Lexically Scoped Type Annotations

Martin Sulzmann
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Jeremy Wazny
Department of Computer Science and Software Engineering
University of Melbourne, Vic. 3010, Australia
jeremyrw@cs.mu.oz.au

Abstract

We observe that closed type annotations impose some serious restrictions on programs making use of typing features such as polymorphic recursion, type classes and guarded recursive data types. The type inferencer often cannot be supplied with the necessary amount of information to type check many reasonable programs. To address this, we introduce a novel form of lexically scoped annotation as an extension of Hindley/Milner. We further show that type inference based on algorithm $\mathcal{W}$ or its variants is not well-suited to deal with type annotations in general. Hence, we propose a novel inference scheme which improves over all previous formulations we are aware of. Our approach has been fully implemented as part of the Chameleon system (an experimental version of Haskell).

1. Introduction

Type annotations are a common feature found in many programming languages. The assumption made in Haskell 98 [10] is that type annotations are *closed*, i.e. all variables appearing in such annotations are implicitly universally quantified.

We show that the restriction to closed annotations unnecessary limits the expressiveness of the languages when it comes to more advanced type extensions. In our first example, we take a look at guarded recursive data types (GRDTs) [34]. We make use of Haskell style syntax [10].

EXAMPLE 1. Consider the following program.

```
data ZERO
data SUCC xs
data List a b = (b=ZERO) => Nil
              | forall b'. (b=SUCC b') =>
                Cons a (List a b')
replace :: Eq a => a->a->List a b->List a b
replace x y xs =
  let f NIL = Nil
      f (Cons z xs) = if z == x then Cons y (f xs)
                      else Cons z (f xs)
  in f xs
```

We define a GRDT `List` which is parameterized in an element type a and a parameter b recording the length of the list. The novelty here is that, depending on the constructor, the length parameter of the list – a type – may change. This is in contrast to algebraic data types where the type would need to be the same, regardless of the constructor.

Function `replace` takes two arguments and a list and replaces all occurrences of the first by the second in the list. The type of `replace` guarantees that the length of the list remains the same. Note that `Eq` refers to the equality type class in Haskell. In some initial environment we find the corresponding method $(==) : \forall a.Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$. For convenience, we have defined a local function `f` to avoid passing around `x` and `y`. However, we are in trouble now.

Type inference for GRDTs is a fairly challenging problem unless we provide a sufficient amount of additional type information [15, 24, 27]. The problem is related to polymorphic recursion [11], where similarly, we would need to guess the most common generalization of the type of recursive functions calls. Indeed, all inference systems we are aware of fail for the above program unless we provide further type information. Unfortunately, there is no closed annotation for `f` such that the program type checks. Hence, the programmer will need to rewrite the program such that the type inferencer succeeds. □

It would be much simpler if we could write a lexically scoped annotation of the form `f::List a c->List a c` where the a refers to `replace`'s annotation and the c is fresh, i.e. universally quantified. In Chameleon, such annotations are supported and the program would type check.

Note that some authors [1, 2] have shown how to encode GRDT style behaviour with existential types. However, these encodings still rely on polymorphic recursion and for such an extension we can make a similar argument which shows that closed annotations are insufficient.

There are further situations where lexically scoped annotations improve over closed annotations. E.g., in the type class context [33] lexically scoped annotations are useful to resolve "ambiguous" constraints.

EXAMPLE 2. Consider

```
class Show a where
  show :: a->String
  read :: String->a
g :: Show a => String -> a -> (String,String)
g s x = (show (read s), show x)
```

where we make use of a standard Haskell type class, except that, for brevity, we lump `read` into the `Show` class. Note that sub-expression `show (read s)` is of type $String$ under the constraint $Show\ b$ where $s : String$ (in an intermediate step we find that `read s` has type b under constraint $Show\ b$). Variable b neither appears in the type nor in the environment. Hence, we cannot automatically decide which `Show` instance to pick. Furthermore, the subsumption test, i.e. checking for correctness of the annotation, may fail The type inferencer would need to guess that $b = a$. We do not want to rely on such guesses, though, since they may may lead to arbitrary program behavior. The proper solution would be to provide an annotation, but there is no closed annotation we can assign so that the program type checks. As in the previous example, we could rewrite the program such that it type checks. However, this is tedious and leads to clumsy program code. The more natural solution is to write `(show (read s))::a` where a refers to `g`'s annotation. In Chameleon we can, and the program type checks then. □

The above examples indicate that there is a need for more powerful annotation mechanisms. In this paper, we introduce *lexically scoped* annotations where the scope of an annotation comprises the entire body of that declaration. Such annotations allow the programmer to give more precise "hints" to the type inferencer.

A surprising observation is that type inference based on algorithm $\mathcal{W}$ or its variants show some interesting behavior when extended to deal with type annotations. In the following example, we make use of multi-parameter type classes and closed annotations which are supported by Hugs [12] and GHC [7] (two popular Haskell implementations).

EXAMPLE 3. Here is a variation of a program which appeared on the Haskell-Cafe mailing list [13].[1]

```
class Foo a b where foo :: a->b->Int
instance Foo Int b
instance Foo Bool b

p y = (let f :: c -> Int
           f x = foo y x
       in f, y + (1::Int))


q y = (y + (1::Int), let f :: c -> Int
                         f x = foo y x
                     in f)
```

We introduce a two-parameter type class `Foo` which comes with a method `foo` of type $\forall a, b.Foo\ a\ b \Rightarrow a \rightarrow b \rightarrow Bool$. The two instance declarations state that $Foo\ Int\ b$ and $Foo\ Bool\ b$ hold for any b.

Consider functions `p` and `q`. In each case the subexpression `y+(1::Int)` forces `y` to be of type Int. The body of function `f x = foo y x` generates the constraint $Foo\ Int\ t_x$ where t_x is the type of `x`. Note that this constraint is equivalent to $True$ due to the instance declaration. We find that `f` has the inferred type $\forall t_x.t_x \rightarrow Int$. We need to check that this type subsumes the annotated type `f::c->Int` which is interpreted as $\forall c.c \rightarrow Int$. The subsumption condition is obviously met. Hence, expressions `p` and `q` are well-typed. □

Let's see what Hugs [12] and GHC [7] say. Expression `p` is accepted by Hugs but rejected by GHC whereas GHC accepts `q` which is rejected by Hugs! Both implementation rely on algorithm $\mathcal{W}$ style inference. The above example shows that different traversals of the abstract syntax tree (AST) may lead to different results. Hugs seems to favor a right-first traversal of the AST whereas GHC favors a left-first traversal.

The above example seems contrived, but we believe that such undesirable behavior will occur more frequently for advanced typing features such as GRDTs and type classes in combination with lexically scoped annotations. Hence, we seek a new, superior inference scheme rather than burdening the programmer with even more annotations.

In summary our contributions are:

- We introduce novel forms of lexically scoped type annotations as extensions of the Hindley/Milner system with constraints (Section 2.1). A novelty of our approach is that a scoped variable may be solely introduced by the constraint part of a type (Section 2.2).
- We verify soundness of our extension by giving an equivalence and type preserving translation to System F (Section 2.3).
- We show that GHC-style pattern annotations [14] can be encoded in our system via a simple source-to-source translation (Section 2.4).
- We introduce a superior inference scheme where inference is independent of a particular traversal of the AST (Section 3).

In Section 4 we conclude the paper and also discuss related work. Due to space limitations, formal proof details have been moved to the appendix. Throughout the paper, we assume that the reader is familiar with the concepts of substitutions, most general unifiers, first-order logic etc [16, 23].

$$\text{(Var-}\forall\text{E)}\ \frac{(x:\sigma)\in\Gamma \quad P_p \wedge C \vdash \sigma \preceq t}{C,V,\Gamma \vdash x : t} \qquad \text{(Abs)}\ \frac{C,V,\Gamma.x:t_1 \vdash e : t_2}{C,V,\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad \text{(App)}\ \frac{C,V,\Gamma \vdash e_1 : t_1 \to t_2 \quad C,V,\Gamma \vdash e_2 : t_1}{C,V,\Gamma \vdash e_1 e_2 : t_2}$$

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

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

$$\text{(LetA2)}\ \frac{\bar{a} = fv(C,t) - fv(V,\Gamma) \quad C_1,V.\bar{a},\Gamma \vdash e_1 : t_1 \quad P_p \models C_1 \supset \exists\bar{a}.(C \wedge t = t_1) \quad \bar{b} = fv(C_1,t_1) - fv(V,\Gamma) \quad C_2,V,\Gamma.(g:\forall\bar{b}.C_1 \Rightarrow t_1) \vdash e_2 : t_2}{C_2,V,\Gamma \vdash \mathsf{let}\ \begin{array}{l} g ::: C \Rightarrow t \\ g = e_1 \end{array}\ \mathsf{in}\ e_2 : t_2}$$

Figure 1. Hindley/Milner with Lexically Scoped Annotations

2. Lexically Scoped Type Annotations

We first present the formal underpinnings of an extension of the Hindley/Milner system with constraints and lexically scoped type annotations (Section 2.1). Then, we consider a few more examples and state some basic results about how scoping affects typing (Section 2.2). We also verify that our formulation is sound by means of a type-preserving translation to System F (Section 2.3). Finally, we show that GHC-style scoped pattern annotations can be easily encoded by our system via a simple source-to-source translation (Section 2.4).

2.1 Type System

We start off by defining the language of expressions and types.

