lambda x_1 \dots x_m . c. \end{aligned}$$

Hence, only the first three kind of rules are successively applied until only *flex-flex pairs* remain.

## 1.5 Why Higher-Order Unification Won’t Work

We argue that full higher-order unification (besides being undecidable) introduces too much non-determinism to produce useful results. On the other hand, pattern unification [20], which is decidable and unitary, is too restrictive for our intended applications. Our conclusion is that we need the power of higher-order unification, but we must restrict the non-determinism by providing guidance to the algorithm. Hence, we set out to investigate guided higher-order unification.

To exhibit the problems of higher-order unification, let us reconsider the example unification of Subsection 1.3. After a number of simplification steps (imitation, decomposition, and imitation, again) we arrived at the equations

$$\begin{aligned} h11 \text{ INVALID} &=?= \text{Int} \\ h12 \text{ INVALID} &=?= \text{INVALID} \end{aligned}$$

which we solved by the imitation  $h11 \mapsto \Lambda x . \text{Int}$  and the projection  $h12 \mapsto \Lambda x . x$  (in this case, imitation would also work:  $h11 \mapsto \Lambda x . \text{INVALID}$ ).

Now suppose we start from

$$h \ a \ =?= \ (\text{InputField Int INVALID}, \text{InputField Char INVALID})$$

where  $a$  is a type variable. All the steps of the previous subsection go through, up to the point where above equations are reached:

$$\begin{aligned} h11 \ a \ =?= \ \text{Int} \\ h12 \ a \ =?= \ \text{INVALID} \end{aligned}$$

At this point, the desired choice is to unify  $a$  with  $\text{INVALID}$  and have  $h11$  ignore its argument, using the imitation  $h11 \mapsto \Lambda x . \text{Int}$  and the projection  $h12 \mapsto \Lambda x . x$ . However, this choice is only one out of three possible choices (choosing projection for both equations fails immediately) and from the point of view of the unification algorithm, all choices look equally plausible. Hence, the algorithm has no way of telling if imitation or projection is desired.

To see why higher-order patterns are too restrictive for our purposes, it is sufficient to consider the term  $h \ \text{INVALID}$  that occurs in the above example. In this term the free variable  $h$  has a constant argument  $\text{INVALID}$ . However, in a pattern (as well as in a relaxed pattern [35]), the arguments of free variables are restricted to distinct bound variables, as in  $\Lambda x . h \ x$ .

## 1.6 What Haskell Does

Let’s have a look at the definition of Haskell for a hint at what can be done. By ruling out lambdas from the type language, Haskell reduces the unification problem to a first-order one. That is:

- In case of a flex-rigid pair, Haskell always chooses imitation.
- If the kind of the flexible head variable does not match the kind of the constructor, then a prefix of suitable kind of the constructor term is substituted for the head variable. If no such prefix exists, the unification fails.
- Flex-flex pairs are treated in the same way as flex-rigid pairs.
- Decomposition applies also in the flex-flex case.

## 1.7 Our Proposal

Our approach is to restrict the syntax of lambda terms so that projection terms are ruled out. Hence, the unification algorithm can

$$\frac{\Sigma, \Gamma \vdash_{k_0} a : \Gamma(a) \quad \Sigma, \Gamma \vdash_{k_0} c : \Sigma(c) \quad \Sigma, \Gamma \vdash_{k_0} t_1 : \kappa_2 \rightarrow \kappa_1 \quad \Sigma, \Gamma \vdash_{k_0} t_2 : \kappa_2}{\Sigma, \Gamma \vdash_{k_0} t_1 t_2 : \kappa_1}$$

**Figure 1. Kinding Judgement**

always choose imitation in the case of a flex-rigid pair. We further restrict the calculus to Church’s  $\lambda I$  terms, *i.e.*, terms where lambda abstraction can only abstract variables that occur free. This restriction enables us to deal with flex-flex pairs almost in the same way as Haskell does: decomposition is applicable to flex-flex pairs with identical head variable and the imitation-style treatment of unconstrained flex-flex pairs is also sound.

It remains to consider the case of flex-flex and flex-rigid pairs where a head variable is restricted by a type class constraint. In the thus restricted flex-rigid case, we do *not* use the general substitution as in Huet’s imitation rule, but rather *the instance declaration for the predicate*. This way, we avoid subsequent choices between imitation and projection by simply taking them in the instance declaration. Flex-flex pairs, where the restrictions on the head variables are different, are not simplified on the spot but rather deferred and kept as constraints (as customary in higher-order unification). This deferral departs from Haskell’s practice (and thus leads to different inferred types) but it is necessary because the instance substitution for the restricted head variable cannot be chosen without the guidance of a rigid term. At program level, we expect that all such flex-flex constraints are resolved (or give rise to typing errors) because top-level programs are usually not polymorphic. Unresolved flex-flex constraints at this point give rise to *unresolved overloading errors*.

## 2 The Formal Type System

Before developing the formal system, we fix some notation. We write  $\overline{x}_n = x_1 x_2 \dots x_n$  to abbreviate lists of syntactic phrases. Furthermore,  $t(\overline{t}_n) = t t_1 t_2 \dots t_n$  is  $n$ -fold nested function application, and  $\overline{\kappa}_n \rightarrow \kappa$  is the corresponding kind  $\kappa_1 \rightarrow \kappa_2 \rightarrow \dots \rightarrow \kappa$ . For lambda calculus, we employ the standard notation and definitions of Barendregt [1]:  $FV(t)$  for the free variables,  $\equiv$  for syntactic identity. We employ Barendregt’s variable convention.

Functions are total set-theoretic functions unless otherwise noted and  $A \rightarrow B$  denotes the set of such functions between sets  $A$  and  $B$ . We write  $\lambda x.e$  for the function  $\{(x, e) \mid x \in A\} \in A \rightarrow B$ .

A final note on our notation for variables and constructors (constants): Haskell’s constructors are capitalized and variables are written in lower case, but the standard in logic programming is exactly the opposite. We decided to break both standards, by distinguishing the two syntactic categories by name instead of by capitalization. We let  $c$  range over constructors and  $a$  range over type variables.

### 2.1 Haskell’s Type Language

The type language of Haskell is generated by the grammar

$$t ::= a \mid c \mid t t$$

where  $a$  is a type variable,  $c$  is a constant (a type constructor), and  $t t$  is type application. This language is itself typed according to the rules of the simply-typed lambda calculus. To avoid confusion,

types of types are called kinds, where kinds are generated by the grammar

$$\kappa ::= * \mid \kappa \rightarrow \kappa$$

The kind  $*$  is the only base kind and it categorizes all types. The kind  $\kappa_1 \rightarrow \kappa_2$  categorizes all (type) constructor functions that map a constructor of kind  $\kappa_1$  to one of kind  $\kappa_2$ . Figure 1 contains the well-known kinding rules. For convenience, we use kinded type variables  $a^\kappa$  or  $f^\kappa$  from now on and drop the kind annotation if the kind is not important. With kinded variables, each term  $t$  has exactly one kind, which we write as  $Kind_\Sigma(t)$ .

