JFP **18** (2): 251–283, 2008. © 2007 Cambridge University Press 251
doi:10.1017/S0956796807006569 First published online 13 September 2007 Printed in the United Kingdom

HM(X) type inference is CLP(X) solving

MARTIN SULZMANN
School of Computing, National University of Singapore, S16 Level 5, 3 Science Drive 2, Singapore 117543, Singapore
(*e-mail:* sulzmann@comp.nus.edu.sg)

PETER J. STUCKEY
NICTA Victoria Laboratory, Department of Computer Science and Software Engineering, University of Melbourne, Victoria 3010, Australia
(*e-mail:* pjs@cs.mu.oz.au)

Abstract

The HM(X) system is a generalization of the Hindley/Milner system parameterized in the constraint domain X. Type inference is performed by generating constraints out of the program text, which are then solved by the domain-specific constraint solver X. The solver has to be invoked at the latest when type inference reaches a let node so that we can build a polymorphic type. A typical example of such an inference approach is Milner's algorithm $\mathscr{W}$. We formalize an inference approach where the HM(X) type inference problem is first mapped to a CLP(X) program. The actual type inference is achieved by executing the CLP(X) program. Such an inference approach supports the uniform construction of type inference algorithms and has important practical consequences when it comes to reporting type errors. The CLP(X) style inference system, where X is defined by Constraint Handling Rules, is implemented as part of the Chameleon system.

1 Introduction

The Hindley/Milner system is one of the most widely used type systems for programming language design and program analysis. Type inference is an important feature and relieves the user from providing an excessive amount of type information. The standard approach toward type inference is to traverse the abstract syntax tree and generate constraints out of the program text. These constraints need to be solved at the latest when inference reaches a let node in order that we can build a type scheme. Type schemes are also known as parametric polymorphic types (or polymorphic types for short). Typical examples of such an inference approach are Milner's (1978) algorithm $\mathscr{W}$ or variants such as algorithms $\mathscr{M}$ (Lee & Yi 1998) and $\mathscr{G}$ (Eo *et al.* 2003). The choice of the specific algorithm only affects the order of traversal of the abstract syntax tree. The main structure of the inference algorithm remains the same. That is, inference employs a combination of interleaved constraint generation and constraint solving to compute the final result type.

In this article, we formalize an inference approach where the entire Hindley/Milner type inference problem is mapped to a logic program. Thus, we can explain Hindley/Milner inference as a two-stage process where we first generate a logic

program and then in a subsequent phase we run the logic program to compute the actual inference result. The crucial difference to the standard approach is that there is no interleaving between constraint generation and solving. Both phases are now clearly separated.

Our main result is much more general. We show that the HM(X) type inference problem can be explained as constraint logic programming over domain X. The constraint logic programming scheme (Jaffar & Lassez 1987; Jaffar *et al.* 1998) defines a family of languages, CLP(X), for constraint domains X, generalizing logic programming from the fixed Herbrand constraint domain to arbitrary domains. Similarly, the HM(X) (Sulzmann *et al.* 1997; Odersky *et al.* 1999; Sulzmann 2000) type system generalizes the Hindley/Milner system by generalizing the domain of type constraints beyond Herbrand constraints. Concrete type instances are obtained by instantiating the abstract constraint domain X. For example, in case of Hindley/Milner, the constraint domain X is set to be the Herbrand constraint domain for which solving is achieved via unification (Robinson 1965). There are plenty of further examples of constraint domains X and their respective solvers in the literature such as record constraints (Rémy 1993) and subtype constraints (Pottier 1998). In our own work (Stuckey & Sulzmann 2005), we show how to describe the type class constraint domain (Wadler & Blott 1989) via Constraint Handling Rules (CHRs) (Frühwirth 1995). We can take advantage of these works and provide CLP(X)-based type inference for record, subtype and type class systems by instantiating X with the domain-specific solver.

The results reported in this article are based on previous work (Sulzmann *et al.* 1999; Sulzmann 2000; Stuckey *et al.* 2003b). The idea of mapping Hindley/Milner type checking and inference to logic programming is well known, at least in the logic programming community (e.g., consider Mycroft & O'Keefe 1984; Lakshman & Reddy 1991; Demoen *et al.* 1999). However, we provide the first formal treatment on the subject including concise soundness and completeness results of type inference.

In summary, our contributions are:

- We give an algorithm $\mathcal{W}$-style constraint-based reformulation of HM(X) type inference that is parameterized in terms of the domain-specific solver for the constraint domain X. The proofs of soundness and completeness of type inference are more "light-weight" than previous substitution-based formulations (Section 3).
- We show that the entire HM(X) type inference problem can be phrased in terms of CLP(X) solving (Section 4). An important advantage of the CLP(X)-based type inference scheme over algorithm $\mathcal{W}$ is an order-independent traversal of the abstract syntax tree (AST). This provides the basis to support better type error diagnosis methods.

We have implemented the CLP(X)-style type inference scheme as part of the Chameleon system (Sulzmann & Wazny 2007), where the constraint domain X can be described by CHRs. Of course, any other system that supports CLP(X) can be used as well. But the Chameleon system supports a number of other features

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

such as tracking of source locations connected to constraints for type error reporting purposes. We refer to Stuckey *et al.* (2006) for an overview.

Next, we highlight the key ideas of our approach. In Sections 2.1 and 2.2, we review the basics behind the CLP(X) and HM(X) systems. Related work is discussed in Section 5. We conclude in Section 6.

1.1 Highlights of CLP(X)-style type inference scheme

In a first step, we translate the type inference problem into a CLP(X) program, that is, set of Horn clauses or *rules*. We use constraints to describe the types of expressions and each rule describes the type of a function. For simplicity, we only consider the Herbrand constraint domain here, which is sufficient to describe constraints arising out of standard Hindley/Milner programs. We perform type inference by running the logic program resulting from the Hindley/Milner program.

Example 1 Consider the following program:

```
g y = let f x = x in (f True, f y)
```

We assume that the type domain supports tuples.

We introduce predicates (also referred to as constraints) $g(t)$ and $f(t)$ to constrain t to the types of functions `g` and `f`, respectively. It is necessary for us to provide a meaning for these constraints, which we will do in terms of rules. The body of each rule will contain all constraints arising from the definition of the corresponding function, which represent that function's type.

For the program above, we may generate rules similar to the following.

$$\begin{array}{lll} g(t) & :- & t = t_y \to (t_1, t_2) \wedge f(t_{f1}) \wedge t_{f1} = \mathit{Bool} \to t_1 \wedge f(t_{f2}) \wedge t_{f2} = t_y \to t_2 \\ f(t) & :- & t = t_x \to t_x \end{array}$$

We adopt the convention that the rule starting with predicate $g(t)$ (also known as rule head) is referred to as the g rule. We assume that let-bound function names are renamed to guarantee that the rule heads contain distinct predicates.

In the g rule, we see that `g`'s type is of the form $t_y \to (t_1, t_2)$, where t_1 and t_2 are the results of applying function `f` to a *Bool* and a t_y value. We represent `f`'s type, at both call sites in the program, by the predicate calls $f(t_{f1})$ and $f(t_{f2})$.

The f rule is much more straightforward. It simply states that t is `f`'s type if t is the function type $t_x \to t_x$, for some t_x, which is clear from the definition of `f`.

We can infer g's type by running the above logic program on the goal $g(t)$. We write $\rightarrowtail_{cl}$ to denote SLD resolution with respect to rule cl.

$$\begin{array}{lll} g(t) & \rightarrowtail_g & t = t_y \to (t_1, t_2) \wedge f(t_{f1}) \wedge t_{f1} = \mathit{Bool} \to t_1 \wedge f(t_{f2}) \wedge t_{f2} = t_y \to t_2 \\ & \rightarrowtail_f & t = t_y \to (t_1, t_2) \wedge t_{f1} = t_x \to t_x \wedge t_{f1} = \mathit{Bool} \to t_1 \wedge f(t_{f2}) \\ & & \wedge\ t_{f2} = t_y \to t_2 \\ & \rightarrowtail_f & t = t_y \to (t_1, t_2) \wedge t_{f1} = t_x \to t_x \wedge t_{f1} = \mathit{Bool} \to t_1 \\ & & \wedge\ t_{f2} = t'_x \to t'_x \wedge t_{f2} = t_y \to f_2 \end{array}$$

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

Before applying a rule, we rename variables to avoid name clashes. For example, see the last derivation step where in `f`'s rule we rename t_x to t'_x. Each variable that occurs only in the rule body is existentially quantified. Hence, we perform inference by exhaustively applying rules until we reach the final constraint. Any solution to the final constraint assigns a valid type for g to the variable t. We can capture `g`'s type succinctly by building the most general unifier of the constraints and applying it to variable t. We see that `g`'s type is $\forall t_y.t_y \rightarrow (Bool, t_y)$.