Expressions	e	::=	$x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ f = e\ \mathsf{in}\ e \mid$
			$\mathsf{let}\ \begin{array}{l} a(f) \\ f = e \end{array}\ \mathsf{in}\ e \mid$
Annot	$a(f)$	::=	$f :: C \Rightarrow t \mid f ::: C \Rightarrow t$
Types	t	::=	$a \mid t \to t \mid T\ \bar{t}$
Type Schemes	σ	::=	$t \mid \forall\bar{a}.C \Rightarrow t$
Constraints	C	::=	$t = t \mid U\ \bar{t} \mid C \wedge C$

In our language we find two forms of annotations. In addition, to the common form of *universal* annotations `::` we also allow for *existential* annotations `:::`. In contrast to the Haskell 98 assumption where annotations are assumed to be closed ours are lexically scoped (we will explain their exact meaning shortly). Note that annotations do not have explicit quantifiers. Hence, variables appearing in those annotations cannot be renamed without changing the programs meaning, and once introduced, these variable names cannot be rebound. In our current implementation we leave all quantifiers implicit but this could be easily changed. For instance, it could be worthwhile to introduce some syntax such as `forall a` and `exists a` to explicitly mark the introduction

[1] For concreteness, we annotate `1` with `Int` because in Haskell `1` is in general only a number, with type $Num\ a \Rightarrow a$.

of a new lexically scoped (universal or existential) type variable. Further, we require type declarations to be conjoined with their value declarations. Such a restriction is not enforced in Haskell 98.

Before we get into the technical details of our system we introduce some notation and make some assumptions explicit. We generally use type writer font for program text and math font for types, constraints etc. We write $\bar{o}$ to denote a sequence of objects $o_1, ..., o_n$ and $\overline{o : t}$ to denote $o_1 : t_1, ..., o_n : t_n$. W.l.o.g., we assume that lambda-bound and let-bound variables have been α-renamed to avoid name clashes. We record these variables in some environment Γ. We assume that primitive functions are recorded in some initial environment Γ_{init}. For brevity, we write $\Gamma.f : \sigma$ to denote set extension $\Gamma \cup \{f : \sigma\}$. Similarly, we write $V.\bar{a}$ to denote $V \cup \bar{a}$. We write "-" to denote set subtraction. We write $fv(o)$ to denote the set of free variables in some object o with the exception that $fv(\{x_1 : \sigma_1, ..., x_n : \sigma_n\})$ denotes $fv(\sigma_1, ..., \sigma_n)$. In case objects have binders, e.g. $\forall a$, we assume that $fv(\forall a.o) = fv([b/a]o) - \{b\}$ where b is a fresh variable and $[b/a]$ a renaming.

Our type language consists of variables a, type constructors T and type application, e.g. $T\ a$. We use common Haskell notation for writing function, pair, list types etc.

We find two kinds of constraints. *Equation* constraints $t_1 = t_2$ among types t_1 and t_2 and *user-defined* constraints $U\ \bar{t}$. Sometimes, we refer to these constraints as *primitive* constraints. We often use set notation for conjunction of constraints and use "," as a short-hand for $\wedge$. For the moment, we assume that the meaning of user-defined constraints is defined by a first-order logic formula P_p. We commonly refer to P_p as the *program theory*. Thus, we can customize our system to deal with the many extensions found for Hindley/Milner. In later parts of the paper (Section 3) when discussing type inference, we will introduce a more specific formalism to describe P_p.

To describe well-typing of expressions we make use of judgments of the form $C, V, \Gamma \vdash e : t$ where C is a constraint, V refers to the set of lexically scoped variables, Γ refers to the set of lambda-bound variables, predefined and user-defined functions, e is an expression and t is a type. The novel part is the set V which plays a similar role for lexically scoped type variables as Γ for lambda-bound variables. Note that we leave the program theory P_p implicit. None of the typing rules affect P_p, hence, we can assume that P_p is fixed for a given expression. We say a judgment is *valid* iff there is a derivation w.r.t. the rules found in Figure 1. Each valid judgment implies that the expression is well-typed.

Let us take a look at the typing rules in detail. Rules (LetA) and (LetA2) are novel, but we will first consider the remaining rules which are mostly standard. In rule (Var-$\forall$E), we build a type instance by making use of a *subsumption* relation among types. Recall that $P_p \wedge C$ denotes the logical interpretation of the program theory conjoined with the current constraint. In general, we define subsumption as follow. Let F be a first-order formula, $(\forall \bar{a}.C \Rightarrow t)$ and $(\forall \bar{a'}.C' \Rightarrow t')$ be two type schemes. Then, $F \vdash (\forall \bar{a}.C \Rightarrow t) \preceq (\forall \bar{a'}.C' \Rightarrow t')$ iff $F \models C' \supset \exists \bar{a}.(C, t = t')$ assuming that there are no name clashes among bound variables $\bar{a}$ and $\bar{a'}$. We assume that $\supset$ refers to Boolean implication and $\models$ refers to the model-theoretic entailment relation. In such a situation, we say that $(\forall \bar{a}.C \Rightarrow t)$ *subsumes* $(\forall \bar{a'}.C' \Rightarrow t')$ w.r.t. F. Note that $F \vdash (\forall \bar{a}.C \Rightarrow t) \preceq t'$ holds iff $F \models \exists \bar{a}.C, t = t'$ (assuming t' is a short-hand for $\forall \emptyset.True \Rightarrow t'$). The last statement holds iff $F \models \phi(C, t = t')$ where ϕ is an appropriate substitution with domain $\bar{a}$. The final statement is the common side condition for building a type instance of a type scheme. The observant reader may notice that we could generalize our notion of subsumption to accommodate subtyping among types. However, our main target is a Haskell-style language with type classes, and the combination of subtyping and type classes is known to be problematic.

Rules (Abs) and (App) are straightforward. Note that the quantifier introduction rule is coupled with the rules for let statements. Rule (Let) is mostly standard with the exception that we additionally need to exclude type variables in scope from being bound. We note that our formulation is more "lazy" as compared to other schemes. We refer to [20, 31] for a discussion.

Rule (LetA) deals with a lexically scoped (universal) annotation. Variables in $C_1 \Rightarrow t_1$ may refer to variables appearing in some annotation from the enclosing scope. These variables are recorded in V. Hence, when building an explicitly quantified type scheme we only quantify over variables which are not in C_2, Γ and V. Our

assumption is that the scope of an annotation is the entire body of that declaration. Hence, when typing the let-body e_1 we extend V by the newly introduced scoped variables. Note that the annotation does *not* scope over the let-body (although the value definition does). The inspiration for this form of annotation is taken from System F. This will become clear in Section 2.3. Note that rule (LetA) allows for polymorphic recursive functions. We extend the environment with $g : \forall \bar{a}.C_1 \Rightarrow t_1$ when typing the function body. For simplicity, we omit the rule to deal with monomorphic recursive functions.

Rule (LetA2) allows us to add type information to typing derivations via a lexically scoped existential type annotation. The side condition $P_p \models C_1 \supset \exists \bar{a}.(C \wedge t = t_1)$ states that the resulting type and constraint for e_1 must at least contain t and C for some $\bar{a}$ which are the freshly introduced variables in this annotation. Note that existential annotations follow the same scoping rules as universal annotations and even share the same scope.

For simplicity, we omit the description of extensions such as algebraic data types and its generalizations [17, 34]. We could extend our system to include such extensions based on our own work [25]. Also, we omit the formal treatment of class and instance declarations, frequently used in examples, which can be modelled by some appropriate environment Γ_{init} and program theory P_p. See [26] for details. Note that our actual implementation supports all these additional features.

2.2 Examples

In our first example we make use of an existential annotation.

EXAMPLE 4. Consider

```
f ::: a->b->(c,c)
f x y = (read x, read y)
```

where we make use of the class declaration from Example 2. Our intention here is to ensure that both occurrences of `read` use the same instance of the `Show` class. □

Obviously, we could easily give a universal annotation here. The advantage of existential annotations is that we do not need to provide exact type information.

Although not explicitly in our syntax we can mix universal and existential quantifiers.

EXAMPLE 5. Consider the following variation of Example 1 where we have supplied function `f` with some extra arguments.

```
f x y NIL = Nil
f x y (Cons z xs) = if z == x then Cons y (f x y xs)
                    else Cons z (f x y xs)
```

As mentioned, it is often crucial to provide type annotations for functions operating on GRDT. Here we will at least need to ensure that the length of the list is universally quantified but we do not need to make any specific assumptions about the element type. Hence, it would be sufficient to annotate `f` with following type $\exists a.\forall b.a \to a \to List\ a\ b \to List\ a\ b$. [2] Although our system does not explicitely support such mixture of existential and universal annotations we encode them by having an outer existential annotation followed by an inner universal annotation. Recall that existential and universal annotations share the same scope.

```
f' = let f'' ::: a->c
         f'' _ =
          let f :: a->a->List a b->List a b
              f x y NIL = Nil
              f x y (Cons z xs) =
                if z == x then Cons y (f x y xs)
                else Cons z (f x y xs)
          in f
     in f'' undefined
```

[2] There may be a potential confusion with existential types. Such types can only be annotated by not inferred.

In the above we assume that variable c is fresh. The program is now sufficiently annotated such that a type inferencer will succeed. Note that function `f'` is operationally equivalent to `f` under a non-strict semantics. It is straightforward to adapt our encoding to work under a strict semantics. □

Via a similar encoding we can in fact encode pattern annotations. In the upcoming Section 2.4 we consider the specific case of GHC-style scoped pattern annotations.

A special feature of our system is that may refer to a variable which only appears in the constraint component of an annotation.

EXAMPLE 6. We introduce a multi-parameter type class `Eval` with method `eval` of type $\forall a, b.Eval\ a\ b \Rightarrow a \rightarrow b$ to describe that evaluation of some term with type a evaluates to a value of type b. The plan is to write a type-directed evaluator for a simply-typed language. Due to space limitations we can only provide a highly simplified version of the program.

```
class Eval a b where eval::a->b
f::Eval a (b,c) => a->b
f x = let g = eval x
      in fst g
```

The observant reader will notice that `f`'s annotation is "ambiguous" because variable c does not appear in the type component. Such annotations may lead to difficulties when giving meaning to programs. But this is not an issue here.