Since unification for this type language boils down to many-sorted (syntactic) term unification (aka kind-preserving unification [12, 9]), it is possible to extend an ML-style type inference algorithm to support higher-order types while retaining its soundness and completeness properties [12, 14]. In the general case, the extension with higher-order polymorphism requires higher-order unification in the type inference algorithm [32], which is undecidable [11].

### 2.2 Syntax

Our type language extends Haskell’s type language by a restricted notion of lambda abstraction. These restrictions are sufficient to retain the most important features of kind-preserving unification. We must distinguish between variables that may be substituted by unification and variables that are introduced in a lambda abstraction. Hence, we call variables which are substituted for during unification *existential variables*, ranged over by  $a, f, g$ , and variables introduced by lambda abstraction *universal variables*, ranged over by  $x, y^3$ . While existential variables always occur free in a type term, universal variables may occur free or bound<sup>4</sup>.

The judgement  $\Sigma, \Gamma \vdash_k t : \kappa$  restricts the judgement for  $\vdash_{k_0}$  to a set of lambda terms in normal form as shown in Figure 2. The restricted lambda terms in the thus defined set  $\Lambda_{\text{GHOU}}$  have a head symbol that is either a constant or an existential variable, and universal variables must always occur free in the corresponding lambda bodies (as in Church’s  $\lambda I$ -calculus).

To stay inside this restricted language, we define substitution so that it performs  $\beta$ -reduction on-the-fly. Since this is standard practice in the literature on logical frameworks, we only give some illustrative cases of the definition: substituting a lambda expression for an existential variable. The definition takes advantage of the assumption that flexible functions are fully  $\eta$ -expanded and that universal variables never appear in function position.

$$\begin{aligned} (\lambda \overline{x}_m.t)[f \mapsto \lambda \overline{y}_n.t] &= \lambda \overline{x}_m.(t[f \mapsto \lambda \overline{y}_n.t]) \\ (f(\overline{t}_n))[f \mapsto \lambda \overline{y}_n.t] &= t[\overline{y}_n \mapsto \overline{t}_n] \\ (c(\overline{t}_m))[f \mapsto \lambda \overline{y}_n.t] &= c(\overline{t}_m[f \mapsto \lambda \overline{y}_n.t]) \\ x[f \mapsto \lambda \overline{y}_n.t] &= x \end{aligned}$$

Due to the restriction on universal variables, substitution for them can never create a  $\beta$ -redex.

On top of this type language, a (constrained) *type scheme* has the form

$$s ::= \forall \overline{a}_i.(P, E) \Rightarrow t$$

<sup>3</sup>Note the connection to unification under mixed prefix [21].

<sup>4</sup>This distinction is often avoided in treatments of higher-order unification, by considering only terms in long  $\beta\eta$ -normal form. Then free and bound variables are immediately apparent.

$$\begin{array}{c}
(k\text{-var-ex}) \quad \Sigma, \Gamma \vdash_k f : \Gamma(f) \\
(k\text{-var-univ}) \quad \Sigma, \Gamma \vdash_k x : \Gamma(x) \\
(k\text{-const}) \quad \Sigma, \Gamma \vdash_k c : \Sigma(c) \\
(k\text{-app}) \quad \frac{\Sigma, \Gamma \vdash_k t_1 : \kappa_2 \rightarrow \kappa_1 \quad \Sigma, \Gamma \vdash_k t_2 : \kappa_2}{\Sigma, \Gamma \vdash_k t_1 t_2 : \kappa_1} \quad \text{if } t_1 \text{ is neither lambda nor universal variable} \\
(k\text{-lam-l}) \quad \frac{\Sigma, \Gamma \vdash_k c : \overline{\kappa_n} \rightarrow * \quad \Sigma, \Gamma \{ \overline{x_m} : \overline{\kappa'_m} \} \vdash_k t_i : \kappa_i}{\Sigma, \Gamma \vdash_k \lambda \overline{x_m}. c (\overline{t'_n}) : \overline{\kappa'_m} \rightarrow *} \quad \text{if } \overline{x_m} \subseteq FV(\overline{t'_n}) \\
(k\text{-lam-l}) \quad \frac{\Sigma, \Gamma \vdash_k f : \overline{\kappa_n} \rightarrow * \quad \Sigma, \Gamma \{ \overline{x_m} : \overline{\kappa'_m} \} \vdash_k t_i : \kappa_i}{\Sigma, \Gamma \vdash_k \lambda \overline{x_m}. f (\overline{t'_n}) : \overline{\kappa'_m} \rightarrow *} \quad \text{if } \overline{x_m} \subseteq FV(\overline{t'_n})
\end{array}$$

Figure 2. Kinding Judgement for Restricted Lambda-Calculus

where  $t$  is a type term of kind  $*$ , the  $\overline{a_i}$  are type variables, and  $(P, E)$  is a *constrained unification problem*. That is

- $P = \{ \overline{\pi_k(g_k)} \}$  is a set of class predicates (each  $\pi_i$  is the name of a type class) on existential variables and
- $E = \{ \overline{t_n =^? t'_n} \}$  is a set of equations on type terms.

A *program* is a triple  $(Classes, Instances, e)$  where

- *Classes* is a set of class declarations of the form

$$\mathbf{class} \pi(f) \mathbf{where} x : \forall \overline{a_i}. (P, E) \Rightarrow t$$

where  $FV(\forall \overline{a_i}. (P, E) \Rightarrow t) = \{f\}$ . It defines a class named  $\pi$ , introduces a type variable  $f$  that ranges over members of the class, and defines the type scheme of the single member value,  $x$ .

- *Instances* is a set of instance declarations of the form

$$\mathbf{inst} \overline{\pi_k(g_k)} \Rightarrow \pi(\lambda \overline{x_m}. c (\overline{t'_n})) \mathbf{where} x = e$$

where

- each free variable occurs at most once in  $\lambda \overline{x_m}. c (\overline{t'_n})$ ,
- $\{ \overline{x_m} \} \subseteq FV(c (\overline{t'_n}))$ , and
- each  $t'_i$  is either one of the  $\overline{x_m}$  or it has the form  $g (\overline{y_l})$  where  $g \notin \{ \overline{x_m} \}$  and  $\{ \overline{y_l} \} \subseteq \{ \overline{x_m} \}$ .