Let us compare our inference strategy against a traditional approach such as algorithm $\mathscr{W}$. In algorithm $\mathscr{W}$, to infer the type of g, we first infer the type of the let function `f`. Inference for let function `f` proceeds by inferring the type $t_x \rightarrow t_x$. Variable t_x is a free variable, that is, only occurs in the type of `f` and has no reference to any of the types of variables from the enclosing scope. Hence, we can universally quantify over t_x and assign `f` the type $\forall t_x.t_x \rightarrow t_x$. Under this type assignment, we continue to perform inference of `(f True, f y)`. At each call site of `f`, we build a generic instance by removing the quantifier and renaming the quantified variables with some fresh variables. The resulting constraints generated for `(f True, f y)` are effectively the same as in the last step of the CLP(X)-style inference system. As expected, algorithm $\mathscr{W}$ computes the same type $\forall t_y.t_y \rightarrow (Bool, t_y)$ for `g`.

The point is that in the CLP(X)-based inference scheme, we do not explicitly generate type schemes for let-defined functions such as `f`. Rather, we use rules to represent the set of types that can be given to `f`. Hence, there is no need to build a generic instance of `f`'s type scheme at a call site. Instead, we simply use the predicate call $f(t)$ to query the let-defined functions type.

In essence, we achieve polymorphism by replicating the constraints for let definitions. An idea that appears several times in the literature. For example, consider Henglein (1992) and Mitchell (2002). In an efficient implementation, we can use memoization and constraint simplification to reduce repeated work.

Because quantification over universal variables is implicit in the CLP(X)-based inference scheme, we need to refine our inference scheme to ensure that all references to free type variables from the environment share the same monomorphic type. Here is an example that explains this point in more detail.

Example 2 The program below is a slightly modified version of the program presented in Example 1.

```
g y = let f x = (y,x) in (f True, f y)
```

The key difference is that `f` now contains a free variable `y`. Since `y` is monomorphic within the scope of `g`, we must ensure that all uses of `y`, in all definitions, are consistent. That is, each rule that makes mention of t_y, `y`'s type, must be referring to the same variable. This is important since the scope of variables used in a rule is limited to that rule alone.

To enforce this, we perform a transformation akin to λ-lifting (also known as closure conversion) but at the type level. Instead of unary predicates of form $f(t)$, we now use binary predicates $f(t, l)$, where the l parameter represents `f`'s environment.

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

For the above program, we generate the following rules:

$$\begin{array}{lll} g(t,l) & \texttt{:-} & t = t_y \to (t_1,t_2) \wedge f(t_{f1},[t_y]) \wedge t_{f1} = \mathit{Bool} \to t_1 \wedge f(t_{f2},[t_y]) \\ & & \wedge\ t_{f2} = t_y \to t_2 \\ f(t,l) & \texttt{:-} & t = t_x \to (t_y,t_x) \wedge l = [t_y] \end{array}$$

We write $[t_1,...,t_n]$ to indicate a type-level list containing n types. Type-level lists are built using the common constructors $\cdot : \cdot$ (cons) and [] (empty list). Hence, $[t_1,...,t_n]$ is, in fact, a shorthand for $t_1 : ... : t_n : []$. The first argument in $f(t,l)$, which we commonly refer to as the t component, will be bound to the function's type. The second component, which we call l, represents a list of unbound, that is, free, variables in scope of that function. Thus, we ensure that whenever the f predicate is invoked from the g rule that t_y, the type of y, is made available to it. So, in essence, the t_y that we use in the f rule will have the same type as t_y in g, rather than simply being a fresh variable known only in g.

Type inference for g proceeds by running the above logic program on the goal $g(t,[])$, where [] represents the empty (type) environment.

$$\begin{array}{lll} g(t,[]) & \rightarrowtail_g & t = t_y \to (t_1,t_2) \wedge f(t_{f1},[t_y]) \wedge t_{f1} = \mathit{Bool} \to t_1 \\ & & \wedge\ f(t_{f2},[t_y]) \wedge t_{f2} = t_y \to t_2 \\ & \rightarrowtail_f & t = t_y \to (t_1,t_2) \wedge t_{f1} = t'_x \to (t'_y,t'_x) \wedge [t_y] = [t'_y] \wedge t_{f1} = \mathit{Bool} \to t_1 \\ & & \wedge\ f(t_{f2},[t_y]) \wedge t_{f2} = t_y \to t_2 \\ & \rightarrowtail_f & t = t_y \to (t_1,t_2) \wedge t_{f1} = t'_x \to (t'_y,t'_x) \wedge [t_y] = [t'_y] \wedge t_{f1} = \mathit{Bool} \to t_1 \\ & & \wedge\ t_{f2} = t''_x \to (t''_y,t''_x) \wedge [t_y] = [t''_y] \wedge t_{f2} = t_y \to t_2 \end{array}$$

We build the most general unifier of the resulting constraints and find that g's type is $\forall t_y.t_y \to ((t_y,\mathit{Bool}),(t_y,t_y))$. Without the l component, we would infer the incorrect type $\forall t_y.\forall t'_y.\forall t''_y.t_y \to ((t'_y,\mathit{Bool}),(t''_y,t_y))$.

Similar ideas using a list of the types of λ-bound variables for inference have been previously described in Henglein (1993) and Birkedal and Tofte (2001). To the best of our knowledge, we are the first to exploit this method in the context of HM(X).

Monomorphic recursion is straightforwardly handled by the approach by equating the type of the recursive call with the type of the function.

Example 3 Consider the simple recursive code

```
f x = (let g y = g x in g x)
```

that is written in our internal syntax as follows

```
f x = (let g y = rec g in λ y. g x in g x)
```

The generated rules are

$$\begin{array}{lll} g(t,l) & \texttt{:-} & t = t_y \to t_1 \wedge l = [t_x] \wedge t_g = t_x \to t_1 \wedge \underline{t_g = t} \\ f(t,l) & \texttt{:-} & t = t_x \to t' \wedge l = [] \wedge g(t_x \to t',[t_x]) \end{array}$$

The underlined constraint ensures that the recursive call to g has the correct type.

Polymorphic recursion is also handleable by the approach, assuming that polymorphic recursive functions have a declared type. We simply generate a rule for

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

the polymorphic recursive function using its declared type. In this case, we need to check that the constraints defining the polymorphic recursive function are implied by the declared type. But we do not consider the issues of checking type declarations further in this article.

Example 4 Consider the polymorphic recursive code

```
f :: ∀a. a -> (a,Bool)
f x = (fst (f x), snd (f True))
```

where `fst :: ∀a,b.(a,b) -> a` and `snd :: ∀a,b.(a,b) -> b` have the usual meaning. The generated rule for f is simply

$$f(t,l) \quad :- \quad t = a \to (a, Bool) \wedge l = []$$

The body of the `f` is translated into constraint C of the form

$$t = t_x \to (t_1, t_2) \wedge l = [] \wedge fst(t_{fst}, []) \wedge t_{fst} = t_3 \to t_1 \wedge f(t_4, []) \wedge t_4 = t_x \to t_3 \\ \wedge\ snd(t_{snd}, []) \wedge t_{snd} = t_5 \to t_2 \wedge f(t_6, []) \wedge t_6 = Bool \to t_5$$

Checking the declared type amounts to determining that $\exists a.t = a \to (a, Bool) \models_X \bar{\exists}_{\{t\}} C$, which is indeed the case. Notation $\bar{\exists}_{\{t\}} C$ denotes that we existentially quantify over all free variables in C but t.

So far, we assumed that X is equivalent to the Herbrand constraint domain. Thus, we can support type inference for standard Hindley/Milner. In our next example, we consider type inference for type classes by describing the constraint domain X with CHRs (Frühwirth 1995).

Example 5 We consider a Haskell-style language with support for type classes.

```
class Foo a b where foo :: a -> b -> Int
instance Foo a b => Foo [a] [b]
f xs y = foo xs (y:xs)
```

The class declaration introduces a two-parameter type class `Foo`, which comes with a method `foo` that has the constrained type $\forall a, b.Foo\ a\ b \Rightarrow a \to b \to Int$. The constraint $Foo\ a\ b$ is defined by the constraint domain X, which, in turn, is defined by the above instance. The instance declaration states that $Foo\ [a]\ [b]$ holds if $Foo\ a\ b$ holds. For simplicity, we ignore the instance body, which does not matter here. Following our previous work (Stuckey & Sulzmann 2005), we can represent such type class relations via CHRs. Here is the translation of the above program to CLP(X), where X is defined by a CHR program. We simplify the presentation by removing the l component, which is unnecessary here.

$$\begin{array}{rcl} Foo\ [a]\ [b] & \Longleftrightarrow & Foo\ a\ b \\ foo(t) & :- & t = a \to b \to Int \wedge Foo\ a\ b \\ f(t) & :- & t = t_{xs} \to t_y \to t_1 \wedge t_{xs} = [a] \wedge t_y = a \wedge foo(t_{xs} \to t_y \to t_1) \end{array}$$

We adopt the convention that predicates starting with lowercase letters refer to the types of functions, that is, such predicates are defined by CLP(X) rules, and predicates

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

starting with uppercase letters refer to constraints defined by the constraint domain X (which is defined via CHRs here). Above the rule for function `f` and method `foo`, we find a CHR that represents the instance declaration. CHRs define rewrite rules among constraints. The above rule states to rewrite $Foo\ [a]\ [b]$ (or an instance of it) to $Foo\ a\ b$. These CHR solving steps are simply performed during the CLP(X) solving process.