The problem is that `g`'s program text gives rise to the constraint $Eval\ a\ (b, c')$. Silently, we assume that `x` has type a and take into account `f`'s annotation. Note that the annotation only provides $Eval\ a\ (b, c)$ which does not match $Eval\ a\ (b, c')$. Of course, we could guess that $c = c'$ but as said earlier this may lead to some undesired behavior. A better approach is to tell the inferencer that the right component of the pair arising out of `g`'s program text is connected to variable c from `f`'s annotation. However, we cannot express this information via a closed annotation. The problem can be solved by supplying the annotation `g::(b,c)`. Note that variable c only appears in the constraint component of `f`'s annotation. □

Note that in the above program we could replace the universal annotation `g::(b,c)` by the existential annotation `g:::(b,c)` with no change in result. In fact, there is no difference between universal and existential annotations if all annotation variables have been introduced the the outer scope.

It's also clear that we could even replace `g::(b,c)` by `g::Eval a (b,c)=>(b,c)`. We simply propagate constraints from outer annotations inwards. Such behaviour is more formally captured by the following lemma. We assume that notation $e_1[e_2]$ refers to one occurrence of expression e_2 in e_1. We use $\equiv$ to denote syntactic equality.

LEMMA 1 (Constraint Scoping). *Let*

$$e_1 \equiv e\left[\begin{array}{l} f :: C_1 \Rightarrow t_1 \\ f = e'\left[\begin{array}{l} g :: C_2 \Rightarrow t_2 \\ g = e'' \end{array}\right] \end{array}\right]$$

and

$$e_2 \equiv e\left[\begin{array}{l} f :: C_1 \Rightarrow t_1 \\ f = e'\left[\begin{array}{l} g :: C_1 \wedge C_2 \Rightarrow t_2 \\ g = e'' \end{array}\right] \end{array}\right]$$

Then $C, V, \Gamma \vdash e_1 : t$ *iff* $C, V, \Gamma \vdash e_2 : t$.

We can restate the above result for any combination of universal and existential annotations.

2.3 Type Soundness

It is fairly easy to show that our typing rules are sound by giving an equivalence preserving transformation where the resulting expression is typable in System F. Following the approach in [9], the idea is to introduce explicit (type) lambda-abstractions for all (universal) type variables. In particular, this stresses the point that the scope of a lexically scoped annotation comprises the entire body of that declaration.

EXAMPLE 7. The translation of

```
f::Eval a (b,c) => a->b
f x = let g :: (b,c)
          g = eval x
      in fst g
```

yields

```
f' = Λ a,b,c.λ eval. λ x. let g' :: (b,c)
                              g' = eval x
                          in fst g'
```

where `f'` has type $\forall a, b, c.(a \to (b, c)) \to a \to b$. For convenience, we make use of a System F variant which supports let statements with explicit annotations. Note that we also need to convert type class constraint in annotations such $Eval\ a\ (b, c)$ into additional function parameters, here $a \to (b, c)$, following the translation scheme for type classes in [8]. □

THEOREM 1 (Type Soundness). *The typing rules in Figure 1 are sound.*

Proof details can be found in Appendix A.1.

2.4 Encoding GHC-style Scoped Annotations

The system by Peyton-Jones and Shields [14] (referred to as the PJS system hereafter) and implemented in GHC [7] allows for closed (universal) function annotations as well as scoped pattern annotations (a.k.a. type sharing annotations) of the form $f\ (p_1 :: t_1)\ ...\ (p_n :: t_n)\ ::\ t\ =\ e$. Type variables in t_1,...,t_n and t scope over all patterns p_1,...,p_n and the function body. The handling of scoped type variables as described by Peyton Jones and Shields, and implemented in GHC is straightforward. W.l.o.g., we may assume that full pattern and result annotations are provided. Otherwise, we can simply introduce annotations $:: a$ where a is a fresh variable. In the PJS system, they maintain a mapping Φ from variables which appear in pattern annotations, to monotypes in their internal representation (which is used for all type-related operations.) Before processing any signature declaration, they first apply Φ to rename any already-bound variables appearing in the type.

In our system, we can easily encode PJS scoped pattern annotations via existential annotations. We give a source-to-source translation. For this purpose, we introduce judgments $V \vdash e \rightsquigarrow e'$ where e is the original PJS program and e' is the translated program. Variable set V holds all variables which appeared in pattern and result annotations in the PJS system in some previous translation step. For simplicity, we only provide the essential

translation rules:

$$\frac{\begin{array}{c} V \vdash e \rightsquigarrow e' \\ \pi \text{ variable renaming on } \mathit{fv}(C,t) - V \end{array}}{V \vdash \begin{array}{ll} f :: C \Rightarrow t & f :: \pi(C) \Rightarrow \pi(t) \\ f = e & \rightsquigarrow \quad f = e' \end{array}}$$

$$\frac{V \cup \mathit{fv}(t_1,...,t_n,t) \vdash e \rightsquigarrow e' \quad ps \text{ fresh}}{f\ (p_1 :: t_1)\ ...\ (p_n :: t_n)\ ::\ t\ =\ e}$$

$$\rightsquigarrow$$

$$V \vdash \quad f\ p_1\ ...\ p_n\ =\ \begin{array}{ll} \text{let} & ps\ :::\ (t,t_1,...,t_n) \\ & ps\ =\ (e',p_1,...,p_n) \\ \text{in} & \mathit{fst}_{(n+1)}\ ps \end{array}$$

In the above, $\mathit{fst}_{(n+1)}$ denotes projection of the first component of a $n+1$ tuple.

In the first case, we rename all variables in function annotations which are not connected to variables in pattern annotations. We shortly explain why this is necessary. The second case translates pattern and result annotations by introducing a `:::` annotation on the right-hand side of the function equation. We make use of n-tuples. Note that variables in the `:::` annotation scope over all patterns and the function body.

EXAMPLE 8. We explain why renaming in the first case above is necessary. Consider the following PJS program.

```
f::a->Bool->Bool
f x (y::a) = (True::a)
```

Note that the lexically scoped variable a introduced in the pattern has no connection to variable a from `f`'s annotation. Assume we neglect to rename variables in function annotations (first case above). Then, we find the following result.

```
f::a->Bool->Bool
f x y = let ps:::(c,b,a)
            ps = (True,x,y)
        in fst ps
```

Note that we have introduced fresh variables b and c for the missing first pattern annotation and the result annotation. For simplicity, we took a short-cut and transformed $True :: a$ to $True$. The point is that the resulting program is not typable in our system. Variable a is shared by an existential and universal annotation and the program text of `ps` forces $a = Bool$. Hence, the above program is not typable. Therefore, we *must* rename all variables in function annotations which are not connected to pattern annotations. Our actual transformation yields the following

```
f::a'->Bool->Bool
f x y = let ps::(c,b,a)
            ps = (True,x,y)
        in fst ps
```

which is typable. □

The above example highlights a subtle difference between the PJS system and ours. In the PJS system, we can write universal annotations and existential pattern annotations at the same level. In our system, we can only write one annotation per function definition. Hence, in our translation we push the PJS-style existential pattern annotation one level inwards and must therefore introduce a new let definition.

We conclude this section by stating that our translation is semantic and type-preserving. The following result is only stated for the standard Hindley/Milner fragment.[3] We write $\Gamma \vdash_{PJS} e : t$ to denote expression e has type t under environment Γ in the PJS system.

THEOREM 2. *Let* $\Gamma \vdash_{PJS} e : t$. *Then,* $\emptyset \vdash e \leadsto e'$ *such that* $\Gamma \vdash e' : t$ *and* e *and* e' *are semantically equivalent.*

3. Type Inference

Type inference is something we take for granted these days. Damas and Milner [3] showed in their pioneering work that the Hindley/Milner system has principal types. Thus, algorithm $\mathcal{W}$ finds the "best" type if the program is typable. The addition of type annotations is often considered as a straightforward extension. Unfortunately, the inference problem for annotations is far from trivial as shown by Example 3. The deeper reason behind the failure of Hugs and GHC is a loss of principal types once we move to more advanced type extensions which support type annotations.

EXAMPLE 9. Here is another example which circulated on the Haskell mailing list [21] and reappeared in [4].

```
class Foo a b where foo :: a->b->Int
instance Foo Int b
instance Foo Bool b

test y = let f :: c->Int
             f x = foo y x
         in f y
```

Note that `test` may be given types $Int \to Int$ and $Bool \to Int$. The constraint $Foo\ t_y\ t_x$ arising out of the program text can be either satisfied by $t_y = Int$ or $t_y = Bool$. However, the "principal" type of `test` is of the form $\forall t_y.(\forall t_x.Foo\ t_y\ t_x) \Rightarrow t_y \to Int$. Note that the system we are considering does not allow for constraints of the form $\forall t_x.Foo\ t_y\ t_x$. Hence, the above example has no (expressible) principal type. □

We draw the following conclusions:

1. The success of inference may depend on the order of traversal of the abstract syntax tree (see Example 3).
2. We lose principal types because the constraint domain is not expressive enough (see Example 9).

There is not much that we can do about (2). The obvious solution seems to allow for more expressive constraints. Then, we can almost trivially achieve principal types at the expense of less-readable types and much costlier inference. Note that inference for guarded recursive data types shares the same problem. Hence, our main focus is to avoid (1).

One of our main tricks is to postpone the subsumption check by generating more expressive constraints. It is standard knowledge that verifying whether type $\forall \bar{a}.C \Rightarrow t$ subsumes type $\forall \bar{a}'.C' \Rightarrow t'$ can logically expressed by the implication constraint $\forall \bar{a}'.(C' \supset \exists \bar{a}.(C \wedge t = t'))$ (w.l.o.g. we assume that there are no name clashes among $\bar{a}$ and $\bar{a}'$). Eventually, such implication constraints must be solved in terms of the constraint domain allowable in the type system. However, the advantage now is that we can make use of type information from the outer context while solving.