This declaration defines an instance of class  $\pi$  for a type starting with type constructor  $c$ . The arguments of  $c$  are restricted according to the type class predicates  $\overline{\pi_k(g_k)}$ . **For each pair  $c$  and  $\pi$  there is at most one instance declaration.**

- $e$  is an expression at the value level, defined by

$$\begin{array}{l|l}
e ::= v & \text{term variables} \\
\quad | \lambda v. e & \text{lambda abstraction} \\
\quad | e e & \text{application} \\
\quad | \text{let } v = e \text{ in } e & \text{let expression}
\end{array}$$

Our restriction on instance declarations is a simple generalization of Haskell98's restriction. For the time being, we ignore the expression in the instance declarations, which declares the member's value. For purposes of type inference, we are only interested in

- the set *Instances*, and
- the typing environment  $\Gamma_0$  generated by *Classes*:

$$\begin{array}{l}
\Gamma_0(x) = \forall f, \overline{a_i}. (\pi(f), P, E) \Rightarrow t \\
\text{iff } \mathbf{class} \pi(f) \mathbf{where} x : \forall \overline{a_i}. (P, E) \Rightarrow t \in \text{Classes}
\end{array}$$

### 2.3 Semantics

Interestingly, we need to define a semantics for our type language. We have not been able to prove soundness and completeness of the transformation rules for our unification algorithm using purely syntactical reasoning. Our semantics is given in terms of a Henkin model [24]. The underlying mathematical structures of Henkin models are typed applicative structures:

DEFINITION 1. A *typed applicative structure*  $\mathcal{A}$  is a triple  $\langle A^\kappa, \mathbf{App}^{\kappa, \kappa'}, \mathbf{Const} \rangle$ , where

- $A^\kappa$  is a kind-indexed family of sets;
- $\mathbf{App}^{\kappa, \kappa'}$  is an indexed family of application operators  $\mathbf{App}^{\kappa, \kappa'} : A^{(\kappa \rightarrow \kappa')} \rightarrow A^\kappa \rightarrow A^{\kappa'}$ ; and
- $\mathbf{Const}$  is constant interpretation functions that maps each constant  $c$  into the appropriate set  $A^{\Sigma(c)}$ .

To become a Henkin model, a typed applicative structure  $\mathcal{A}$  must satisfy two additional criteria: it must be *extensional* and it must have a certain well-defined meaning function  $\mathcal{A}[\![ \cdot ]\!]$  (*environment model condition*). Or more formally:

DEFINITION 2. A *Henkin model* is a typed applicative structure  $\mathcal{A} = \langle A^\kappa, \mathbf{App}^{\kappa, \kappa'}, \mathbf{Const} \rangle$  where

- $\mathcal{A}$  is *extensional*, i.e., every  $\mathbf{App}$  is a one-to-one mapping from  $A^{(\kappa \rightarrow \kappa')}$  into  $A^\kappa \rightarrow A^{\kappa'}$ , and
- the meaning function  $\llbracket \Sigma, \Gamma \vdash_k t : \kappa \rrbracket \eta$  is a well-defined total function (given in Figure 3) where environments  $\eta$  are partial functions from variables to  $\bigcup_\kappa A^\kappa$  so that  $\eta \models \Gamma$ , that is,  $\text{dom}(\eta) = \text{dom}(\Gamma)$  and  $(\forall x \in \text{dom}(\eta)) \eta(x) \in A^{\Gamma(x)}$ .

We need a few definitions to state our intended Henkin model. First, we need a strong notion of two semantic values being different. The kind-indexed relation  $\perp_\kappa$  captures this notion. It is the usual equality at base types. At function types, it differs from the usual pointwise notion of inequality.

DEFINITION 3. Let  $\langle A^\kappa, \mathbf{App}^{\kappa, \kappa'}, \mathbf{Const} \rangle$  be a typed applicative structure. The kind-indexed binary relation  $\perp_\kappa$  (strongly different) is defined inductively.

- $x \perp_* y$  iff  $x \neq y$  where  $x, y \in A^*$ ;

$$\begin{aligned}
\llbracket \Sigma, \Gamma \vdash_k x : \kappa \rrbracket \eta &= \eta(x) \\
\llbracket \Sigma, \Gamma \vdash_k c : \kappa \rrbracket \eta &= \mathbf{Const}(c) \\
\llbracket \Sigma, \Gamma \vdash_k t_1 t_2 : \kappa \rrbracket \eta &= \mathbf{App}^{k, \kappa}(\llbracket \Sigma, \Gamma \vdash_k t_1 : \kappa' \rightarrow \kappa \rrbracket \eta) (\llbracket \Sigma, \Gamma \vdash_k t_2 : \kappa' \rrbracket \eta) \\
\llbracket \Sigma, \Gamma \vdash_k \lambda x'. t : \kappa' \rightarrow \kappa \rrbracket \eta &= \text{the unique } f \in A^{k' \rightarrow \kappa} \text{ s.th.} \\
&\quad \forall d \in A^{k'}. \mathbf{App} f d = \llbracket \Sigma, \Gamma \{x' : \kappa'\} \vdash_k t : \kappa \rrbracket \eta [x' \mapsto y]
\end{aligned}$$

**Figure 3. Environment Semantics**

- $f \perp_{\kappa' \rightarrow \kappa''} g$  (where  $f, g \in A^{k' \rightarrow \kappa''}$ ) iff
  - $(\forall x \in A^{k'}) f(x) \perp_{\kappa''} g(x)$  and
  - $(\forall x', x'' \in A^{k'}) x' \perp_{\kappa'} x'' \Rightarrow f(x') \perp_{\kappa''} g(x'')$ .

**DEFINITION 4.** A function  $f \in A^{k' \rightarrow \kappa''}$  is strongly injective iff  $(\forall x', x'' \in A^{k'}) x' \perp_{\kappa'} x'' \Rightarrow f(x') \perp_{\kappa''} f(x'')$ .

For our intended Henkin model for  $\Lambda_{\text{GHOU}}$ , we define the following typed applicative structure  $I = \langle A^k, \mathbf{App}^{k, k'}, \mathbf{Const} \rangle$ :

- $A^*$  is the set of closed  $\Lambda_{\text{GHOU}}$ -terms of kind  $*$ , i.e.,  $A^* = \{t \mid \Sigma, \emptyset \vdash_k t : *\}$ .
- $A^{k' \rightarrow \kappa''}$  is the set of strongly injective functions from  $A^{k'}$  to  $A^{\kappa''}$ , i.e.,  $A^{k' \rightarrow \kappa''} = \{f : A^{k'} \rightarrow A^{\kappa''} \mid f \text{ strongly injective}\}$ .
- $\mathbf{App}^{k, k'} : A^{k' \rightarrow \kappa'} \rightarrow A^k \rightarrow A^{\kappa'}$  is function application:  $f \mapsto (x \mapsto f(x))$ .
- $\mathbf{Const}(c)$  is the curried term formation operator for the constant symbol  $c$ .