Here is the inference derivation for function `f`.

$$\begin{array}{rll} f(t) & \rightarrowtail_f & t = t_{xs} \rightarrow t_y \rightarrow t_1 \wedge t_{xs} = [a] \wedge t_y = a \wedge foo(t_{xs} \rightarrow t_{xs} \rightarrow t_1) \\ & \rightarrowtail_{foo} & t = t_{xs} \rightarrow t_y \rightarrow t_1 \wedge t_{xs} = [a] \wedge t_y = a \wedge Foo\ a'\ b' \\ & & \wedge\ t_{xs} \rightarrow t_{xs} \rightarrow t_1 = a' \rightarrow b' \rightarrow Int \\ & \leftrightarrow_X & t = [a] \rightarrow a \rightarrow Int \wedge t_{xs} = [a] \wedge t_y = a \wedge Foo\ [a]\ [a] \\ & & \wedge\ a' = [a] \wedge b' = [a] \wedge t_1 = Int \\ & \leftrightarrow_X & t = [a] \rightarrow a \rightarrow Int \wedge t_{xs} = [a] \wedge t_y = a \wedge Foo\ a\ a \\ & & \wedge\ a' = [a] \wedge b' = [a] \wedge t_1 = Int \end{array}$$

In the last two derivation steps, we simplify constraints giving equivalent constraints with respect to the constraint domain X, by first building the most general unifier, and then applying a constraint handling rule. We find that `f` has type $\forall a.Foo\ a\ a \Rightarrow [a] \rightarrow a \rightarrow Int$.

In summary, we can support type inference for a wide range of systems by plugging in the domain-specific solver for X into the generic CLP(X) solving engine. Furthermore, in the CLP(X)-based inference scheme, we can maintain a strict phase distinction between constraint generation and solving. We first generate the CLP(X) program and then we run the CLP(X) program on some appropriate goal, for example, the constraints corresponding to the top-most expression, to obtain the inference result. In a traditional inference scheme such as algorithm $\mathscr{W}$, we find a mix of constraint generation and solving because each let statement invokes the solver to infer the type of the let-defined function. Only then, we can proceed to generate the constraints out of the let body.

The formal details of phrasing HM(X) type inference in terms of CLP(X) solving are given in Section 4. The main benefit of the CLP(X)-based type inference scheme is an order-independent traversal of the AST.

A separate constraint viewpoint allows us to improve type error diagnosis significantly. For details, see Stuckey *et al.* (2003a, 2003b, 2004, 2006); here, we only give a brief overview. The separate constraint viewpoint avoids the traversal bias of algorithms such as algorithm $\mathscr{W}$, and can explain the real nature of a type error that is caused by a set of conflicting locations. We can expose multiple reasons for a type error, and explain the reasons for an expression having a particular type.

Consider the following program, where `toLower :: Char->Char` and `toUpper :: Char->Char`. There are two minimal unsatisfiable sets of constraints in the generated constraint describing the type of k. The unsatisfiable sets of constraints arise from the two highlighted sets of locations:

```
k x = if x then (toUpper x) else (toLower x)
k x = if x then (toUpper x) else (toLower x)
```

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

A change at the shared location may fix both errors, so by Occam's razor is more likely to be the source of the problem. We could report the error as:

```
Problem :Test expression in if must be Bool
Types :  Char (test argument)
         Bool (test)
Conflict:k x = if x then (toUpper x) else (toLower x)
```

Although our earlier work (Stuckey *et al.* 2003b, 2004) used a particular form of constraint domain X, the same methods extend to arbitrary domains. We only need a constraint solver to determine minimal unsatisfiable constraints and a constraint simplifier to display types as succinctly as possible. For determining smaller sets of location that cause a given type, we need an implication tester that determines if $C \supset D$ for domain X. Hence, the approach extends to any constraint domain X.

Finally, we remark that type inference using CLP(X) systems can be very efficient. A CLP(X) system is specialized for SLD resolution and constraint solving, and hence is very efficient. If the CLP(X) system supports tabling, it can also memorize earlier answers to avoid repeated computation and use early projection (Fordan & Yap 1998) for simplifying intermediate answers, although the implementation of Demoen *et al.* (1999) found it was unnecessary even for substantial programs. Thus, this approach not only provides a clean theoretical understanding of type inference, which supports more complicated error reasoning, but also leads to practical efficient type inference.

In summary, the advantages of the CLP(X) approach are: (a) better understanding of type inference by the separation of concerns, (b) flexible and accurate type error diagnosis, and (c) efficient implementation of type inference.

2 Background

2.1 The CLP(X) framework

We assume familiarity with the basics of first-order logic. We use common notation for Boolean conjunction ($\wedge$), implication ($\supset$), equivalence ($\leftrightarrow$), and universal ($\forall$) and existential quantifiers ($\exists$). We let $\exists_V.F$ denote the logical formula $\exists a_1 \cdots \exists a_n.F$, where $V = \{a_1, \ldots, a_n\}$, and let $\bar{\exists}_V.F$ denote $\exists_{fv(F)-V}$, where fv returns the set of free variable in its argument. We let $\tilde{\exists}.F$ denote the existential closure of F, and $\tilde{\forall}.F$ the universal closure.

We use $\bar{s}$ to represent a sequence of objects $s_1, \ldots, s_n$. A substitution $[t_1/a_1, \ldots, t_n/a_n]$, also written $[\bar{t}/\bar{a}]$, simultaneously replaces each variable a_i by term t_i.

The CLP(X) scheme defines a class of languages, parametric in the choice of *constraint domain* X. A constraint domain defines the meaning of terms and constraints. We give a simplified definition of the CLP(X) scheme that suffices for our purposes.

For our purposes, a constraint domain X consists of a *signature* Σ_X, which defines the function and predicates symbols and their arities, and a *constraint theory* $\mathscr{T}_X$, which is the set of true formulae over Σ_X. We use the notation $F \models_X F'$ to mean $\mathscr{T}_X \wedge F \models F'$, that is, all models of $\mathscr{T}_X$ and F also model F'.

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

The language of terms and constraints in CLP(X) is:

$$\begin{array}{llll} \text{Terms} & t & ::= & a \mid T\ \bar{t} \\ \text{Constraints} & C, D & ::= & True \mid U\ \bar{t} \mid C \wedge C \mid \exists a.C \end{array}$$

where a is a variable and T is a function symbol in Σ_X and U is a predicate symbol in Σ_X. We will often write $\exists \bar{a}.C$ as a short-hand for $\exists a_1 \cdots \exists a_n.C$, similarly, $\forall \bar{a}.C$

We assume the signature includes (right associative) binary function symbol $\cdot \to \cdot$ written infix, constant [] representing the empty list, and (right associative) binary function symbol $\cdot : \cdot$, written infix, representing *cons*. We assume the signature includes binary predicate symbol $\cdot = \cdot$, written infix. We assume that the theory $\mathscr{T}_X$ ensures $=$ is an equality relation on terms in Σ_X, and $:$ is a Herbrand constructor, that is, $\forall t_1.\forall t_2.\forall t_3.\forall t_4.t_1 : t_2 = t_3 : t_4 \supset t_1 = t_3 \wedge t_2 = t_4$ We assume *True* is an always satisfiable constraint, that is, an identity for $\wedge$.

For example, for (pure) Hindley/Milner type inference, the constraint domain is a Herbrand domain H. For example, $\Sigma_H = (\{Int, Bool, \cdot \to \cdot, [\cdot], [], \cdot : \cdot, \}, \{\cdot = \cdot\})$, $\mathscr{T}_H$ is the complete axiomatization of (finite tree) Herbrand domains (Maher 1988).

A CLP(X) rule defines the meaning of new predicate symbols in terms of domain X. Let Π be a set of predicate symbols disjoint from those in Σ_X. The language of CLP(X) rules is defined as

$$\begin{array}{llll} \text{Head} & H & ::= & p(a_1, \ldots, a_n) \\ \text{Atom} & L & ::= & p(t_1, \ldots, t_n) \\ \text{Goal} & G & ::= & L \mid C \mid G \wedge G \\ \text{Rule} & R & ::= & H\ \texttt{:-}\ G \end{array}$$

where p is a n-ary predicate symbol from Π and $\bar{a} \equiv a_1, \ldots, a_n$ are distinct variables, and $t_1, \ldots, t_n$ are terms. A *program* P is set of rules. Notice that we use a different notation for predicates $p(t_1, \ldots, t_n)$ (also referred to as atoms) to separate them clearly from the predicates defined by the domain X. Predicates defined by the domain X start with upper-case letters (apart from $\cdot = \cdot$), whereas predicates defined by the CLP(X) program start with lower-case letters. Rules are implicitly universally quantified, hence the role of variables is just place-holders in rules. We can therefore freely α-rename bound variables.