Another objective of our inference method is to remove the "left-right" bias when typing let statements. E.g., in an algorithm $\mathcal{W}$-style scheme we first generate constraints out of the body of the let function, solve them, build the type of the function, and then we can move on to consider the body of the let statement. In our scheme, we make use of instantiation constraints $f(t, l, v)$ which denote that function `f` has type t under the environment l of lambda-bound variables and the environment v of lexically scoped variables. The actual

[3] The formal description of the PJS system in [14] only covers the standard Hindley/Milner system.

$$\text{(Var-x)}\quad \frac{(x:t_1)\in\Gamma \quad t_2 \text{ fresh}}{V,E,\Gamma,x \vdash_C (t_2 = t_1 \mid t_2)}$$

$$\text{(Var-f)}\quad \frac{t,l,v \text{ fresh} \quad f\in E \quad C=\{f(t,l,v), l=\langle\overline{t_x}\rangle, v=\langle\bar{a}\rangle\}}{\{\overline{x:t_x}\},\bar{a},E,f \vdash_C (C \mid t)}$$

$$\text{(VarA-f)}\quad \frac{t,l,v \text{ fresh} \quad f_a\in E \quad C=\{f_a(t,l,v), l=\langle\overline{t_x}\rangle, v=\langle\bar{a}\rangle\}}{\{\overline{x:t_x}\},\bar{a},E,f \vdash_C (C \mid t)}$$

$$\text{(Abs)}\quad \frac{V,E,e,\Gamma.x:t_1 \vdash_C (C \mid t_2) \quad t_1,t_3 \text{ fresh}}{V,E,\Gamma,\lambda x.e \vdash_C (C, t_3 = t_1 \to t_2 \mid t_3)}$$

$$\text{(App)}\quad \frac{V,E,\Gamma,e_1 \vdash_C (C_1 \mid t_1) \quad V,E,\Gamma,e_2 \vdash_C (C_2 \mid t_2) \quad t_3 \text{ fresh}}{V,E,\Gamma,e_1\ e_2 \vdash_C (C_1, C_2, t_1 = t_2 \to t_3 \mid t_3)}$$

$$\text{(Let)}\quad \frac{V,E.f,\Gamma,e_2 \vdash_C (C \mid t)}{V,E,\Gamma,\mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_C (C \mid t)}$$

$$\text{(LetA2)}\quad \frac{V,E,\Gamma,e_2 \vdash_C (C \mid t)}{V,E,\Gamma,\mathsf{let}\ \begin{array}{l} f ::: C_1' \Rightarrow t_1' \\ f = e_1 \end{array}\ \mathsf{in}\ e_2 \vdash_C (C \mid t)}$$

$$\text{(LetA)}\quad \frac{\begin{array}{c} V,E.f_a,\Gamma,e_2 \vdash_C (C' \mid t') \quad \Gamma = \overline{x:t} \quad V = \bar{a} \quad \bar{b} = fv(C_1',t_1') - V \quad t,l,v \text{ fresh} \\ C'' = \{C', \forall\bar{b}.(C_1', t = t_1', l = \langle\bar{t}\rangle, v = \langle\bar{a},\bar{b}\rangle \supset f(t,l,v))\} \end{array}}{V,E,\Gamma,\mathsf{let}\ \begin{array}{l} f :: C_1' \Rightarrow t_1' \\ f = e_1 \end{array}\ \mathsf{in}\ e_2 \vdash_C (C'' \mid t')}$$

Figure 2. Constraint Generation

type of `f` is described by type relations $f(t,l,v) \Longleftrightarrow C$. Thus, we can visit the body of the let function and the body of the let statement independently. We apply a transformation akin to lambda-lifting but at the type level to ensure that all such type relations are available globally. Hence, we essentially obtain a constraint logic program making use of implication constraints rather than sets of primitive constraints. We can query the type of an expression by running the constraint logic program and solving implication constraints.

We continue in Section 3.1 where we translate expressions to implication constraints and type relations. Section 3.2 shows how to perform type inference in terms of constraint solving. Section 3.3 compares the benefits of our scheme to standard formulations such as $\mathcal{W}$.

3.1 Translation to Constraints and CHRs

The language of enriched constraints and type relations is as follow.

$$\begin{array}{lrcl} \text{Constraints} & D & ::= & t = t \mid U\ \bar{t} \mid D \wedge D \mid \\ & & & f(t,l,v) \mid f_a(t,l,v) \\ \text{ImpConstraints} & C & ::= & D \mid C \wedge C \mid \\ & & & \forall\bar{a}.(D \supset \exists\bar{b}.C) \\ \text{SIMP} & R & ::= & u \Longleftrightarrow C \end{array}$$

We extend the constraint domain from Section 2.1 with special-purpose user-defined constraints $f(t,l,v)$. We assume that for each function f there is such a constraint. In case f is universally type-annotated we assume another constraint $f_a(t,l,v)$. Commonly, constraint $f(t,l,v)$ refers to the "inference" result of function f and $f_a(t,l,v)$ refers to the "annotation". The domain of *implication* constraints is necessary to represent the subsumption condition. We adopt the notation that symbols C refer to implication constraints and symbols D refer to sets of equations and user-defined constraints.

$$\text{(Var)}\ V,E,\Gamma,v \vdash_R \emptyset \qquad \text{(App)}\ \frac{V,E,\Gamma,e_1 \vdash_R P_1 \quad V,E,\Gamma,e_2 \vdash_R P_2}{V,E,\Gamma,e_1\ e_2 \vdash_R P_1 \cup P_2} \qquad \text{(Abs)}\ \frac{V,E,\Gamma.x:t,e \vdash_R P \quad t \text{ fresh}}{V,E,\Gamma,\lambda x.e \vdash_R P}$$

$$\text{(Let)}\ \frac{\begin{array}{c}\Gamma = \overline{x:t} \quad V = \bar{a} \quad t,l,v,lr,vr \text{ fresh} \\ V,E,\Gamma,e_1 \vdash_R P_1 \quad V,E.g,\Gamma,e_2 \vdash_R P_2 \\ V,E,\Gamma,e_1 \vdash_C (C_1 \mid t_1) \\ P = P_1 \cup P_2 \cup \{f(t,l,v) \Longleftrightarrow l = \langle \bar{t} \mid lr\rangle, v = \langle \bar{a} \mid vr\rangle, t = t_1, C_1\}\end{array}}{V,E,\Gamma,\mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_R P}$$

$$\text{(LetA2)}\ \frac{\begin{array}{c}\Gamma = \overline{x:t} \quad V = \bar{a} \quad \bar{b} = fv(C_1',t_1') - V \quad t,l,v,lr,vr \text{ fresh} \\ V.\bar{b},E,\Gamma,e_1 \vdash_R P_1 \quad V,E.g,\Gamma,e_2 \vdash_R P_2 \quad V.\bar{b},E,\Gamma,e_1 \vdash_C (C_1 \mid t_1) \\ P = P_1 \cup P_2 \cup \{f(t,l,v) \Longleftrightarrow l = \langle \bar{t} \mid lr\rangle, v = \langle \bar{a},\bar{b} \mid vr\rangle, t = t_1, C_1, t = t_1', C_1'\}\end{array}}{V,E,\Gamma,\mathsf{let}\ \begin{array}{l} f ::: C_1' \Rightarrow t_1' \\ f = e_1 \end{array}\ \mathsf{in}\ e_2 \vdash_R P}$$

$$\text{(LetA)}\ \frac{\begin{array}{c}\Gamma = \overline{x:t} \quad V = \bar{a} \quad \bar{b} = fv(C_1',t_1') - V \\ V.\bar{b},E.f_a,\Gamma,e_1 \vdash_R P_1 \quad V,E.f_a,\Gamma,e_2 \vdash_R P_2 \\ V.\bar{b},E.f_a,\Gamma,e_1 \vdash_C (C_1 \mid t_1) \quad t,l,v,lr,vr \text{ fresh} \\ P = P_1 \cup P_2 \cup \left\{ \begin{array}{lcl} f_a(t,l,v) & \Longleftrightarrow & v = \langle \bar{a} \mid vr\rangle, t = t_1', C_1' \\ f(t,l,v) & \Longleftrightarrow & l = \langle \bar{t} \mid lr\rangle, v = \langle \bar{a},\bar{b} \mid vr\rangle, t = t_1, C_1 \end{array} \right\}\end{array}}{V,E,\Gamma,\mathsf{let}\ \begin{array}{l} f :: C_1' \Rightarrow t_1' \\ f = e_1 \end{array}\ \mathsf{in}\ e_2 \vdash_R P}$$

Figure 3. CHR Generation

We make use of *simplification* Constraint Handling Rules (CHRs) [5] of the form $u \Longleftrightarrow C$ where u refers to a user-defined constraint to describe function type relations. In fact, CHRs are also a highly useful mechanism to describe program theories.

EXAMPLE 10. The instance relations from Example 3 can be described by the following set of CHRs: $\{Foo\ Int\ b \Longleftrightarrow True \mid Foo\ Bool\ b \Longleftrightarrow True\}$. □

We note that CHR simplification rules are already Turing complete, however, we may want to take advantage of a richer set of CHRs for a more natural encoding of program theories. We skip the details and refer the interested reader to [26, 6]. The important point is that we can translate the entire typing problem to a set of CHRs and implication constraint.

The actual translation process is a refinement of the method described in [30]. In contrast to [30], we make use of implication constraints to represent the subsumption condition rather than checking for subsumption separately. Generation of implication constraints and CHRs if formulated in terms of two deduction systems described in Figures 2 and 3. Deduction rules are specified in terms of judgments of the form $V,E,\Gamma,e \vdash_C (C \mid t)$ and $V,E,\Gamma,e \vdash_R P$ where the set of lexically scoped variables V, environment E of all let-defined and pre-defined functions, environment Γ of lambda-bound variables, and expression e are input parameters and constraint C, type t and set of CHRs P are output parameters. We silently assume that sets V and Γ are

treated as ordered lists. Recall that $V.a$ is a short-hand for (now ordered) set extension $V \cup \{a\}$. For clarity, we introduce list notation on the level of types to manage environments representing the sequence of types of all lambda-bound variables and all lexically scoped type variables. E.g., we write $\langle l_1, \ldots, l_n \rangle$ to denote the list of types $l_1, \ldots, l_n$. We write $\langle l | r \rangle$ to denote the list of types with head l and tail r. We often write $\langle \overline{l} | r \rangle$ to denote list with elements $l_1, ..., l_n$ and tail r. It may happen that $\overline{l}$ denotes the empty set. In such situations, we assume that $\langle \emptyset | r \rangle$ is equivalent to r.