Let's consider some examples for illustration, where we write the function type constructor,  $\rightarrow$ , as a prefix operator:

- $A^* \supseteq \{ \text{Int, Bool, Float, List Int, List Bool, List Float, Pair Int Int, Pair Int Bool, } \rightarrow \text{ Int Int, } \rightarrow \text{ Int Bool} \}$
- $A^{* \rightarrow *} \ni \lambda \mathcal{M}. \text{List } t$
- $A^{* \rightarrow * \rightarrow *} \supseteq \{ \lambda s. \lambda \mathcal{M}. \text{Pair } s t, \lambda s. \lambda \mathcal{M}. \rightarrow s t \}$
- $\mathbf{Const}(\text{Int}) = \text{Int}$
- $\mathbf{Const}(\text{List}) = \lambda \mathcal{M}. \text{List } t$
- $\mathbf{Const}(\text{Pair}) = \lambda s. \lambda \mathcal{M}. \text{Pair } s t$

**LEMMA 1.** For each  $c$ ,  $\mathbf{Const}(c)$  is strongly injective.

However, before we can accept this as a definition of a Henkin model, we need to make sure that

$$\begin{aligned}
&\text{the unique } f \in A^{k' \rightarrow \kappa} \text{ s.th.} \\
&\quad \forall d \in A^{k'}. \mathbf{App} f d = \llbracket \Sigma, \Gamma \{x' : \kappa'\} \vdash_k t : \kappa \rrbracket \eta [x' \mapsto y]
\end{aligned}$$

indeed exists, i.e., we have to prove that it is strongly injective. Fortunately, this comes out for free from the next theorem.

Every function that we deal with in  $\Lambda_{\text{GHOU}}$  must be (at least) injective so that unification can apply the decomposition rule without limitation. In brief, the decomposition rule for a variable, say  $f$ , requires that

$$f t_1 \dots t_n = f t'_1 \dots t'_n \quad \text{iff} \quad t_1 = t'_1 \quad \dots \quad t_n = t'_n.$$

Whereas the direction from right to left follows directly from the compatibility of  $\beta$ -equivalence, the direction from left to right requires more attention. In fact, it is only valid if  $f$  is always an injective function. Unfortunately, it is hard to prove directly that every functional  $\Lambda_{\text{GHOU}}$ -term is injective, but it is possible to prove the stronger claim of strong injectivity.

**THEOREM 1.** For all  $\Lambda_{\text{GHOU}}$ -terms  $t$ , it holds that:

for all  $\eta_1, \eta_2, \Gamma$  such that judgement  $J$  holds (where  $J$  is  $\Sigma, \Gamma \vdash_k t : \kappa$ ) and  $\eta_i \models \Gamma$ , for  $i = 1, 2$ , and for all  $X \subseteq FV(t)$ :

if for all  $x \in FV(t)$  either

- $x \notin X$  and  $\eta_1(x) = \eta_2(x)$  or
- $x \in X$  and  $\eta_1(x) \perp_{\Gamma(x)} \eta_2(x)$

then

- if  $\kappa = \kappa' \rightarrow \kappa''$  then  $\llbracket J \rrbracket \eta_i$  is strongly injective, for  $i = 1, 2$ ; and
- exactly one of the following is true
  - $X \cap FV(t) = \emptyset$  and  $\llbracket J \rrbracket \eta_1 = \llbracket J \rrbracket \eta_2$ ; or
  - $X \cap FV(t) \neq \emptyset$  and  $\llbracket J \rrbracket \eta_1 \perp_{\kappa} \llbracket J \rrbracket \eta_2$ .

**PROPOSITION 1.** The typed applicative structure  $I$  is a Henkin model for  $\Lambda_{\text{GHOU}}$ .

Next, we turn to solving equations in  $\Lambda_{\text{GHOU}}$ .

## 2.4 Guided Higher-Order Unification

Guided higher-order unification simplifies a constrained unification problem  $(P, E)$ <sup>5</sup>. We need some preliminaries to give the definition of a solution of  $(P, E)$ .

Instance satisfaction relates a set of instance declarations *Instances* to a predicate  $\pi$  applied to a type  $t$ .  $\text{Instances} \vdash \pi(t)$  holds exactly if the collected instance declarations indicate that type  $t$  belongs to the class  $\pi$ . In particular, a type constructor  $\lambda \overline{x}_m. c(\overline{t}_n)$  is an instance of  $\pi$  if there is an instance declaration

$$\mathbf{inst} \overline{\pi}_k(\overline{g}_k) \Rightarrow \pi(\lambda \overline{x}_m. c(\overline{t}'_n)) \in \text{Instances}$$

with the same head symbol  $c$ , and for each argument  $t'_i$  of  $c$  in this declaration

- if  $t'_i$  is  $x_i$ , the corresponding subterm  $t_i$  of  $c \overline{t}'_n$  is also  $x_i$ ,
- if  $t'_i$  is  $f \overline{y}_l$ , where  $f$  is unconstrained, the corresponding subterm  $t_i$  is an arbitrary term  $t$  of the same kind as  $f$ , and
- if  $t'_i$  is  $g_j \overline{y}_l$ , where  $g_j$  is one of the variables mentions in  $\overline{\pi}(g_k)$ , the corresponding subterm  $t_i$  of  $c \overline{t}'_n$  must be an instance of  $\pi_j$ .

<sup>5</sup>Since we are leaving some flex-flex pairs unresolved, it might be more appropriate to speak of pre-unification.