A goal G is executed by SLD resolution with the rules in P. Let $G = G_1 \wedge p(\bar{t}) \wedge G_2$ and α-renaming of rule R in P of the form $p(\bar{a})\ \texttt{:-}\ G_3$ such that $fv(p(\bar{a})\ \texttt{:-}\ G_3) \cap fv(G) = \emptyset$, we create new goal $G' \equiv G_1 \wedge [\bar{t}/\bar{a}]G_3 \wedge G_2$. We write this as $G \rightarrowtail_R G'$.

A *derivation* for goal G using program P exhaustively applies SLD resolution, written $G \rightarrowtail_P^* G'$. The derivation is failed if G' is not a constraint or $\models_X \neg\tilde{\exists} G'$ when G' is a constraint, and successful otherwise. An *answer* for successful derivation is $\tilde{\exists}_{fv(G)}.G'$.

Example 6 Given the program for Example 3:

$$\begin{array}{lll} g(t,l) & \texttt{:-} & t = t_y \to t_1 \wedge l = [t_x] \wedge t_g = t_x \to t_1 \wedge t_g = t \\ f(t,l) & \texttt{:-} & t = t_x \to t_2 \wedge l = [] \wedge g(t_x \to t_2, [t_x]) \end{array}$$

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

The goal $f(a, [])$ has the successful derivation:

$$\begin{array}{lll} f(a,[]) & \rightarrowtail_f & a = t \wedge [] = l \wedge t = t_x \rightarrow t_2 \wedge l = [] \wedge g(t_x \rightarrow t_2, [t_x]) \\ & \rightarrowtail_g & a = t \wedge [] = l \wedge t = t_x \rightarrow t_2 \wedge l = [] \wedge t_x \rightarrow t_2 = t' \wedge [t_x] = l' \\ & & \wedge\ t' = t'_y \rightarrow t'_1 \wedge l' = [t'_x] \wedge t'_g = t'_x \rightarrow t'_1 \wedge t'_g = t' \\ & \leftrightarrow & a = t'_x \rightarrow t'_1 \wedge l = [] \wedge t = t'_x \rightarrow t'_1 \wedge t_x = t'_x \wedge t_2 = t'_1 \\ & & \wedge\ t' = t'_x \rightarrow t'_1 \wedge l' = [t'_x] \wedge t'_g = t'_x \rightarrow t'_1 \end{array}$$

where the last step simply gives an equivalent form of the constraints by substitution. The answer is the constraint $\exists t'_x \exists t'_1 . a = t'_x \rightarrow t'_1$.

We will restrict ourselves to programs P, which have at most one rule for each predicate symbol, that is, there are no two rules $p(\bar{a})$:- G and $p(\bar{a}')$:- G' with the same predicate symbol in the head. For these programs, we can interpret the rule L :- G as a logical formula: $\tilde{\forall} . L \leftrightarrow \tilde{\exists}_{fv(L)} . G$. Variables appearing exclusively on the right-hand side of a rule are existentially quantified. For example, the rule from Example 1 $f(t)$:- $t = t_x \rightarrow t_x$ is interpreted as $\forall t . f(t) \leftrightarrow (\exists t_x . t = t_x \rightarrow t_x)$. The logical interpretation of a program P, written $[\![P]\!]$, is simply the conjunction of the interpretation of each rule. This is a simplified form of the *program completion* (Jaffar *et al.* 1998), which defines the logical semantics of a CLP(X) program.

The following result is a consequence of the usual soundness and completeness results for CLP(X) (Jaffar & Lassez 1987; Jaffar *et al.* 1998).

> **Theorem 1 (Soundness and completeness of CLP(X) derivations)** Let P be a program, where for each predicate symbol there is at most one rule. Then $G \rightarrowtail_P^* G'$ implies that $[\![P]\!] \models G \leftrightarrow \tilde{\exists}_{fv(G)} . G'$

2.2 The HM(X) framework

We review the basics of the HM(X) system. In Odersky *et al.* (1999) and Sulzmann (2000), the constraint domain X was described in terms of a cylindric algebra (Henkin *et al.* 1971), which represents an algebraic formulation of a first-order theory. Here, we follow the CLP(X) description and describe X semantically in terms of a first-order logic.

The types t of the HM(X) scheme are simply terms in X and constraints C for the HM(X) scheme are simply constraints in X. In Odersky *et al.* (1999) and Sulzmann (2000), we also introduced some subtype constraints, which we ignore here for simplicity. We can straightforwardly support subtype constraints as long as the constraint domain X facilitates them.

Notice that constraints may be existentially quantified, see the upcoming typing rules (HM$\forall$ Intro) and (HM$\exists$ Intro).

The language of expressions and types schemes for HM(X) is as follows.

$$\begin{array}{llll} \text{Expressions} & e & ::= & f \mid x \mid \lambda x.e \mid e\ e \mid \mathsf{let}\ f = e\ \mathsf{in}\ e \mid \mathsf{rec}\ f\ \mathsf{in}\ e \\ \text{Type Schemes} & \sigma & ::= & t \mid \forall \bar{\alpha} . C \Rightarrow t \end{array}$$

We support the usual expressions such as function application and abstraction, nonrecursive let-defined functions and monomorphic recursive functions. Notice

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

$$\text{(HMVar)}\quad C,\Gamma \vdash v:\sigma \quad (v:\sigma \in \Gamma)$$

$$\text{(HMEq)}\quad \frac{C,\Gamma \vdash e:t_1 \quad C \models_X t_1 = t_2}{C,\Gamma \vdash e:t_2} \qquad \text{(HMLet)}\quad \frac{C,\Gamma \vdash e:\sigma \quad C,\Gamma{+\!+}[f:\sigma] \vdash e':t'}{C,\Gamma \vdash \mathsf{let}\ f = e\ \mathsf{in}\ e':t'}$$

$$\text{(HMAbs)}\quad \frac{C,\Gamma{+\!+}[x:t_1] \vdash e:t_2}{C,\Gamma \vdash \lambda x.e:t_1 \to t_2} \qquad \text{(HMApp)}\quad \frac{C,\Gamma \vdash e_1:t_1 \to t_2 \quad C,\Gamma \vdash e_2:t_1}{C,\Gamma \vdash e_1\ e_2:t_2}$$

$$\text{(HM}\forall\text{ Intro)}\quad \frac{C \wedge D,\Gamma \vdash e:t \quad \bar{a} \notin fv(\Gamma,C)}{C \wedge \exists \bar{a}.D,\Gamma \vdash e:\forall \bar{a}.D \Rightarrow t} \qquad \text{(HM}\forall\text{ Elim)}\quad \frac{C,\Gamma \vdash e:\forall \bar{a}.D \Rightarrow t' \quad C \models_X [\bar{t}/\bar{a}]D}{C,\Gamma \vdash e:[\bar{t}/\bar{a}]t'}$$

$$\text{(HM}\exists\text{ Intro)}\quad \frac{C,\Gamma \vdash e:\sigma \quad a \notin fv(\Gamma,\sigma)}{\exists a.C,\Gamma \vdash e:\sigma} \qquad \text{(HMRec)}\quad \frac{C,\Gamma{+\!+}[f:t] \vdash e:t}{C,\Gamma \vdash \mathsf{rec}\ f\ \mathsf{in}\ e:t}$$

Fig. 1. HM(X) typing rules.

that source expressions containing recursive let-defined functions such as

$$\mathsf{let}\quad g\ =\ \mathsf{let}\ f\ =\ \lambda x.f\ x\ \mathsf{in}\ e$$

must be de-sugared into

$$\mathsf{let}\quad g\ =\ (\mathsf{let}\ f\ =\ (\mathsf{rec}\ f'\ \mathsf{in}\ \lambda x.f'\ x)\ \mathsf{in}\ e$$

W.l.o.g., we assume that λ-bound and let-bound variables have been renamed to avoid name clashes. We commonly use $x, y, z, \ldots$ to refer to λ-bound variables and $f, g, h, \ldots$ to refer to user- and predefined functions. Both sets of variables are recorded in a variable environment Γ. We treat Γ as a list of type assignments of the form $[x_1 : \sigma_1, \ldots, x_n : \sigma_n]$. We use list concatenation $+\!+$ to indicate extension of environment Γ with type assignment $(x : \sigma)$ written $\Gamma{+\!+}[x : \sigma]$. We write $(x : \sigma) \in [x_1 : \sigma_1, \ldots, x_n : \sigma_n]$ to denote that x is equal to x_i and σ is equal to σ_i for some $i \in \{1, ..., n\}$. We assume that $fv([x_1 : \sigma_1, ..., x_n : \sigma_n]) = fv(\sigma_1) \cup ... \cup fv(\sigma_n)$. We use common shorthand notation $\mathsf{let}\ f\ x_1\ \cdots\ x_n = e$ for $\mathsf{let}\ f = \lambda x_1.\cdots\lambda x_n.e$ and omit the leading let for top-level functions.