Let us take a closer look at the constraint generation rules in Figure 2. We find three rules dealing with variables. Rule (Var-x) deals with lambda-bound variables. There should be no surprises. Rule (Var-f) deals with unannotated functions, therefore, we query f's type via $f(t, l, v)$. We set l equal to the (ordered) sequence of types of all lambda-bound variables and v equal to the (ordered) sequence of all lexically scoped type variables which are in scope. Note that both sequences of types may be actually larger than found at f's definition site. Hence, we apply a simple trick and leave parameters l and v "open" at definition sites (see CHR generation rules (Let) and (LetA)). Rule (VarA-f) is similar but deals with type annotated functions. It is crucial to use $f_a(t, l, v)$ otherwise we may create cycles in case of polymorphic recursive functions. Rules (Abs) and (App) are standard. Note that in rule (Let) we only collect the constraints out of expression e_2 for simplicity. Strictly speaking, we also would need to collect the "effect" e_1 has on e_2. We can either demand that let-defined functions are realizable, i.e. actually used, or include an "anonymous" call to f in C. The technical details of anonymous calls are described in [29]. In rule (LetA) we additionally include an implication constraint to check the correctness of f's annotation. We state that by unifying the type components the constraint from f's annotations must imply the constraint arising out of f's function body. Note that the administrative constraint $l = \langle \overline{t} \rangle, v = \langle \overline{a}, \overline{b} \rangle$ must be included on the lhs of $\supset$ because the type relation for f is specified relative to the environments l and v. Further note that the quantifier $\forall \overline{b}$ ensures that the annotation is polymorphic enough. Rule (LetA2) is identical to (Let). Note that existential annotation do not need to be verified.

The CHR (i.e. type relation) generation rules are in Figure 3. Rules (Var), (Abs) and (App) are straightforward. In rule (Var) we assume that v refers to lambda-bound and user-defined functions. Rule (Let) is more interesting. Out of expressions e_1 and e_2 we collect the CHRs and generate the constraints arising out of e_1. Then, we build a new CHR for f where components l and v are appropriately constrained. We refer to this CHR as f's *inference* CHR. Note that l and v are left "open" (we define $l = \langle \overline{t} | lr \rangle, v = \langle \overline{a} | vr \rangle$). As discussed earlier, at use sites "calls" to f may include further types of lambda-bound variable and lexically scoped variables. In rule (LetA) we additionally build a CHR describing f's annotation. We refer to this CHR as f's *annotation* CHR. Note that the inference CHR refers to the entire set of lexically scoped variable (including those from the annotation) whereas the annotation CHR only refers to the variables from the enclosing scope. This is important because when building an explicit type scheme from a CHR we quantify over all variables which are not captured by l and v. In case of an annotation, we use f_a and therefore should be allowed to quantify over the freshly introduced variables. Note that we make use of newly introduced lexically scoped variables when processing e_1. Rule (LetA2) applies the same scoping rules as (LetA). Existential annotations do not imply an extra annotation CHR. Instead, we include the constraints from the annotation in the inference CHR.

3.2 Type Inference via Constraint Solving

The actual type inference is performed by solving constraints w.r.t. CHRs. The knowledgable reader will have noticed that our form of CHRs is *non-standard*. We allow for implication constraints on right-hand sides whereas the *standard* formulation only allows for sets of primitive constraints.

The operational semantics of standard CHRs can easily be described in terms of a rewriting transformation among constraint *stores*, i.e. sets of primitive constraints. We can apply a simplification CHR if we find a constraint in the store which matches the left-hand side of a rule. Then, we simply replace this constraint by the right-hand side of the CHR. Individual rule applications are denoted $D_1 \longrightarrow_R D_2$ whereas we write $D_1 \longrightarrow_P^* D_2$ to denote exhaustively applying of a set P of standard CHRs on a initial store D_1 yielding a final store D_2.

The solving process becomes less clear once we allow for implication constraints. For a given set P of (possibly non-standard) CHRs and a implication constraint C we need to find a constraint D such that D "solves" C w.r.t. P. Logically, we can describe this task as follow. Each CHR $U\,\bar{t} \iff C$ can logically be interpreted as $\forall\bar{a}.U\ \bar{t} \leftrightarrow (\exists\bar{b}.C)$ where $\bar{a} = \mathit{fv}(\bar{t})$ and $\bar{b} = \mathit{fv}(C) - \bar{a}$. Then, we say that D *solves* C w.r.t. P iff (1) $P \models D \supset C$ where P refers to the conjunction of logical interpretation of CHRs and (2) D consists of sets of primitive constraints excluding special-purpose user-defined constraints. Together, conditions (1) and (2) guarantee that we can find a solution which is representable in the constraint domain from Section 2.1.

We omit a concrete description of a particular solver for implication constraints. In our own work [25], we have already given a generic solver which is parameterized in terms of a standard CHR solver. Note that this solver applies to our setting by first unfolding all non-standard CHRs. For specific constraint domains such as equations we can even further improve solving. That is, find "better" solutions. Although, we will not be able to find "principal" solutions in general. Recall Example 9. Hence, we simply assume that we are given a sound implication solver satisfying the following definition. This will allow us to obtain sound inference (see upcoming Theorem 3).

DEFINITION 1 (Sound Implication Solver). *Let P be a set of non-standard CHRs and C a implication constraint. We say that a implication solver computes a valid result D, written $C \longrightarrow^*_P D$, iff D solves C w.r.t. P (i.e. $P \models D \supset C$ and D consists of primitive constraints excluding special-purpose constraints).*

Before performing inference we still need to translate primitive functions to CHRs. Consider an expression e under environment Γ. We assume that Γ can be split into a component Γ_{init} and Γ_λ such that $\mathit{fv}(\Gamma_{\mathit{init}}) \subseteq \mathit{fv}(\Gamma_\lambda)$ and types in Γ_λ are simple, i.e. not universally quantified. For each function $f : \forall\bar{a}'.C' \Rightarrow t'$ in Γ_{init} we generate a CHR $f(t,l,v) \iff t = t', C'$ where t and l are fresh. CHRs are recorded in $P_{E_{\mathit{init}}}$ and symbols f are recorded in E_{init}. In such a situation, we write $P_{E_{\mathit{init}}}, E_{\mathit{init}} \sim \Gamma_{\mathit{init}}, \Gamma_\lambda$.

Then, type inference proceeds as follows: We compute $\emptyset, E_{\mathit{init}}, \Gamma_\lambda, e \vdash_C (C \mid t)$ and $\emptyset, E_{\mathit{init}}, \Gamma_\lambda, e \vdash_R P$. Assume the set P_p of CHRs describes the program theory. To infer the type of e, we perform $C \longrightarrow^*_{P_{E_{\mathit{init}}}, P, P_p} D$. We use notation $D_=$ to refer to the set of equations in D and notation D_u to refer to the set of user-defined constraints in D. If $D_=$ is satisfiable, then e has type $\forall\bar{a}.\phi(D_u) \Rightarrow \phi(t)$ where $\phi = \mathit{mgu}(D_=)$ and $\bar{a} = \mathit{fv}(\phi D_u, \phi t) - \mathit{fv}(\phi\Gamma_\lambda)$. Note that in case of standard Hindley/Milner we have that D_u equals True.

EXAMPLE 11. Consider the following program

```
f :: a->a
f x = let g :: a->a
          g y = True
      in g x
```

which is ill-typed. The program text of `g` demands that a in `g`'s annotation must be Bool. However, variable a which was introduced by `f`'s annotation is universally quantified. Indeed, our inference scheme will detect that.

Here are the CHRs generated out of `f`'s program text:

$$\begin{array}{rcl}
f_a(t,l,v) & \iff & l = vr, t = a \to a \\
f(t,l,v) & \iff & l = lr, v = \langle a \mid vr\rangle, t = t_x \to t_1, \\
 & & g_a(t,l',v'), l' = \langle t_x\rangle, v' = \langle a\rangle, \\
 & & (t_g = a \to a, l'' = \langle t_x\rangle, v'' = \langle a\rangle \\
 & & \quad \supset g(t_g, l'', v'')) \\
g_a(t,l,v) & \iff & l = \langle vr\rangle, t = a \to a \\
g(t,l,v) & \iff & l = \langle t_x \mid tr\rangle, v = \langle a \mid vr\rangle, t = t_y \to \mathit{Bool}
\end{array}$$

Correctness of `f`'s annotation is stated via the following constraint $\forall a.t_f = a \to a, l = \langle\rangle, v = \langle a\rangle \supset f(t_f, l, v)$. Note that the constraint stating correctness of `g`'s annotation is part of the constraint on the rhs of `f`'s inference CHR.

Informally, implication CHR solving proceeds as follow. In some intermediate step, when solving $f(t_f, l, v)$, we come across the inner subsumption problem $(t_g = a \to a, l'' = \langle t_x \rangle, v'' = \langle a \rangle, g(t_g, l'', v''))$ which can only be solved by $a = Bool$ (see the definition of `g`'s inference CHR). Hence, $f(t_f, l, v) \longrightarrow^* v = \langle Bool | vr \rangle, D$ for some D. Then, $\forall a.t_f = a \to a, l = \langle\rangle, v = \langle a \rangle \supset f(t_f, l, v)$ can be simplified to $\forall a.v = \langle a \rangle \supset \exists vr.v = \langle Bool | vr \rangle$. However, this implication constraint can only be solved by $False$. Hence, type inference fails.

On the other hand, the following variation is correct.

```
f ::: a->a
f x = let g :: a->a
          g y = True
      in g x
```

Note that `f` has now an existential annotation. The translation to constraints and CHRs is almost the same as before. The difference is that there is no annotation CHR for `f` and constraint $t = a \to a$ is additionally included on the rhs of `f`'s inference CHR. When inferring `f`'s type we solve as before the inner subsumption problem by $a = Bool$. The type t of `f` is now constrained by $t = a \to a$. Hence, inference for `f` yields type $Bool \to Bool$. □

We can state that our inference scheme is sound. As a technical side condition, we demand that let-defined functions are "realizable", i.e. used in the program text. We say expression e is *let-realizable* iff for each sub-expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ we have that $x \in fv(e_2)$. We can easily drop this side condition by slightly refining our inference scheme which is described in detail in a previous version of this paper [29].