<i>(strip)</i>	$P, E \cup \{\lambda x.t =? \lambda x.t'\}, \sigma$	$\rightarrow P, E \cup \{t =? t'\}, \sigma$
<i>(decomp)</i>	$P, E \cup \{c(\bar{t}_n) =? c(\bar{t}'_n)\}, \sigma$	$\rightarrow P, E \cup \{\overline{t_n =? t'_n}\}, \sigma$
<i>(flex-rigid-1)</i>	$P, E \cup \{f^\kappa(\bar{t}_m) =? c(\bar{t}'_n)\}, \sigma$	$\rightarrow \theta(P), \theta(E \cup \{\overline{t_m =? t'_{l+m}}\}), \theta \circ \sigma$ where $\theta = f^\kappa \mapsto c(\bar{t}'_l)$ and $l + m = n$ and $(\forall \pi) \pi(f^\kappa) \notin P$ and $\text{Kind}_\Sigma(c(\bar{t}'_l)) = \kappa$ and $f^\kappa \notin FV(\bar{t}'_l)$
<i>(flex-rigid-2)</i>	$P \cup \{\pi(f)\}, E \cup \{f(\bar{t}_m) =? c(\bar{t}'_n)\}, \sigma$	$\rightarrow \theta(P) \cup \{\overline{\pi_k(g_k)}\}, \theta(E) \cup \{\overline{t''_n[x_m \mapsto \theta(t_m)] =? t'_n}\}, \theta \circ \sigma$ where $\theta = f \mapsto \lambda \bar{x}_m.c(\bar{t}''_n)$ and $\text{inst } \overline{\pi_k(g_k)} \Rightarrow \pi(\lambda \bar{x}_m.c(\bar{t}''_n))$ and $f \notin FV(\bar{t}'_n)$
<i>(flex-flex-1)</i>	$P, E \cup \{f^\kappa(\bar{t}_n) =? g^\kappa(\bar{t}'_n)\}, \sigma$	$\rightarrow \theta(P), \theta(E \cup \{\overline{t_n =? t'_n}\}), \theta \circ \sigma$ where $(\forall \pi) \pi(f^\kappa) \in P$ iff $\pi(g^\kappa) \in P$ or $\kappa = *$ and $\theta = f^\kappa \mapsto g^\kappa$
<i>(flex-flex-2)</i>	$P, E \cup \{f^\kappa(\bar{t}_m) =? g(\bar{t}'_n)\}, \sigma$	$\rightarrow \theta(P), \theta(E \cup \{\overline{t_m =? t'_{l+m}}\}), \theta \circ \sigma$ where $\theta = f^\kappa \mapsto g(\bar{t}'_l)$ and $l + m = n$ and $(\forall \pi) \pi(f^\kappa) \notin P$ and $\pi(g) \notin P$ or $\kappa = *$ and $\text{Kind}_\Sigma(g(\bar{t}'_l)) = \kappa$ and $f \notin FV(\bar{t}'_l)$

**Figure 4. Guided Higher-Order Unification**

These side conditions are enforced by an auxiliary judgement  $\text{Instances}, \overline{\pi_k(g_k)} \vdash t' \triangleright t$ .

DEFINITION 5. The instance satisfaction relation  $\text{Instances} \vdash \pi(t)$  is defined by

$$\frac{\text{inst } \overline{\pi_k(g_k)} \Rightarrow \pi(\lambda \bar{x}_m.c(\bar{t}'_n)) \in \text{Instances} \quad \text{Instances}, \overline{\pi_k(g_k)} \vdash t'_i \triangleright t_i \text{ for each } i}{\text{Instances} \vdash \pi(\lambda \bar{x}_m.c(\bar{t}'_n))}$$

$$\frac{\text{Instances}, \overline{\pi_k(g_k)} \vdash x_i \triangleright x_i}{\text{Instances}, \overline{\pi_k(g_k)} \vdash f(\bar{y}_l) \triangleright t(\bar{y}_l)}$$

$$\frac{\text{Kind}_\Sigma(f) = \text{Kind}_\Sigma(f)}{\text{Instances}, \overline{\pi_k(g_k)} \vdash f(\bar{y}_l) \triangleright t(\bar{y}_l)}$$

$$\frac{\text{Instances} \vdash \pi_j(\lambda \bar{y}_l.t)}{\text{Instances}, \overline{\pi_k(g_k)} \vdash g_j \bar{y}_l \triangleright t}$$

DEFINITION 6. A substitution  $\sigma$  is a solution to constrained unification problem  $(P, E)$ , written  $\sigma \models (P, E)$ , iff

- for each  $t =? t' \in E$ , it holds that  $\sigma(t) = \sigma(t')$ , and
- for each  $\pi(g) \in P$ ,  $\text{Instances} \vdash \pi(\sigma(g))$ .

The rewriting system in Figure 4 specifies the guided unification procedure. A configuration of the rewriting system is a triple  $P, E, \sigma$  where

- $P = \{\pi_1(g_1), \dots\}$  a set of predicates on existential variables;
- $E = \{t_1 =? t'_1, \dots\}$  a set of equations;
- $\sigma = \{g_1 \mapsto t_1, \dots\}$  a substitution of existential variables by  $\Lambda_{\text{GHOU}}$ -terms.

The rules are to be interpreted similarly as in the first stage of Nipkow's article [27]. All rules are closed under symmetry of the  $=?$  relation. In the *(strip)* rule, the bound variables on both sides can be assumed equal due to  $\alpha$ -conversion. This rule never fails: If one side does not start with a lambda, then the term is implicitly  $\eta$ -expanded. In the *(decomp)* rule, constants may also be universal variables (as customary in higher-order unification) but this case will not arise in  $\Lambda_{\text{GHOU}}$ . An attempt to unify two terms with different head symbols fails immediately, although this is not reflected in the rules.

Equations where one head symbol is an existential variable and the other head symbol is a constant are handled by the rules *(flex-rigid-1)* and *(flex-rigid-2)* respectively. The former rule handles the cases where there are no class constraints restricting the free variable  $f$ . Those cases mimic the current behavior of Haskell98: the free variable  $f$  gets substituted by the constant on the right hand side applied to leading arguments depending on the kinds involved. The rule only applies if all kinds match and the free variable  $f$  does not occur in the prefix of the right hand side. The latter rule is used to guide the unification process when determining a the substitution for the free variable  $f$ . If there is a constraint on the free variable  $f$ , we use the instance declarations of  $f$ 's predicate  $\pi$  as a substitution; given the side conditions on instance declarations and the head symbol  $c$ , this selection is deterministic.

If there is an equation with an existential variable on either side, we only handle the equation if we can definitely decide how to simplify it. This is either the case if both variables are applied to the same number of arguments and also both variable are constrained by exactly the same set of predicates, or we have a different number of arguments but no constraints at all. In the former case *(flex-flex-1)* we unify both variables and decompose the terms, in the latter case *(flex-flex-2)* we again mimic Haskell98 as in the *(flex-rigid-1)* rule.

The unification rules are sound and complete, that is, they do not change the set of solutions of a constrained unification problem.

LEMMA 2. *For any transformation  $P, E, \sigma \rightarrow P', E', \theta \circ \sigma$ , it holds that  $\psi \models (\theta(P), \theta(E))$  iff  $\psi \models (P', E')$ .*

## 2.5 Termination

A study of the proof for Lemma 2 reveals that the condition  $f \notin FV(\overline{t'_n})$  is not needed for proving soundness and completeness of rule (*flex-rigid-2*). However, this side condition is a necessary condition for the rules to terminate.