We briefly discuss the typing rules in Figure 1, which make use of typing judgments of the form $C, \Gamma \vdash e : t$, where C is a constraint, Γ an environment, e an expression, and t a type. In rule (HMVar), we assume that v refers to either a λ- or a let-bound variable. Rule (HMEq) allows us to change the type t_1 of expression e to t_2 if both types are equivalent under the (assumption) constraint C. This rule is not strictly necessary but is convenient to have in some proofs such as the upcoming proof of Theorem 4 (Soundness of CLP(X)-style type inference) in Appendix B. In rule (HM$\forall$ Intro), we build type schemes by pushing in the "affected" constraint D. The existentially quantified constraint $\exists \bar{a}.D$ in the conclusion guarantees that if the "final" constraint in a typing derivation is satisfiable, all "intermediate" constraints must be satisfiable as well. In rule (HM$\forall$ Elim), we build a type instance by demanding that any model of our constraint domain X that satisfies C also satisfies the instantiated

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

$$\text{(TIVar)} \quad \frac{x : (\forall \bar{a}.D \Rightarrow t) \in \Gamma \qquad b \text{ fresh}}{\Gamma, x \vdash_W (\exists \bar{a}.(b = t \wedge D) \mid b)}$$

$$\text{(TIAbs)} \quad \frac{\Gamma{+\!\!+}[x : a], e \vdash_W (C \mid c) \quad a, b \text{ fresh}}{\Gamma, \lambda x.e \vdash_W (\exists a.\exists c.(b = a \to c \wedge C) \mid b)}$$

$$\text{(TIApp)} \quad \frac{\Gamma, e_1 \vdash_W (C_1 \mid a_1) \quad \Gamma, e_2 \vdash_W (C_2 \mid a_2) \quad a \text{ fresh}}{\Gamma, e_1\ e_2 \vdash_W (\exists a_1.\exists a_2.(C_1 \wedge C_2 \wedge a_1 = a_2 \to a) \mid a)}$$

$$\text{(TILet)} \quad \frac{\Gamma, e \vdash_W (C_1 \mid a) \quad \Gamma{+\!\!+}[f : \forall a.C_1 \Rightarrow a], e' \vdash_W (C_3 \mid b)}{\Gamma, \mathsf{let}\ f = e\ \mathsf{in}\ e' \vdash_W ((\exists a.C_1) \wedge C_3 \mid b)}$$

$$\text{(TIRec)} \quad \frac{\Gamma{+\!\!+}[f : a], e \vdash_W (C \mid b) \quad a \text{ fresh}}{\Gamma, \mathsf{rec}\ f\ \mathsf{in}\ e \vdash_W (\exists b.(C \wedge a = b) \mid a)}$$

Fig. 2. HM(X) type inference algorithm $\mathscr{W}$-style.

constraints $[\bar{t}/\bar{a}]D$, written $C \models_X [\bar{t}/\bar{a}]D$. Rule (HM$\exists$ Intro) allows us to simplify constraints by "hiding" variables not appearing anywhere but in constraint C. This is very useful when presenting inferred types to the user in our CLP(X) style inference scheme. See the upcoming discussion in Section 4.2 right after Theorem 4. Some readers may expect to find a dual rule (HM$\exists$ Elim). Elimination of $\exists$ is a form of weakening that is a meta-rule of the system. See Lemma 2 in Appendix B.

Rule (HMRec) allows for arbitrary (monomorphic) recursive values, not just for functions. This requires that the dynamic semantics of our language is nonstrict. In case of a strict language, we simply must guarantee that recursive values are functions. We briefly addressed how to deal with polymorphic recursion in the introduction. The remaining rules are those familiar from Hindley/Milner.

A point worth mentioning is that we do not require types to be in certain syntactic canonical form. For example, function $\lambda x.x$ can be given types $\forall a.a \to a$ and $\forall a, b.a = b \Rightarrow a \to b$. Both types are equivalent but we may favor $\forall a.a \to a$ for presentation purposes. In case of standard Hindley/Milner, we can always achieve a canonical representation of types by building the most general unifier. Perhaps surprisingly, there are variants of Hindley/Milner where a wrong choice of canonical form leads to incomplete type inference. We refer to Kennedy (1996) and Sulzmann (2001) for a discussion. Hence, we do not enforce syntactic canonical forms of types here. For an expression to be well typed, we only require that the constraints appearing in type judgments must be satisfiable.

Before we introduce our CLP(X)-style type inference approach, we review the classic algorithm $\mathscr{W}$ in the next section.

3 Constraint-based algorithm $\mathscr{W}$

In Figure 2, we introduce an algorithm $\mathscr{W}$ style inference system to give a syntax-directed description of the typing rules from the previous section. We employ inference judgments of the form $\Gamma, e \vdash_W (C \mid a)$, where environment Γ and expression

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

e are input values and constraint C and type a are output values. We maintain the invariant that a is a variable and $fv(C) = fv(\Gamma) \cup \{a\}$. Such a canonical representation of inference judgments, also found in Zenger (1999), makes building of type schemes in case of let-defined functions rather straightforward. See rule (TILet), which combines the rules for quantifier introduction with the rule for let statements.

Rules (TIAbs), (TIApp), and (TIRec) generate the appropriate constraints out of the program text. Like other inference algorithms, we need to generate "fresh" variables. We could represent "freshness" in a sufficiently rich logic (Urban *et al.* 2004) but we choose here to use a "half-logical" formulation of inference. As it is standard, rule (TIVar) combines variable introduction with quantifier elimination.

In contrast to Odersky *et al.* (1999) where we follow the "classic" formulation and thread through a substitution, representing the most general unifier of the constraints accumulated so far, we choose here a purely constraint-based formulation. For example, the constraint $a = b \to b \wedge b = Int$ represents the substitution $[Int \to Int/a, Int/b]$. In general, the output pair $(C \mid t)$ is a representation of the solutions in X of t.

We can straightforwardly verify that any inference derivation is also derivable in the system from the previous section.

> **Theorem 2 (Soundness of $\mathscr{W}$-style type inference)** Let Γ be an environment and e an expression such that $\Gamma, e \vdash_W (C \mid a)$ for some constraint C and type a. Then, $C, \Gamma \vdash e : \alpha$.

The result can be proven by straightforward induction over $\vdash_W$.

To state completeness, we introduce a comparison relation $\vdash^i_X$ among type schemes. We define $C_1 \vdash^i_X (\forall \bar{a}_2.C_2 \Rightarrow t_2) \preceq (\forall \bar{a}_3.C_3 \Rightarrow t_3)$ iff $C_1 \wedge C_3 \models_X \exists \bar{a}_2.(C_2 \wedge t_2 = t_3)$, where we assume that there are no name clashes between $\bar{a}_2$ and $\bar{a}_3$. The comparison relation can be easily extended to types by considering t as a shorthand for $\forall a.a = t \Rightarrow a$, where a is fresh.

In case $C \vdash^i_X \sigma_1 \preceq \sigma_2$, we say that σ_1 is *more general* than σ_2. We will verify that for any type derived by the HM(X) typing rules, there is a more general type derived by the inference algorithm.

We say that Γ is *realizable* in C iff for each $x : \sigma \in \Gamma$ there exists a type t such that $C \vdash^i_X \sigma \preceq t$.

> **Theorem 3 (Completeness of $\mathscr{W}$-style type inference)** If $C, \Gamma \vdash e : \sigma$ and Γ is realizable in C, then $\Gamma, e \vdash_W (C' \mid a)$ such that $C \vdash^i_X (\forall a.C' \Rightarrow a) \preceq \sigma$ and $C \models_X \exists a.C'$.

The realizability condition is necessary to establish $C \models_X \exists a.C'$ in case of variables. In case of let statements, we need $C \models_X \exists a.C'$ to establish $C \vdash^i_X (\forall a.C' \Rightarrow a) \preceq \sigma$. The details of the proof are given in Appendix A.

The constraint-based reformulation of algorithm $\mathscr{W}$ represents a first step in rephrasing HM(X) type inference as CLP(X) solving. Constraint generation proceeds in the same way. The major difference is that each let-defined function is turned into a CLP(X) rule. This is what we discuss next.