THEOREM 3 (Soundness). *Let P_p be a program theory and $P_{E_{init}}, E_{init} \sim \Gamma_{init}, \Gamma_\lambda, \emptyset, E_{init}, \Gamma_\lambda, e \vdash_C (C \mid t)$ and $\emptyset, E_{init}, \Gamma_\lambda, e \vdash_R P$ such that $C \longrightarrow^*_{P_{E_{init}}, P, P_p} D$ and $\phi = mgu(D_=)$ and e is let-realizable. Then, $\phi(D_u), \emptyset, \phi(\Gamma_{init} \cup \Gamma_\lambda) \vdash e : \phi t$.*

A proof can be found in Appendix A.3.

3.3 Comparison with Algorithm $\mathcal{W}$ Style Inference

We claim that any program typable in a system based on algorithm $\mathcal{W}$ is also typable in our scheme. The informal argument is that we are able to check for subsumption "later" due to our use of implication constraints. Hence, we can take advantage of possibly crucial type information, which has yet to arise, to solve implication constraints. e.g. we can type check Example 3. In fact, we even claim that our scheme is superior over algorithm $\mathcal{W}$ extended with implication constraints. Recall that a second advantage of our scheme is that we are more flexible when processing let statements.

EXAMPLE 12. Consider the following variation of Example 3.

```
r y = (y+(1::Int), let p z = let f::c->Int
                                 f x = foo y x
                             in f z
                   in p)
```

Assume we use a algorithm $\mathcal{W}$ style inference system where we generate implication constraints. At each let node we will need to solve these constraints because we need to build a type for the function definition. In the above program, we need to solve at node `let p`. More specifically, we encounter an implication constraint of the form $\forall t_x.True \supset Foo\ t_y\ t_x$ (expressing the subsumption condition for `f`'s annotation). If the algorithm $\mathcal{W}$ style system traverses the AST in a right-first order we fail. We miss the crucial information that $t_y = Int$. However, our scheme succeeds. □

Obviously, our inference scheme is still incomplete for certain examples. In the case of Example 9 we need to solve $\forall t_x.(True \supset Foo\ t_y\ t_x)$ in some intermediate step. Our solver fails unless we provide `test` with

an explicit annotation. An important consideration is whether such programs should be considered "invalid." The problem is that it would be difficult for a type inference system to distinguish between an insufficiently-annotated program and one containing a "real" type error. An important advantage of our scheme is that we can naturally incorporate the type error reporting techniques from [30, 28], allowing us to assist the user in case inference fails. Even though our implication solver is (necessarily) incomplete, we are able to pinpoint precisely which program locations give rise to constraints that lead to unsolvable implications. A simple example can be found in Appendix B.

4. Conclusion

We have introduced novel forms of lexically scoped annotations and provided evidence for their usefulness. We have studied their formal properties in detail. Our form of annotations generalize the "type-lambda" annotations in SML [19] and the "type-sharing" annotations in GHC [7]. A novelty of our system is that we may refer to scoped variables in the constraint component of annotations. The Mondrian language [18] also supports universally scoped annotations. However, we do not know whether Mondrian annotations allow for "scoped" constraints, due to a lack of a formal description. Simonet and Pottier [24] describe a system which supports (closed) annotations of the form $\exists a.\forall b.t$. As shown, we can encode such annotations by having an outer existential annotation followed by an inner universal annotation.

Type inference for annotations turned out be difficult. We have introduced a novel inference scheme as a combination of two of our own methods [30, 25]. Our scheme is a clear improvement over formulations based on algorithm $\mathcal{W}$ or its variants (currently employed by state of the art compilers for the Haskell and ML family). We can type more programs and support naturally sophisticated type error reporting. In this context, it is worth mentioning recent work by Pottier and Rémy [22]. They extend the constraint language with a "let" construct which resembles our use of type relations to provide for another alternative type inference approach. However, their main motivation is to support more efficient type scheme generalization and instantiation. That is, they do not study the impact annotations have on type inference with constraints nor do they consider type error reporting. Our extensions have been fully implemented as part of the Chameleon system [32]. We are not aware of any other system which covers such expressive annotations.

An interesting topic for future work is to consider the combination of higher-rank types and our form of lexically scoped annotations.

References

[1] A. I. Baars and S. D. Swierstra. Typing dynamic typing. In *Proc. of ICF'02*, pages 157–166. ACM Press, 2002.

[2] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proc. of Haskell Workshop'02*, pages 90–104. ACM Press, 2002.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. of POPL'82*, pages 207–212. ACM Press, January 1982.

[4] K. F. Faxén. Haskell and principal types. In *Proc. of Haskell Workshop'03*, pages 88–97. ACM Press, 2003.

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

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

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

[8] C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. In *ESOP'94*, volume 788 of *LNCS*, pages 241–256. Springer-Verlag, April 1994.

[9] R. Harper and J. C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[10] Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.

[11] F. Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.

[12] Hugs home page. haskell.cs.yale.edu/hugs/.

[13] M. P. Jones and J. Nordlander, 2000. http://haskell.org/pipermail/haskell-cafe/2000-December/001379.html.

[14] S. Peyton Jones and M. Shields. Lexically scoped type variables. http://research.microsoft.com/Users/simonpj/, 2004.

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

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

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

[18] E. Meijer and K. Claessen. The design and implementation of Mondrian. In *Haskell Workshop*, June 1997.

[19] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

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

[21] S. Peyton-Jones, 2000. http://www.haskell.org//pipermail/haskell/2000-December/006301.html.

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

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

[24] V. Simonet and F. Pottier. Constraint-based type inference with guarded algebraic data types. Submitted to *ACM Transactions on Programming Languages and Systems*, June 2004.

[25] P. J. Stuckey and M. Sulzmann. Solutions of implication constraints yield type inference for more general algebraic data types. Submitted to ICFP'05.

[26] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proc. of ICFP'02*, pages 167–178. ACM Press, 2002.

[27] P. J. Stuckey and M. Sulzmann. Type inference for guarded recursive data types, 2005. Submitted to ICALP'05.

[28] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Proc. of Haskell'04*, pages 80–91. ACM Press, 2004.

[29] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improved inference for checking type annotations. Technical Report TRA2/05, The National University of Singapore, 2005.

[30] P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Proc. of Haskell Workshop'03*, pages 72–83. ACM Press, 2003.

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

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

[33] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76. ACM Press, 1989.

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

$$\text{(Var)} \qquad \frac{(x:\sigma) \in \Gamma}{\Gamma \vdash_F x : \sigma}$$

$$\text{(T-Abs)} \qquad \frac{\Gamma \vdash_F E : t \quad \bar{a} \cap fv(\Gamma) = \emptyset}{\Gamma \vdash_F \Lambda \bar{a}.E : \forall \bar{a}.t}$$

$$\text{(T-App)} \qquad \frac{\Gamma \vdash_F E : \forall \bar{a}.t}{\Gamma \vdash_F E\,[\bar{t}] : [\bar{t}/\bar{a}]t}$$

$$\text{(Abs)} \qquad \frac{\Gamma.x : t_1 \vdash_F E : t_2}{\Gamma \vdash_F \lambda x.E : t_1 \to t_2}$$

$$\text{(App)} \qquad \frac{\Gamma \vdash_F E_1 : t_1 \to t_2 \quad \Gamma \vdash_F E_2 : t_1}{\Gamma \vdash_F E_1\, E_2 : t_2}$$

$$\text{(Let)} \qquad \frac{\Gamma.g :: \sigma \vdash_F E : \sigma \quad \Gamma.g : \sigma \vdash_F E' : t'}{\Gamma \vdash_F \mathsf{let}\ g : \sigma = E\ \mathsf{in}\ E' : t'}$$

Figure 4. System F Variant

A. Proofs

A.1 Proof of Theorem 1

We make use of a predicative version of System F. For simplicity, we adopt the type language from Section 2 (good enough for our purposes here). The System F expressions are as follow:

$$E \quad ::= \quad x \mid E\ E \mid \lambda x.E \mid E\,[\bar{t}] \mid \Lambda \bar{a}.E \mid \mathsf{let}\ g :: \sigma = E\ \mathsf{in}\ E$$

For convenience, we include let statements with explicit annotations. We use judgments $\Gamma \vdash_F E : t$ to describe well-typing of terms. The typing rules are in Figure 4. Note that the formulation of (Let) includes recursive function definitions.

Our task now is to translate each expression typable according to Figure 1 into an equivalent System F typable expression. To deal with constraints we would first need to translate constraints into expressions. More precisely, we would need to associate proof terms with constraints and turn (logical) proof statements such as $P_p \models C_1 \supset C_2$ into proof term transformation functions. We note that this is fairly standard material in case of type classes and subtyping. Hence, w.l.o.g. we focus on standard Hindley/Milner extended with lexically scoped universal and existential annotations. Hence, we abbreviate $True, V, \Gamma \vdash e : t$ by $V, \Gamma \vdash e : t$ and use "specialized" typing rules for simplicity. We define a type-erasure function on Hindley/Milner and System

F expressions as follow:

$$
\begin{array}{l}
\mathit{erasure}(x) = x \\
\mathit{erasure}(E\ [\bar{t}]) = \mathit{erasure}(E) \\
\mathit{erasure}(\lambda x.E) = \lambda x.\mathit{erasure}(E) \\
\mathit{erasure}(\lambda x.e) = \lambda x.\mathit{erasure}(e) \\
\mathit{erasure}(\Lambda \bar{a}.E) = \mathit{erasure}(E) \\
\mathit{erasure}(E_1\ E_2) = \mathit{erasure}(E_1)\ \mathit{erasure}(E_2) \\
\mathit{erasure}(e_1\ e_2) = \mathit{erasure}(e_1)\ \mathit{erasure}(e_2) \\
\mathit{erasure}(\mathsf{let}\ g :: \sigma = E\ \mathsf{in}\ E') = \mathsf{let}\ g = \mathit{erasure}(E)\ \mathsf{in}\ \mathit{erasure}(E') \\
\mathit{erasure}(\mathsf{let}\ \begin{array}{c} a(g) \\ f = e \end{array}\ \mathsf{in}\ e') = \mathsf{let}\ g = \mathit{erasure}(e)\ \mathsf{in}\ \mathit{erasure}(e')
\end{array}
$$

Our theorem is derived form the following statement.