Suppose we are given a rule (*flex-rigid-2'*) defined just like rule (*flex-rigid-2*), but without the condition  $f \notin FV(\overline{t'_n})$ . The following configuration leads to non-termination:

$$\{\pi(f)\}, \{f x =^? c (f y)\}, \sigma$$

In the presence of the instance declaration  $\mathbf{inst} \overline{\pi(g)} \Rightarrow \pi(\lambda x.c (g x))$ , the configuration rewrites in one (*flex-rigid-2'*) step to

$$\{\pi(g)\}, \{g x =^? c (g y)\}, (f \mapsto \lambda x.c (g x)) \circ \sigma$$

Since the resulting predicate and equation are just renamings of the original ones, the same rule (*flex-rigid-2'*) applies over and over again. The problem is that the substitution for  $f$  introduces a variable  $g$  that matches up in the next step with the constant  $c$  which is also introduced by the same substitution.

Still, this restriction is not sufficient to ensure termination. The reason for this is that the occur check cannot be applied in the usual obvious way. Since some decompositions are delayed (cf. rules (*flex-flex-2*)) and a constrained flex-rigid match only leads to a partial substitution (*flex-rigid-2*), the occurrence of a variable in a substitution need not become obvious at some point. For that reason, we are relying on a heuristic to stop the unification procedure. Roughly, the heuristic marks each symbol (variable or constant) in the original set of equations with a unique label and labels each equation with the empty set. Whenever a rule applies, the newly generated equations inherit the label from the originating equation. The exception here is the (*flex-rigid-2*), where we add the pair of the label of variable  $f$  and the constant  $c$  to the set on the newly generated equations. The fresh variables generated by this rule application inherit the label from variable  $f$ . Rewriting steps (reporting failure to unify) if the pair  $(f, c)$  is already present in the set of label pairs of the equation triggering the rule (*flex-rigid-2*).

## 2.6 Type Inference

The type inference rules in Figure 5 are the straightforward adaption of Milner's algorithm  $W$  [22] to our extended type language.

The algorithm is formulated as a syntax-directed typing judgement  $\vdash^W$ . Each rule  $(P, E), \sigma, \Gamma \vdash^W e : t$  states for a certain form of expression  $e$  considered under typing assumptions  $\Gamma$  what its type  $t$  is and which constraints  $(P, E)$  the free type variables must satisfy considering a certain substitution  $\sigma$ .

The (*i-var*) and (*i-lam*) rules are standard. The (*i-app*) rule uses our guided higher-order unification procedure instead of a first-order unification algorithm. The (*i-let*) rule uses a generalization procedure  $Gen$ . Its purpose is to generate a type scheme by generalizing over all free variables that are not occurring free in the type assumptions and by adding only such constraints that are affected by free

$$\begin{array}{c}
 (i\text{-var}) \frac{\Gamma(x) = \forall \overline{a_i}. (P, E) \Rightarrow t \quad \overline{b_i} \text{ new}}{(P, E)[a_i \mapsto b_i], \emptyset, \Gamma \vdash^W x : t[a_i \mapsto b_i]} \\
 (i\text{-lam}) \frac{(P, E), \sigma, \Gamma \{x \mapsto a\} \vdash^W e : t \quad a \text{ new}}{(P, E), \sigma, \Gamma \vdash^W \lambda x. e : \sigma(a) \rightarrow t} \\
 (i\text{-app}) \frac{\begin{array}{c} (P_1, E_1), \sigma_1, \Gamma \vdash^W e_1 : t_1 \\ (P_2, E_2), \sigma_2, \sigma_1 \Gamma \vdash^W e_2 : t_2 \\ a \text{ new} \end{array}}{\sigma_2(P_1) \cup P_2, \sigma_2(E_1) \cup E_2 \cup \{\sigma_2(t_1) =^? t_2 \rightarrow a\} \rightarrow^* P_3, E_3, \sigma_3} \\
 \frac{}{(P_3, E_3), \sigma_3, \Gamma \vdash^W e_1 e_2 : \sigma_3(a)} \\
 (i\text{-let}) \frac{\begin{array}{c} (P_1, E_1), \sigma_1, \Gamma \vdash^W e_1 : t_1 \\ ((P_2, E_2), s) = Gen(\Gamma, P_1, E_1, t_1) \\ (P_3, E_3), \sigma_2, \sigma_1 \Gamma \{x \mapsto s\} \vdash^W e_2 : t_2 \\ (P, E) = (\sigma_2(P_2) \cup P_3, \sigma_2(E_2) \cup E_3) \end{array}}{(P, E), \sigma_2 \circ \sigma_1, \Gamma \vdash^W \text{let } x = e_1 \text{ in } e_2 : t_2}
 \end{array}$$

Figure 5. Type Inference

variables. This is specified as follows:

$$\begin{aligned}
 Gen(\Gamma, P, E, t) &= ((P_1, E_1), \forall \overline{a_i}. (P_2, E_2) \Rightarrow t) \\
 &\text{where } \{\overline{a_i}\} = FV(t) \setminus FV(\Gamma) \\
 &\text{and } P_2 = \{\pi(t) \in P \mid FV(t) \cap \{\overline{a_i}\} \neq \emptyset\} \\
 &\text{and } E_2 = \{t =^? t' \in E \mid FV(t =^? t') \cap \{\overline{a_i}\} \neq \emptyset\} \\
 &\text{and } E_1 = E \setminus E_2 \text{ and } P_1 = P \setminus P_2
 \end{aligned}$$

In addition, our implementation performs predicate normalization before generalization in (*i-let*).

DEFINITION 7 (PREDICATE NORMALIZATION).

$$\begin{aligned}
 norm(\emptyset) &= \emptyset \\
 norm(P \cup \{\pi(t)\}) &= \\
 &\text{if } t \equiv \lambda \overline{x_m}. c(\overline{t'_n}) \\
 &\text{and } \mathbf{inst} \overline{\pi_k(g_k)} \Rightarrow \pi(\lambda \overline{x_m}. c(\overline{t'_n})) \in Instances \\
 &\text{then } norm(P \cup \{\pi_j(\lambda \overline{y_l}. t_i) \mid t'_i = g_j(\overline{y_l})\}) \\
 &\text{else } norm(P) \cup \{\pi(t)\}
 \end{aligned}$$

Predicate normalization terminates always, due to the syntactic restriction on instance declarations. If normalization finds a suitable instance declaration in scope (using *Instances*), it decomposes the argument of the predicate  $\pi$  and recursively tries to normalize the arguments. If there is no suitable instance declaration (perhaps because  $t$  is  $\eta$ -equivalent to an existential variable), then the predicate is returned without change.