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

4 HM(X) type inference is CLP(X) solving

As highlighted, the basic idea is that for each definition `f = e`, we introduce a CLP(X) rule of the form $f(t, l)$ `:-` G by performing a form of λ-lifting on the level of types. A similar concept was introduced previously in Birkedal & Tofte (2001). The type parameter t refers to the type of `f`, whereas l refers to the set of types of λ-bound variables in scope (i.e., the set of types of free variables that come from the enclosing definition). The reason for l is that we must ensure that λ-bound variables remain monomorphic. The goal G contains the constraints generated out of expression `e` plus some additional constraints restricting l. Thus, we can explain HM(X) type inference as running the CLP(X) program resulting from e on the constraints generated out of e. Before we dive into the formal details, we explain one more subtle point of our CLP(X)-style type inference scheme.

So far, we assumed that at the definition and call sites of `f` we set l to the exact set of types of all free (λ) variables in scope. Hence, we actually need to compute the exact set before we can generate the CLP(X) program. We can avoid these tedious computations by using a slightly different approach. The following example shows how this works.

Example 7 Consider

```
k z = let h w = (w,z)
      in  let f x = let g y = (x,y)
                    in (g 1, g True, h 3)
          in f z
```

A (partial) description of the CLP(X) program resulting from the above program text might look as follows. For simplicity, we leave out the constraints generated out of expressions. We write t_x to denote the type of λ-bound variable `x` and so on.

$$\begin{array}{lllll} \text{(k)} & k(t,l) & \texttt{:-} & l = [] \wedge \cdots \\ \text{(h)} & h(t,l) & \texttt{:-} & l = [t_z] \wedge \cdots \\ \text{(f)} & f(t,l) & \texttt{:-} & l = [t_z] \wedge \cdots \\ \text{(g)} & g(t,l) & \texttt{:-} & l = [t_z, t_x] \wedge \cdots \end{array}$$

In each CLP(X) rule, the l parameter refers exactly to the set of types of all free (λ) variables in scope of the corresponding function.

Consider the subexpression `(g 1, g True, h 3)`. At each instantiation site, we need to specify correctly the sequence of types of λ-bound variables that were in scope at the function definition site. For example, λ-variables `z` and `x` are in scope of `g y = ...`, whereas only `z` is in scope of `h w = ...`. Among others, we generate

$$g(t_1, l_1) \wedge l_1 = [t_z, t_x] \wedge t_1 = \mathit{Int} \rightarrow t_1' \wedge\ g(t_2, l_2) \wedge l_2 = [t_z, t_x] \wedge t_2 = \mathit{Bool} \rightarrow t_2' \\ \wedge\ h(t_3, l_3) \wedge l_3 = [t_z] \wedge t_3 = \mathit{Int} \rightarrow t_3' \wedge \cdots$$

The point is that at function instantiation sites our constraint generation algorithm needs to remember correctly the sequence of types of λ-variables that were in scope at the function definition site. To avoid such tedious calculations, the sequence of types of λ-bound variables in scope for function definitions is left "open." We

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

indicate this by writing $t_1 : \cdots : t_n : r$, which denotes a (type-level) list with an n-element list $[t_1, ..., t_n]$, representing the types of λ-bound variables, but an unbounded tail represented by a fresh type variable r. The set of types of λ-bound variables at function instantiation sites corresponds to stack of type of λ-bound variables in the sequence of their definition.

On the basis of this scheme, our actual translation scheme yields the following result:

$$\begin{array}{lll}
\text{(k)} & k(t,l) & :- \quad t = t_1 \to t_2 \wedge f(t,l_1) \wedge l_1 = [t_z] \wedge t_1 = t_z \\
\text{(h)} & h(t,l) & :- \quad \underline{l = t_z : r} \wedge t = t_w \to (t_w, t_z) \\
\text{(f)} & f(t,l) & :- \quad l = t_z : r \wedge t = (t'_1, t'_2, t'_3) \wedge g(t_1, l_1) \\
 & & \quad \wedge\ l_1 = [t_z, t_x] \wedge t_1 = Int \to t'_1 \\
 & & \quad \wedge\ g(t_2, l_2) \wedge l_2 = [t_z, t_x] \wedge t_2 = Bool \to t'_2 \\
 & & \quad \wedge\ h(t_3, l_3) \wedge \underline{l_3 = [t_z, t_x]} \wedge t_3 = Int \to t'_3 \\
\text{(g)} & g(t,l) & :- \quad l = t_z : t_x : r \wedge t = t_y \to (t_x, t_y)
\end{array}$$

In the h rule, we require that variable `z`, whose type is t_z, is in scope plus possibly some more variables (see underlined constraint). Observe that in rule f, we pass in the (somewhat redundant) variable t_x as part of the x parameter at the instantiation site of `h` (see underlined constraint). There is no harm in doing so, because there is no reference to variable t_x on the right-hand side of rule h.

For example, consider the following derivation step:

$$h(t_3, l_3) \wedge l_3 = [t_z, t_x] \rightarrowtail_h l_3 = t'_z : r' \wedge t_3 = t'_w \to (t'_w, t'_z) \wedge l_3 = [t_z, t_x],$$

where we denote renamed rule variables via a prime. We find that $l_3 = t'_z : r' \wedge l_3 = [t_z, t_x]$ implies $t'_z = t_z$ and $r' = [t_x]$. Thus, we establish that both references of t_z in rules h and f refer to the same type without having to compute the exact set of λ-bound variables in scope of `h` at the call site $h(t_3, l_3)$.

We are now well prepared to take a look at the formal translation scheme that consists of two main parts: generating constraints from expressions and building of CLP(X) rules for function definitions.

4.1 Translation to CLP(X)

Constraint generation is similar to algorithm $\mathscr{W}$ (see Figure 2). A minor difference is that we return type terms, not just variables. The essential difference is that we additionally need to record information about the predicates connected to let-defined (or primitive) functions. Hence, we use constraint generation judgments of the form $E, \Gamma, e \vdash_{Cons} (G \mid t)$, where the environment E of all let-defined and predefined functions, environment Γ of λ-bound variables, and expression e are input parameters and goal G and type t are output parameters. The details are given in Figure 3.

In rule (CGVar-x), we simply look up the type of a λ-bound variable in Γ. In rule (CGVar-f), the goal $f(t,l) \wedge l = [t_{x_1}, \ldots, t_{x_n}]$ demands on instance of `f` on type t, where $(t_{x_1}, \ldots, t_{x_n})$ refers to the set of types of λ-bound variables in scope. In essence, we build a generic instance of `f`'s type. The actual type of `f` will be described by a

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

$$\text{(CGVar-x)}\quad \frac{(x:t)\in\Gamma}{E,\Gamma,x\vdash_{Cons}\ (\mathit{True}\mid t)}$$

$$\text{(CGVar-f)}\quad \frac{f\in E\quad t,\,l\text{ fresh}}{E,[x_1:t_1,\ldots,x_n:t_n],f\vdash_{Cons}\ (f(t,l)\wedge l=[t_1,\ldots,t_n]\mid t)}$$

$$\text{(CGAbs)}\quad \frac{E,\Gamma{+\!+}[x:t_1],e\vdash_{Cons}\ (G\mid t_2)\quad t_1\text{ fresh}}{E,\Gamma,\lambda x.e\vdash_{Cons}\ (G\mid t_1\to t_2)}$$

$$\text{(CGApp)}\quad \frac{E,\Gamma,e_1\vdash_{Cons}\ (G_1\mid t_1)\quad E,\Gamma,e_2\vdash_{Cons}\ (G_2\mid t_2)\quad t\text{ fresh}}{E,\Gamma,e_1\ e_2\vdash_{Cons}\ (G_1\wedge G_2\wedge t_1=t_2\to t\mid t)}$$

$$\text{(CGLet)}\quad \frac{\begin{array}{c}E\cup\{f\},\Gamma_\lambda,e_2\vdash_{Cons}\ (G\mid t)\\ \Gamma_\lambda=[x_1:t_1,\ldots,x_n:t_n]\quad a,l\text{ fresh}\end{array}}{E,\Gamma_\lambda,\mathsf{let}\ f=e_1\ \mathsf{in}\ e_2\vdash_{Cons}\ (G\wedge f(a,l)\wedge l=[t_1,\ldots,t_n]\mid t)}$$

$$\text{(CGRec)}\quad \frac{E,\Gamma{+\!+}[f:a],e\vdash_{Cons}\ (G\mid t)\quad a\text{ fresh}}{E,\Gamma,\mathsf{rec}\ f\ \mathsf{in}\ e\vdash_{Cons}\ (G\wedge a=t\mid t)}$$

Fig. 3. Constraint generation.

$$\text{(RGVar)}\quad E,\Gamma,v\vdash_{Def}\ \emptyset$$

$$\text{(RGAbs)}\quad \frac{E,\Gamma{+\!+}[x:t],e\vdash_{Def}\ P\quad t\text{ fresh}}{E,\Gamma,(\lambda x.e)\vdash_{Def}\ P}$$

$$\text{(RGApp)}\quad \frac{E,\Gamma,e_1\vdash_{Def}\ P_1\quad E,\Gamma,e_2\vdash_{Def}\ P_2}{E,\Gamma,e_1\ e_2\vdash_{Def}\ P_1\cup P_2}$$

$$\text{(RGLet)}\quad \frac{\begin{array}{c}E,\Gamma,e_1\vdash_{Cons}\ (G\mid t)\quad \Gamma=[x_1:t_1,\ldots,x_n:t_n]\quad l,r\text{ fresh}\\ E,\Gamma,e_1\vdash_{Def}\ P_1\quad E\cup\{f\},\Gamma,e_2\vdash_{Def}\ P_2\\ P=P_1\cup P_2\cup\{f(t,l)\ \texttt{:-}\ G\wedge l=t_1:\ldots:t_n:r\}\end{array}}{E,\Gamma,\mathsf{let}\ f=e_1\ \mathsf{in}\ e_2\vdash_{Def}\ P}$$

$$\text{(RGRec)}\quad \frac{E,\Gamma,e\vdash_{Def}\ P}{E,\Gamma,\mathsf{rec}\ f\ \mathsf{in}\ e\vdash_{Def}\ P}$$

Fig. 4. CLP(X) rule generation.