LEMMA 2. *Let* $V, \Gamma \vdash e : t$. *Then* $\Gamma \vdash_F E : t$ *for some* E *such that* $\mathit{erasure}(e) = \mathit{erasure}(E)$.

PROOF. The proof proceeds by structural induction over e. We only show the interesting cases.

Case (Var-$\forall$E): We have that

$$\frac{(x : \forall \bar{a}.t) \in \Gamma}{V, \Gamma \vdash x : [\bar{t}/\bar{a}]t}$$

We find that $\Gamma \vdash_F x : \forall \bar{a}.t$. Application of rule (T-App) yields $\Gamma \vdash_F x\ [\bar{t}] : [\bar{t}/\bar{a}]t$. We have that $\mathit{erasure}(x) = \mathit{erasure}(x\ [\bar{t}])$. Hence, we are done.

Case (LetA): We have that

$$\frac{\begin{array}{c} V, \Gamma \vdash e_1 : t_1 \quad \bar{a} = \mathit{fv}(t_1) - \mathit{fv}(V, \Gamma) \\ V, \Gamma.g : \forall \bar{a}.t_1 \vdash e_2 : t_2 \end{array}}{V, \Gamma \vdash \mathsf{let}\ \begin{array}{c} g :: t_1 \\ g = e_1 \end{array}\ \mathsf{in}\ e_2 : t_2}$$

By induction we have that $\Gamma.g : \forall \bar{a}.t_1 \vdash_F E_1 : t_1$ (1) and $\Gamma.g : \forall \bar{a}.t_1 \vdash_F E_2 : t_2$ where $\mathit{erasure}(e_1) = \mathit{erasure}(E_1)$ (2) and $\mathit{erasure}(e_2) = \mathit{erasure}(E_2)$. From (1) and rule (T-Abs), we conclude that $\Gamma.g : \forall \bar{a}.t_1 \vdash_F \Lambda \bar{a}.E_1 : \forall \bar{a}.t_1$ (3). From (2), (3) and rule (Let), we derive that $\Gamma \vdash_F \mathsf{let}\ g :: \forall \bar{a}.t_1 = E_1\ \mathsf{in}\ E_2 : t_2$. The type-erasure of expressions in final judgments are equivalent. Hence, we are done.

Case (LetA2): We have that

$$\frac{\begin{array}{c} V, \Gamma \vdash e_1 : t_1' \quad \bar{a} = \mathit{fv}(t_1') - \mathit{fv}(V, \Gamma) \\ V, \Gamma.g : \forall \bar{a}.t_1 \vdash e_2 : t_2 \\ \models \exists \mathit{fv}(t_1).t_1 = t_1' \end{array}}{V, \Gamma \vdash \mathsf{let}\ \begin{array}{c} g ::: t_1 \\ g = e_1 \end{array}\ \mathsf{in}\ e_2 : t_2}$$

By induction we have that $\Gamma \vdash_F E_1 : t_1'$ (1) and $\Gamma.g : \forall \bar{a}.t_1' \vdash_F E_2 : t_2$ where $\mathit{erasure}(e_1) = \mathit{erasure}(E_1)$ (2) and $\mathit{erasure}(e_2) = \mathit{erasure}(E_2)$. The remaining steps are essentially the same as in case of (LetA). □

A.2 Proof of Lemma 1

Assumptions: Let

$$e_1 \equiv e\left[\begin{array}{l} f :: C_1 \Rightarrow t_1 \\ f = e'\left[\begin{array}{l} g :: C_2 \Rightarrow t_2 \\ g = e'' \end{array}\right] \end{array}\right]$$

and

$$e_2 \equiv e\left[\begin{array}{l} f :: C_1 \Rightarrow t_1 \\ f = e'\left[\begin{array}{l} g :: C_1 \wedge C_2 \Rightarrow t_2 \\ g = e'' \end{array}\right] \end{array}\right]$$

Then $C, V, \Gamma \vdash e_1 : t$ iff $C, V, \Gamma \vdash e_2 : t$.

PROOF. It is sufficient to verify that $C \wedge C_2, V, \Gamma.h : \forall \bar{a}.C_1 \Rightarrow t_1 \vdash e : t$ iff $C \wedge C_2, V, \Gamma.h : \forall \bar{a}.C_1 \wedge C_2 \Rightarrow t_1 \vdash e : t$ where $fv(C_2) \subseteq V$.

We proceed by structural induction over e. We only consider the case for rule (Var-$\forall$E).

Note that when considering proof trees associated to derivation we find that the V and Γ components in lower nodes are a subset of those in higher nodes (see rules (Abs), (Let) and (LetA)). The constraint component in lower nodes is entailed by the the one found in higher nodes (see rules (Let) and (LetA)).

Hence, assume $V \subseteq V', \Gamma \subseteq \Gamma'$ and $P_p \models C' \supset C \wedge C_2$. We find that $C', V', \Gamma'.h : \forall \bar{a}.C_1 \Rightarrow t_1 \vdash h : [\bar{t}/\bar{a}]t$ iff $C', V', \Gamma'.h : \forall \bar{a}.C_1 \wedge C_2 \Rightarrow t_1 \vdash h : [\bar{t}/\bar{a}]t$ (consider (Var-$\forall$E)). □

A.3 Proof of Theorem 3

First, we state some auxiliary lemmas and definitions.

REMARK 1. *A silent assumption is that in case of $F \models \exists a.C$ we assume there exists a $\phi = [t/a]$ such that $F \models \phi(C)$. Note that $F \models \exists a.C$ holds iff for any model M of F we have that there exists $\phi = [t/a]$ such that $M \models \phi(C)$. Hence, our interpretation of $F \models \exists a.C$ is such that we assume we have a model M of F such that $M \models [t/a]C$ for some t which is good enough for our purposes.*

In a statement $F_1 \models F_2$ we commonly leave implicit the universal quantifier over any unbound variables in F_1 and F_2.

Often, we make use of the fact that $F \models C \supset C'$ is equivalent to the statement $F, C \supset C'$. Recall that F, C is a short-hand for $F \wedge C$.

We state that under a more general environment and weaker constraint we can derive a more specific type. Note that $P_p \wedge C_2' \vdash \Gamma' \preceq \Gamma$ states component-wise subsumption among types in Γ and Γ'.

LEMMA 3 (Weakening). *Let P_p be the program theory, $C_2, V, \Gamma \vdash e : t_2$, $P_p \wedge C_2' \vdash \Gamma' \preceq \Gamma$ and $P_p \models C_2' \supset \exists \bar{a}_2.(C_2 \wedge t_2 = t_2')$ where $\bar{a}_2 \cap fv(C_2', t_2', V, \Gamma, \Gamma') = \emptyset$. Then $C_2', V, \Gamma' \vdash e : t_2'$.*

PROOF. We proceed by structural induction over e. We only show the interesting cases.

Case (Var-$\forall$E):

$$\frac{(x : \forall \bar{a}.D \Rightarrow t) \in \Gamma \quad P_p \models C_2 \supset [\bar{t}/\bar{a}]D}{C_2, V, \Gamma \vdash x : [\bar{t}/\bar{a}]t}$$

Note that we have replaced the instantiation side condition by its logical equivalent. From the premise, $P_p \models C_2 \supset \phi(D)$ (1) for some $\phi = [\bar{t}/\bar{a}]$.
By assumption $(x : \forall \bar{a'}.D' \Rightarrow t') \in \Gamma'$ (2),
$P_p \wedge C_2' \wedge D \models \phi'(D' \wedge t' = t)$ (3) for some $\phi' = [\bar{t'}/\bar{a'}]$, and
$P_p \wedge C_2' \models \phi_2(C \wedge \phi(t) = t_2')$ (4) for some $\phi_2 = [\bar{t_2}/\bar{a_2}]$.

From (1) we derive that $P_p \models \phi_2 C_2 \supset \phi_2 \circ \phi D$ (5). From (4) we derive that $P_p, C_2' \models \phi_2 C_2$ (6). By construction, ϕ does not any variables other than $fv(D, t)$. Hence, from (3), (5) and (6) we conclude that

$P_p, C_2' \models \phi' D', \phi'(t') = \phi_2 \circ \phi(t)$ (7). From (4) we derive that $P_p, C_2' \models \phi_2 \circ \phi(t) = t_2'$ (8). Hence, from (7) and (8), we conclude that $P_p, C_2' \vdash (\forall \bar{a}'.D' \Rightarrow t') \preceq t_2'$. Hence, application of rule (Var) yields $C_2', V, \Gamma' \vdash x : t_2'$ and we are done.

Case (LetA):

$$\frac{\begin{array}{c}\bar{a} = fv(C_1, t_1) \backslash fv(C_2, V, \Gamma) \\ C_2 \wedge C_1, V.\bar{a}, \Gamma.(f : \forall \bar{a}.C_1 \Rightarrow t_1) \vdash e_1 : t_1 \\ C_2, V, \Gamma.(f : \forall \bar{a}.C_1 \Rightarrow t_1) \vdash e_2 : t_2\end{array}}{C_2, V, \Gamma \vdash \mathsf{let} \begin{array}{l} f :: C_1 \Rightarrow t_1 \\ f = e_1 \end{array} \mathsf{in}\ e_2 : t_2}$$

Application of the induction hypothesis to the left and right premise yields. $C_2' \wedge C_1, V.\bar{a}, \Gamma'.(f : \forall \bar{a}.C_1 \Rightarrow t_1) \vdash e_1 : t_1$ (we use $C_2'' = C_2' \wedge C_1$ and $t_2'' = t_1$) and $C_2', V, \Gamma.(f : \forall \bar{a}.C_1 \Rightarrow t_1) \vdash e_2 : t_2'$. Note that w.l.o.g. we assume that there are no name clashes. Then, rule (LetA) yields $C_2', V, \Gamma \vdash \mathsf{let}\ f :: C_1 \Rightarrow t_1; f = e_1\ \mathsf{in}\ e_2 : t_2'$.