## 3 Evaluation

In this section, we reconsider the examples from Section 1 in the light of the proposed extensions. We also present an example which yields a typing with remaining flex-flex pairs.

### 3.1 Functors

In this example, our desire was to express the identity functor  $\Lambda x. x$  and functor composition  $\Lambda x. f (g x)$  using instances of the *Functor* class. It turns out that we cannot model the identity functor because  $\lambda x.x$  is not an  $\Lambda_{\text{GHOU}}$ -term and we cannot model

functor composition directly, either. However, with a little bit of foresight, we can make all functor instances composable! The idea is to write

```
instance (Functor g) =>
  Functor (λx.[g x]) where
  fmap h = g
  where g [] = []
        g (x:xs) = fmap h x : g xs
```

```
instance (Functor g) =>
  Functor (λx.Maybe (g x)) where ...
```

instead of the instances in Section 1.1. Hence, one use of the identity type `Id` is still required to start a stack of composable functors, but there is no need for an explicit composition operator.

### 3.2 Contravariant Functors

The example of the motivation section, where the function arrow is abstracted over its first parameter, works as desired.

```
instance OpFunctor (λx.x -> a) where
  opmap h g = g . h
```

### 3.3 WASH/CGI

The instance declarations for class `InputHandle` may now be written as follows:

```
-- as before
instance InputHandle (InputField a) where ...
instance InputHandle (RadioGroup a) where ...
-- instead of defining data type FL:
instance (InputHandle h) =>
  InputHandle (λx.[h x]) where ...
-- instead of defining data type F2:
instance (InputHandle h1, InputHandle h2) =>
  InputHandle (λx.(h1 x, h2 x)) where ...
-- lifting of F0 is not possible:
instance InputHandle F0 where ...
```

In the last case, the introduction of the `F0` data type cannot be avoided:

```
data F0 x = F0
```

Since `F0` stands for a constant function  $\lambda x.()$ , it does not fit into our framework because it is not a  $\lambda I$  term.

### 3.4 Flex-Flex Pairs

To show how a typing with remaining flex-flex pairs could occur, we present another example. Consider the following algebraic data type `T` and the associated function `fun`:

```
data T f = T (f Char) -- f has kind * -> *
fun (T x) = fmap ord x
```

The type constructor `T` abstracts over types that result of applying a type constructor `f` to the type of characters. The function `fun` takes a value of an instance of a type constructed from `T` and maps the library function `ord` over it using `fmap`. For simplicity, we assume the following typings:

```
fmap :: (Functor g) => (a -> b) -> g a -> g b
ord :: Char -> Int
```

For typing `fun`, the inferencer considers the pattern on the left hand side of the declaration. The result is a type assumption for the pattern variable `x`:

```
x :: f Char
```

When inferring the type of the body of the function `fun`, we first consider the first function application `fmap ord` and get the following unification problem:

```
Functor g, (a -> b) -> g a -> g b =?= (Char -> Int) -> c
```

Using the rules (*decomp*) and (*flex-rigid-1*), we get:

```
fmap ord :: (Functor g) => g Char -> g Int
```

The type inference of the second function application `(fmap ord) x` leads to the new unification problem:

```
Functor g, g Char -> g Int =?= f Char -> d
```

which simplifies to

```
Functor g, g Char =?= f Char, g Int =?= d
```

by using the (*decomp*) rule. The second equation can be eliminated using (*flex-flex-2*), but the first one is a remaining flex-flex pair since we cannot be sure that `f` and `g` necessarily is the same type constructor. Hence, the resulting type for `fun` is:

```
fun :: (Functor g, g Char =?= f Char) => T f -> g Int
```

### 3.5 Discussion

The two places where our framework seems to fall short of our expectations is that

- it does not include constant functions and projections, and
- it does not include the identity functions.

However, both additions are not desirable because they either defer parts of the unification or they lead to ambiguities.

The addition of constant functions and projections complicates the applicability of the rules for flex-flex pairs. Since these rules only hold for strongly injective functions, we have to make sure that neither head variable may be instantiated with a constant function later on. In addition, we have to make sure that no argument term “disappears”, for example, if the term only appears as the argument of a variable that may be instantiated with a constant function.

The addition of identity functions introduces ambiguities at the level of instance declarations. The typical way of introducing such an identity would be as in the example for `Functor`:

```
instance Functor (λx.x) where ...
```

This declaration would play the role of a default instance declaration. For example, consider the equation  $f a = ? \text{Tree Int}$  where `Functor f`. In the absence of an instance for `Tree`, we might use the default instance and obtain the equation  $a = ? \text{Tree Int}$ . It might be worthwhile considering such an extension. However, we refrained from doing so in this work for two reasons: On the one hand, this kind of “overlapping instances” has no counterpart in standard Haskell, and on the other hand, the resulting unification procedure will not be deterministic anymore.

We have omitted subclasses from our definitions. However, they are easy to add at the price of complicating the definition of instance satisfaction (Definition 5) in the usual way [16] and by insisting that corresponding terms in instance declarations match. The latter is easy to check using (decidable) pattern unification, since instance terms happen to be higher-order patterns [20].

## 4 Related Work

The most closely related work analyzes algorithms for equality and unification in the presence of notational definitions [34]. The authors introduce a notion of strictness of a lambda term,  $\lambda x.t$ , that enables them to prove that a strict term behaves like an injective function under  $\beta$ -reduction. Given a definition  $a = \lambda \overline{x}_m.t$ , where the term is strict in  $\overline{x}_m$ , equality of terms with head symbol  $a$  can be decided by just checking the arguments of  $a$  for equality, thus avoiding the need for expanding the definition of  $a$ . A similar improvement is possible in the unification algorithm. Our notion of a strongly injective function is also motivated by the desire for a strong decomposition rule in the unification algorithm (*flex-flex-1*). It would be interesting to compare the notion of strictness with strong injectivity, but this may not be straightforward because strictness is defined syntactically whereas strong injectivity is defined semantically, by recourse to a specific model.

### 4.1 Type Inference

A polymorphic typing discipline with a type reconstruction algorithm is one of the key features of a number of successful functional programming languages, like ML [23] and Haskell [9]. Starting from the algorithm of Hindley [10] and Milner [22], a variety of extensions has been proposed and implemented. Among them, the biggest boost has come from the introduction of parametric overloading, as it was suggested by a number of authors [18, 19, 38, 5]. This proposal was modified and integrated in the programming language Haskell in the form of type classes. A type class is an inductively defined set of types. A type class qualification (or class constraint) can be used to restrict a polymorphic type to the members of a particular class (or classes). A number of works have tried to explain type inference for Haskell from first principles [28, 29].