CLP(X) rule where the set of types of λ-bound variables is left open. Notice that in case $f\notin E$, function f is undefined. If f is defined, we will add f to E when typing the body of the let statement. See the upcoming rule (RGLet) for rule generation in Figure 4. Type assignments in the environment Γ are ordered according to the scope of variables. See rule (CGAbs). Rules (CGApp) and (CGRec) contain no surprises.

In rule (CGLet), we process a let statement by recording the predicate associated to the CLP(X) rule of $\mathsf{let}\ f=e_1\ \mathsf{in}\ e_2$. Then, we collect the constraints arising from the let body e_2. In algorithm $\mathscr{W}$, we also collect the constraints from e_1. In the

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

CLP(X)-style inference scheme, we collect these constraints by querying the type of f via its predicate. We say a let-defined function f is *let-realizable* if f is actually used in the let-body e_2. If this is the case, the constraint $f(a,l) \wedge l = [t_1,...,t_n]$ is redundant (and can therefore can be omitted) because the goal G already contains a call to f. In the upcoming section, we provide examples explaining this point in more detail.

Generation of CLP(X) rules is formulated in terms of judgments of the form $E, \Gamma, e \vdash_{Def} P$, where input parameters E, Γ, and e are as before and the set P of CLP(X) rules is the output parameter. For each function definition, we generate a new rule. See Figure 4 for details. As discussed, we leave the set of types of λ-bound variables open at definition sites. See rule (RGLet). If Γ is empty, we set $l = r$.

4.2 Type inference via CLP(X) solving

The actual type inference applies the CLP(X) program, which is the set of CLP(X) rules generated, to the resulting constraint. More formally, let (Γ, e) be an HM(X) type inference problem where we assume that Γ can be split into a component Γ_{init} and Γ_λ such that $fv(\Gamma_{init}) \subseteq fv(\Gamma_\lambda)$ and types in Γ_λ are simple, that is, not universally quantified. In essence, we demand that if a type scheme in Γ contains an unbound variable, it must be mentioned in some simple type. For each function f in Γ_{init}, we introduce a binary predicate symbol f, which we record in E_{init}. We build a set $P_{E_{init}}$ of CLP(X) rules by generating for each $f : \forall \bar{a}.C \Rightarrow t \in \Gamma_{init}$ the rule $f(t', l) \text{ :- } C \wedge t' = t$, where t' and l are fresh. In such a situation, we write $P_{E_{init}}, E_{init} \sim \Gamma_{init}, \Gamma_\lambda$.

Type inference proceeds as follows: We first compute $E_{init}, \Gamma_\lambda, e \vdash_{Cons} (G \mid t)$ and $E_{init}, \Gamma_\lambda, e \vdash_{Def} P$. To infer the type of e, we run $P \cup P_{E_{init}}$ on goal G. By construction, $P \cup P_{E_{init}}$ is *terminating*. That is, $G \rightarrowtail^*_{P \cup P_{E_{init}}} D$ for some D, where D is a constraint (it only contains predicates defined by the constraint domain X). If D is unsatisfiable, we report a type error. Otherwise, we can conclude that expression e has type $\forall \bar{a}.D \Rightarrow t$, where $\bar{a} = fv(D,t) - fv(\Gamma_\lambda)$.

The termination argument for $P \cup P_{E_{init}}$ goes as follows. To each let-defined function symbol f, we assign a unique number based on a depth-first left-to-right traversal of the AST. We assume that numbers will increase during the traversal. Then, for each generated rule $f(t,l) \text{ :- } G \wedge l = t_1 : ... : t_n : r$ in P, we find that the number of let-defined function symbols appearing in G is greater than the number of f. Immediately, we can conclude that the generated CLP(X) P program is nonrecursive. Hence, running any goal on $P \cup P_{E_{init}}$ will terminate.

We can verify that the types thus computed are derivable in the HM(X) type system from Section 2.2 (soundness) and any HM(X) type can be computed by the CLP(X)-style inference scheme (completeness).

Theorem 4 (Soundness of CLP(X)-style type inference)
Let $P_{E_{init}}, E_{init} \sim \Gamma_{init}, \Gamma_\lambda$, and $E_{init}, \Gamma_\lambda, e \vdash_{Cons} (G \mid t)$ and $E_{init}, \Gamma_\lambda, e \vdash_{Def} P$ such that $G \rightarrowtail^*_{P_{E_{init}} \cup P} D$. Then, $D, \Gamma_{init} \cup \Gamma_\lambda \vdash e : t$.

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

For presentation purposes, we may want to "normalize" the constraint D and type t into an equivalent but more readable form. Let us consider Example 2 again. Our (slightly abbreviated) translation scheme for the program text

```
g y = let f x = (y,x) in (f True, f y)
```

generates

$$\begin{array}{lll} g(t,l) & :- & t = t_y \to (t_1,t_2) \wedge f(t_{f1},[t_y]) \wedge t_{f1} = Bool \to t_1 \wedge f(t_{f2},[t_y]) \\ & & \wedge\ t_{f2} = t_y \to t_2 \\ f(t,l) & :- & t = t_x \to (t_y,t_x) \wedge l = t_y : r \end{array}$$

Function f is let-realizable, that is, used in the body of the let statement. Therefore, we abbreviate the translation by omitting the constraint $f(a,l) \wedge l = [t_y]$, which would usually appear on the right-hand side of the CLP(X) rule g according to the constraint generation rule (CGLet).

We infer g's type by executing

$$\begin{array}{lll} g(t,[]) & \rightarrowtail_g & t = t_y \to (t_1,t_2) \wedge f(t_{f1},[t_y]) \wedge t_{f1} = Bool \to t_1 \\ & & \wedge\ f(t_{f2},[t_y]) \wedge t_{f2} = t_y \to t_2 \\ & \rightarrowtail_f & t = t_y \to (t_1,t_2) \wedge t_{f1} = t'_x \to (t'_y,t'_x) \wedge [t_y] = t'_y : r' \wedge t_{f1} = Bool \to t_1 \\ & & \wedge\ f(t_{f2},[t_y]) \wedge t_{f2} = t_y \to t_2 \\ & \rightarrowtail_f & t = t_y \to (t_1,t_2) \wedge t_{f1} = t'_x \to (t'_y,t'_x) \wedge [t_y] = t'_y : r' \wedge t_{f1} = Bool \to t_1 \\ & & \wedge\ t_{f2} = t''_x \to (t''_y,t''_x) \wedge [t_y] = t''_y : r'' \wedge t_{f2} = t_y \to t_2 \end{array}$$

On the basis of the above soundness result, we find that g has type

$$\forall t,t_1,t_2,t_{f1},t'_x,t'_y,r',t_y,t_{f2},t''_x,t''_y,r''. \left(\begin{array}{l} t = t_y \to (t_1,t_2) \wedge t_{f1} = t'_x \to (t'_y,t'_x) \wedge [t_y] = t'_y : r' \wedge t_{f1} = Bool \to t_1 \\ \wedge\ t_{f2} = t''_x \to (t''_y,t''_x) \wedge [t_y] = t''_y : r'' \wedge t_{f2} = t_y \to t_2 \end{array} \right) \Rightarrow t$$

In this example, we employ the Herbrand domain H, that is, HM(H). Hence, we can normalize the above type by building the most general (Herbrand) unifier.

$$\forall t,t_1,t_2,t_{f1},t'_x,t'_y,r',t_y,t_{f2},t''_x,t''_y,r''. \left(\begin{array}{l} t = t_y \to ((t_y,Bool),(t_y,t_y)) \wedge t_1 = (t_y,Bool), t_2 = (t_y,t_y) \\ \wedge\ t_{f1} = Bool \to (t_y,Bool) \wedge t'_x = Bool \wedge t_y = t'_y \wedge r' = [] \\ \wedge\ t_{f2} = t_y \to (t_y,t_y) \wedge t''_x = t_y \wedge t_y = t''_y \wedge r'' = [] \end{array} \right) \Rightarrow t$$

Notice that in general all equations $[s_1,...,s_n] = s'_1 : ... : s'_k : r$ where $k \leqslant n$ can be replaced by $s_i = s'_i$ for $i = 1,...,k$ and $r = [s_{k+1},...,s_n]$. Recall that $\cdot : \cdot$ and [] are Herbrand constructors. For the above example, we therefore find that $[t_y] = t'_y : r' \wedge [t_y] = t''_y : r''$ are replaced by $t_y = t'_y \wedge t_y = t''_y \wedge r' = [] \wedge r'' = []$. Since r' and r'' appear nowhere else, we can remove the constraints $r' = []$ and $r'' = []$. This is justified by typing rule (HM$\exists$ Intro) in Figure 1 and the fact that $\exists r'.r' = []$ is equivalent to *True*. Thus, we arrive at a "pure" constraint without the added constructors $\cdot : \cdot$ and [].