Note that we do not necessarily know that C_2'' is satisfiable given that $C_2 \wedge C_1$ is satisfiable.

The other cases can be proven similarly. □

We slightly generalize the relation $\sim$ (introduced in Section 3.2) among CHRs P and environments E, Γ and Γ_λ by taking into account the set V of scoped type variables. We assume that $Termfv(\{x_1, \sigma_1, ..., x_n : \sigma_n\}) = \{x_1, ..., x_n\}$.

We define $P_E, E \overset{V}{\sim} \Gamma, \Gamma_\lambda$ iff

1. Γ_λ only consists of simple types, $Termfv(\Gamma) = E, fv(\Gamma) \subseteq fv(\Gamma_\lambda, V)$,
2. $P_E = \{f(t, l, v) \Longleftrightarrow C \mid f \in E\}$,
3. $\Gamma = \{f : \forall \bar{a}.C' \Rightarrow t \mid f \in E, fv(\Gamma_\lambda) = \bar{t}, V = \bar{b}, f(t, l, v), l = \langle \bar{t} \rangle, v = \langle \bar{b} \rangle \longrightarrow^*_{P_E} C', \bar{a} = fv(C', t) - fv(\Gamma_\lambda, V)\}$.

Note that in the third condition, we can further simplify constraints such as $l = \langle \bar{t} \rangle$ in C' by building the m.g.u. The important point is that the type represented by the CHR and recorded in the environment are (logically but not necessarily syntactically) equivalent.

We can derive Theorem 3 from the following lemma.

LEMMA 4. *Let P_p be the program theory. Let $P_E, E \overset{V}{\sim} \Gamma, \Gamma_\lambda$, $V, E, \Gamma_\lambda, e \vdash_C (C \mid t)$ and $V, E, \Gamma_\lambda, e \vdash_R P$ such that $C \longrightarrow^*_{P, P_E, P_p} D$ and e is let-realizable. Then, $D, V, \Gamma \cup \Gamma_\lambda \vdash e : t$.*

PROOF. The proof proceeds by structural induction over e. We only show the most interesting case of (universal) type annotations. Part of our assumptions are stated in Figure 5.

Let $\sigma = \forall fv(C_1', t_1') - fv(\Gamma_\lambda, V).C_1' \Rightarrow t_1'$. We find that $P_E \cup \{f_a(t, l, v) \Longleftrightarrow v = \langle \bar{a} | vr \rangle, t = t_1', C_1'\}, E \cup \{f_a\} \overset{V}{\sim} \Gamma.f : \sigma, \Gamma_\lambda$. We also know that $C'' \longrightarrow^*_{P_E, P, P_p} D$ (1). Hence, application of the induction hypothesis to e_2 yields $D, V, \Gamma.f : \sigma \cup \Gamma_\lambda \vdash e_2 : t'$ (2).

By assumption all let-defined expressions are realizable. Hence, $C_1 \longrightarrow^*_{P_E, P, P_p} D_1$ (3) for some constraint D_1. Application of the induction hypothesis to e_1 yields $D_1, V, \Gamma.f : \sigma \cup \Gamma_\lambda \vdash e_1 : t_1$ (4).

From (1) and soundness of implication CHR solving we conclude that $P_E, P, P_p \models D \supset \forall \bar{b}.(C_1', t = t_1', l = \langle \bar{t} \rangle, v = \langle \bar{a}, \bar{b} \rangle \supset f(t, l, v))$ (5). We slightly misuse notation and assume that depending on the context CHRs may also refer to their logical interpretation. In (5) we unify t, l and v and obtain $P_E, P, P_p \models D \supset \forall \bar{b}.(C_1' \supset f(t_1', \langle \bar{t} \rangle, \langle \bar{a}, \bar{b} \rangle))$ (6).

From the logical interpretation of CHRs, we derive that $P_E, P, P_p \models f(t_1', \langle \bar{t} \rangle, \langle \bar{a}, \bar{b} \rangle) \leftrightarrow \bar{\exists}_{fv(\Gamma_\lambda, V.\bar{b}, t_1')}.(l = \langle \bar{t} \mid tr \rangle, v = \langle \bar{a}, \bar{b} \mid vr \rangle, t_1' = t_1, C_1)$. We use $\bar{\exists}_W.C$ as a short-hand for $\exists fv(C) - W.C$. From (3) we

$$\frac{\begin{array}{c} V, E.f_a, \Gamma_\lambda, e_2 \vdash_C (C' \mid t') \\ \Gamma_\lambda = \overline{x:t} \quad V = \bar{a} \quad \bar{b} = fv(C'_1, t'_1) - V \quad t,l,v \text{ fresh} \\ C'' = \{C', \forall \bar{b}.(C'_1, t = t'_1, l = \langle \bar{t} \rangle, v = \langle \bar{a}, \bar{b} \rangle \supset f(t,l,v))\} \end{array}}{V, E, \Gamma_\lambda, \mathsf{let} \begin{array}{l} f :: C'_1 \Rightarrow t'_1 \\ f = e_1 \end{array} \mathsf{in}\ e_2 \vdash_C (C'' \mid t')}$$

$$\frac{\begin{array}{c} \Gamma_\lambda = \overline{x:t} \quad V = \bar{a} \quad \bar{b} = fv(C'_1, t'_1) - V \\ V.\bar{b}, E.f_a, \Gamma_\lambda, e_1 \vdash_R P_1 \quad V, E.f_a, \Gamma_\lambda, e_2 \vdash_R P_2 \\ V.\bar{b}, E.f_a, \Gamma_\lambda, e_1 \vdash_C (C_1 \mid t_1) \quad t,l,v,lr,vr \text{ fresh} \\ P = P_1 \cup P_2 \cup \\ \left\{ \begin{array}{lcl} f_a(t,l,v) & \iff & v = \langle \bar{a} \mid vr \rangle, t = t'_1, C'_1 \\ f(t,l,v) & \iff & l = \langle \bar{t} \mid lr \rangle, v = \langle \bar{a}, \bar{b} \mid vr \rangle, t = t_1, C_1 \end{array} \right\} \end{array}}{V, E, \Gamma_\lambda, \mathsf{let} \begin{array}{l} f :: C'_1 \Rightarrow t'_1 \\ f = e_1 \end{array} \mathsf{in}\ e_2 \vdash_R P}$$

Figure 5. Case (LetA)

can conclude that $P_E, P, P_p \models C_1 \leftrightarrow \bar{\exists}_{fv(C_1)}.D_1$. Hence, we find that $P_E, P, P_p \models f(t'_1, \langle \bar{t} \rangle, \langle \bar{a}, \bar{b} \rangle) \supset \bar{\exists}_{fv(\Gamma_\lambda, V.\bar{b}, t'_1)}.(t'_1 = t_1, D_1)$ (7).

From (6) and (7), we conclude that $P_E, P, P_p \models D \supset \forall \bar{b}.(C'_1 \supset \bar{\exists}_{fv(\Gamma_\lambda, V.\bar{b}, t'_1)}.(t'_1 = t_1, D_1))$ which is equivalent to $P \models D \wedge C'_1 \supset \bar{\exists}_{fv(\Gamma_\lambda, V.\bar{b}, t'_1)}.(t'_1 = t_1, D_1)$ (8). Note that as usual we drop the implicit universal quantifiers. Constraints in D, C'_1 and D_1 are "fully" solved, hence, only depend on the constraint relations described in P_p.

From (4) and (8) by application of weakening we obtain that $D \wedge C'_1, V, \Gamma.f : \sigma \cup \Gamma_\lambda \vdash e_1 : t'_1$ (9). It remains to apply rule (LetA) to (2) and (9) which yields $D, V, \Gamma \cup \Gamma_\lambda \vdash \mathsf{let} \begin{array}{l} f :: C'_1 \Rightarrow t'_1 \\ f = e_1 \end{array} \mathsf{in}\ e_2 : t'$ and we are done. □

B. Type Error Reporting

In our implementation, we annotate all constraints with information about the location(s) they arose from. In the case of Example 9 we can report to the programmer the precise set of locations which gave rise to the *Foo* constraint that makes the implication unsolvable (in our system.) The system could report:

```
ex9.hs:5:ERROR: Inferred type does not
                subsume declared type
Declared:forall a. a -> Int
Inferred:forall a. Foo b a => a -> Int
Problem :Constraint Foo b a, from following
         location, is unmatched.
         f :: c -> Int
         f x = foo y x
```

The system can be easily extended to provide additional hints to the programmer. A simple heuristic might involve finding all instances that could be used to eliminate the problematic constraint and suggesting to the programmer to add enough additional type information to have one of them apply.

C. Lexically Scoped Instance Declarations

We allow that the scope of an instance declaration comprises of the entire body of that declaration. This is similar to the way we treat type annotations.

Consider a class declaration, with a single method:

`class` $TC\ \bar{\alpha}$ `where` `m` :: $C \Rightarrow t$

and an instance of this class:

`instance` $C'' \Rightarrow TC\ \bar{t}$ `where` `m` $= e$

Implicitly, `m`'s inferred type must subsume, $[\bar{t}/\bar{a}](C \Rightarrow t)$, i.e. the type specified in the class declaration, with the instance types substituted in place of the type class variables.

As usual, we generate constraints out of the expression e, by employing judgement $V, E, \Gamma, e \vdash_C (C', t')$, where, in this case we set $V = fv(\bar{t}, C'')$ to allow the instance types to scope over all methods.

The following constraint represents the requirement that the method's inferred type subsumes its implicit declared type.

$\forall_{fv(C,t,\bar{a},V)}.(\bar{a} = \bar{t}, C'', C, t = t' \supset C')$