Subsequently, further extensions have been investigated. In his thesis [13], Jones has investigated qualified types. Qualified types can be instantiated to a range of interesting type systems, among them type classes, record types, and subtyping. This work has paved the way for more explorations. Multi-parameter type classes [31] generalize type classes from sets of types to inductively defined relations on types. Unfortunately, they come with non-trivial ambiguity problems that render them virtually useless. This problem has prompted Jones [17] to add functional dependencies to multi-parameter type classes. Their addition resolves many ambiguity problems arising in practice. The semantic foundation of this work has been laid in earlier work on simplification and improvement of type class constraints during type reconstruction [15].

In another line of work, type classes have been generalized to constructor classes [12]. A constructor class defines a set of type constructors. This work has also become part of the Haskell language, and it is indeed the basis for convenient programming with monads, which is in turn essential to write Haskell programs that use I/O. Type inference for constructor classes can be implemented using first-order many-sorted unification.

### 4.2 Higher-order Unification

At this point, we can only give a very brief impression of a complicated research area. More information may be found in Dowek's survey article [4]. Higher-order unification has first been considered by Huet [11]. Huet showed that the general problem is undecidable and gave a semi-algorithm (which may not terminate) for its solution. The undecidability result was later strengthened to include

second order terms, too [8, 7]. Miller [20] defined higher-order patterns, for which the unification problem is decidable and unitary, regardless of the order of the terms. In the monadic case, where all constant symbols are unary, unification is decidable for second order terms [6], but undecidable at higher orders [26]. Matching is believed decidable [40], but concrete results only exist for second and third order problems [11, 2, 3].

Our work is inspired by Miller's work on pattern unification [20] and its generalization by Prehofer [35]. For the implementation, Nipkow's transformation of a rule-based description of pattern unification into a readily implementable functional version proved very helpful, although our Haskell implementation can avoid some of its complications [27].

## 5 Conclusion

The construction of an extension of Haskell's type class system with a restricted notion of lambda abstraction is not a straightforward task. We have achieved to define a system that types a number of interesting examples that were not possible before, although some could be made to work at the price of some awkward programming.

Our extension is conservative in the sense that all programs typeable in Haskell98 are still typeable in the extended language. On the downside, type schemes get more complicated because the constraints now include the remaining flex-flex pairs where at least one head variable carries a predicate. In some contrived cases that involve data structures with arrow-kinded type parameters the extended language infers types with such constraints.

## References

- [1] H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.
- [2] G. Dowek. A second order pattern matching algorithm in the cube of typed  $\lambda$ -calculi. In *Proceedings of Mathematical Foundation of Computer Science Lecture Notes in Computer Science 520*, pages 151–160, 1991. Rapport de Recherche 1585, INRIA, 1992.
- [3] G. Dowek. Third order matching is decidable. In *Proceedings of the 1992 IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1992.
- [4] G. Dowek. Higher-order unification and matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 16, pages 1009–1062. North-Holland, 2001.
- [5] D. Duggan, G. V. Cormack, and J. Ophel. Kinded type inference for parametric overloading. *Acta Inf.*, 33(1):21–68, 1996.
- [6] W. M. Farmer. A unification algorithm for second-order monadic terms. *Annals of Pure and Applied Logic*, 39:131–174, 1988.
- [7] W. M. Farmer. Simple second-order languages for which unification is undecidable. *Theoretical Comput. Sci.*, 87(1):25–41, Sept. 1991.
- [8] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Comput. Sci.*, 13(2):225–230, Feb. 1981.
- [9] Haskell 98, a non-strict, purely functional language. <http://www.haskell.org/definition>, Dec. 1998.

- [10] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
- [11] G. Huet. A unification algorithm for typed  $\lambda$ -calculus. *Theoretical Comput. Sci.*, 1(1):27–57, 1975.
- [12] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In Arvind, editor, *Proc. Functional Programming Languages and Computer Architecture 1993*, pages 52–61, Copenhagen, Denmark, June 1993. ACM Press, New York.
- [13] M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK, 1994.
- [14] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, May 1995.
- [15] M. P. Jones. Simplifying and improving qualified types. In S. Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pages 160–169, La Jolla, CA, June 1995. ACM Press, New York.
- [16] M. P. Jones. Typing Haskell in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical Reports, 1999. <ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-1999/1999-28.pdf>.
- [17] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Proc. 9th European Symposium on Programming*, number 1782 in *Lecture Notes in Computer Science*, pages 230–244, Berlin, Germany, Mar. 2000. Springer-Verlag.
- [18] S. Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proc. 2nd European Symposium on Programming 1988*, number 300 in *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1988.
- [19] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, page x, San Francisco, California, USA, June 1992.
- [20] D. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
- [21] D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, Oct. 1992.
- [22] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.
- [23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [24] J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
- [25] G. Nadathur and D. Miller. An overview of  $\lambda$  PROLOG. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827, Seattle, 1988. ALP, IEEE, The MIT Press.
- [26] P. Narendran. Some remarks on second order unification. Technical Report 89/356/18, University of Calgary, July 1989.
- [27] T. Nipkow. Functional unification of higher-order patterns. In *Proc. of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.
- [28] T. Nipkow and C. Prehofer. Type checking type classes. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 409–418, Charleston, South Carolina, Jan. 1993. ACM Press.
- [29] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In J. Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, number 523 in *Lecture Notes in Computer Science*, pages 1–14, Cambridge, MA, 1991. Springer-Verlag.
- [30] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.
- [31] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: An exploration of the design space. In J. Launchbury, editor, *Proc. of the Haskell Workshop*, Amsterdam, The Netherlands, June 1997. Yale University Research Report YALEU/DCS/RR-1075.
- [32] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, 1988. ACM Press.
- [33] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
- [34] F. Pfenning and C. Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, number 1657 in *Lecture Notes in Computer Science*, pages 179–193, Kloster Irsee, Germany, Mar. 1998.
- [35] C. Prehofer. Decidable higher-order unification problems. In *Automated Deduction CADE-12, 12th International Conference on Automated Deduction*. Springer, 1994.
- [36] M. Schmidt-Schauß and K. U. Schulz. Decidability of bounded higher order unification. Technical Report Frank-15, Universität Frankfurt, 2001.
- [37] P. Thiemann. Wash/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages, Proceedings of the Fourth International Workshop, PADL'02*, number 2257 in *Lecture Notes in Computer Science*, pages 192–208, Portland, OR, USA, Jan. 2002. Springer-Verlag.
- [38] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, Jan. 1989. ACM Press.
- [39] Web authoring system in Haskell (WASH). <http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH>, Mar. 2001.
- [40] D. A. Wolfram. *The Clausal Theory of Types*. Cambridge tracts in Theoretical Computer Science. Cambridge University Press, 1993.