In fact, we can also remove the constraints connected to variables t_1, t_2, t_{f1}, t'_x, t'_y, t_{f2}, t''_x, and t''_y because they do not appear in the "output" constraint $t = t_y \to ((t_y,Bool),(t_y,t_y))$. This step is again justified by typing rule (HM$\exists$ Intro) in Figure 1

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

and the fact that

$$\exists t_1, t_2, t_{f1}, t'_x, t'_y, t_{f2}, t''_x, t''_y.\left(\begin{array}{l} t = t_y \rightarrow ((t_y, Bool), (t_y, t_y)) \wedge t_1 = (t_y, Bool), t_2 = (t_y, t_y) \\ \wedge\ t_{f1} = Bool \rightarrow (t_y, Bool) \wedge t'_x = Bool \wedge t_y = t'_y \\ \wedge\ t_{f2} = t_y \rightarrow (t_y, t_y) \wedge t''_x = t_y, t_y = t''_y \end{array}\right)$$

is equivalent to $t = t_y \rightarrow ((t_y, Bool), (t_y, t_y)$. Hence, g's type can be equivalently represented by

$$\forall t, t_y.t = t_y \rightarrow ((t_y, Bool), (t_y, t_y)) \Rightarrow t$$

which we can display as $\forall t_y.t_y \rightarrow ((t_y, Bool), (t_y, t_y))$.

In general, normalization of types will depend on the specific constraint domain X in use. For instance, in Haskell 98 (Peyton Jones 2003), we remove "redundant" superclass constraints, for example, $\forall a.(Ord\ a \wedge Eq\ a) \Rightarrow a$ is normalized to $\forall a.Ord\ a \Rightarrow a$.

Next, we discuss the purpose of the "let-realizability" constraint $f(a, l) \wedge l = [t_1, ..., t_n]$ in rule (CGLet).

Example 8 Consider the following ill-typed expression.

```
e = let f = True True
    in False
```

If we omit the constraint $f(a, l) \wedge l = [t_1, ..., t_n]$ in rule (CGLet), the translation to CLP(X) yields

$$\begin{array}{lll} f(t) & :- & t_1 = Bool \wedge t_1 = t_2 \rightarrow t_3 \wedge t_2 = Bool \wedge t_3 = t \\ e(t) & :- & t = Bool \end{array}$$

For simplicity, we also omit the l component, which does not matter here.

Type inference for expression `e` succeeds, although function `f` is ill-typed. We find that $e(t) \rightarrowtail^* t = Bool$. The problem is that there is no occurrence of `f` in the let body, hence we never execute the CLP(X) rule belonging to `f`. In a traditional inference approach such as $\mathscr{W}$, inference for `e` proceeds by first inferring the type of `f` immediately detecting that `f` is not well-typed. Therefore, our actual translation scheme generates

$$\begin{array}{lll} f(t) & :- & t_1 = Bool \wedge t_1 = t_2 \rightarrow t_3 \wedge t_2 = Bool \wedge t_3 = t \\ e(t) & :- & t = Bool \wedge f(a) \end{array}$$

The conclusion is that the "let-realizability" constraint $f(a, l) \wedge l = [t_1, ..., t_n]$ in rule (CGLet) is necessary to guarantee soundness of the CLP(X)-style inference scheme with respect to the HM(X) typing rules. We conjecture that under a nonstrict semantics rule (CGLet) is still sound (in the sense of programs will not go wrong at run-time) if we omit $f(a, l) \wedge l = [t_1, ..., t_n]$. In this respect, typing of programs in CLP(X) seems more flexible than typing in HM(X).

We conclude this section by stating completeness.

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

Theorem 5 (Completeness of CLP(X)-style type inference)
Let $P_{E_{init}}, E_{init} \sim \Gamma_{init}, \Gamma_\lambda$ and $C', \Gamma_{init} \cup \Gamma_\lambda \vdash e : t'$. Then, $E_{init}, \Gamma_\lambda, e \vdash_{Cons} (G \mid t)$ and $E_{init}, \Gamma_\lambda, e \vdash_{Def} P$ for some goal G, type t and CLP(X) program P such that $C' \vdash^i_X (\forall \bar{a}.D \Rightarrow t) \preceq t'$ where $G \rightarrowtail^*_{P_{E_{init}} \cup P} D$ and $\bar{a} = fv(D, t) - fv(\Gamma_\lambda)$.

Proofs for the above results can be found in Appendix 6.

5 Related work and discussion

There are numerous works that study type inference for Hindley/Milner-style systems. We refer to Pottier and Rémy (2005) and the references therein.

Most works on Hindley/Milner-style type inference focus on the domain-specific solver X and employ standard inference algorithms such as $\mathscr{W}$, $\mathscr{M}$, etc. The basic structure of such standard algorithms is the same. Type inference proceeds by generating constraints out of the program text while traversing the AST. We will need to solve these constraints at the latest once we visit a let node in order that we can build a type scheme. We refer to Fuh and Mishra (1990), Aiken and Wimmers (1992) and Palsberg and Smith (1996) for a selection of early works on solving constraints. To the best of our knowledge, the first work on solving constraints via CHRs in the context of type inference is our own work reported in Glynn *et al.* (2000) which subsequently led to Stuckey & Sulzmann (2005). Further works on using CHRs to solve type constraints include Alves and Florido (2002) and Coquery and Fages (2002).

There are only a few works that consider a fundamentally different inference approach where the *entire* type inference is mapped to a constraint problem.

The earliest reference we can find in the literature is some work by Dietzen and Pfenning (1991) who employ λProlog's (Nadathur & Miller 1988) higher order abstraction facilities for type inference. Effectively, they translate the Hindley/Milner inference problem into a "nested" Horn clause program. For instance, the program text

```
g y = let f x = (y,x) in (f True, f y)
```

from the earlier Example 2 is (roughly) translated to

$$g(t) \ \texttt{:-} \ \left(\begin{array}{l} t = t_y \rightarrow (t_1, t_2) \wedge f(t_{f1}) \wedge t_{f1} = Bool \rightarrow t_1, f(t_{f2}) \wedge t_{f2} = t_y \rightarrow t_2 \\ \wedge \ (\ f(t) \quad \texttt{:-} \quad t = t_x \rightarrow (t_y, t_x)\ ) \end{array} \right)$$

in Dietzen and Pfenning's approach. Notice the "nested" Horn clause f, which captures the type of `f` and also has a reference to the type variable t_x from the enclosing function g. Hence, different calls to f will refer to the same t_x.

Similar ideas of phrasing Hindley/Milner type inference in terms of a calculus with higher order abstraction can be found in the work of Müller (1994) and Liang (1997). Pottier and Rémy (2005) introduce a constraint domain with an explicit "let" construct for the same purpose.

In contrast to these works, Mycroft and O'Keefe (1984) map Hindley/Milner type checking of logic programs to a logic program. In some later work, Lakshman and Reddy (1991) established a semantic soundness result that was missing in Mycroft

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

and O'Keefe (1984). Demoen *et al.* (1999) extend this approach to allow inference and ad hoc overloading. They also provide a specialized solver for disjunctive Herbrand constraints to improve worst case behavior.

6 Conclusion

In this work, we extend the approach of Demoen *et al.* (1999) to handle expressions containing nested let definitions (which do not arise in logic programs). To translate HM(X) type inference to CLP(X) rules and solving, we perform a form of λ-lifting on the level of types. A similar idea can be found in the work by Birkedal and Tofte (2001). Most importantly, we abstract away from the Herbrand constraint domain to an arbitrary constraint domain X. We formally verify for the first time that Hindley/Milner inference is equivalent to CLP(X) solving. We can cover a wide range of Hindley/Milner-style systems by appropriately instantiating X with a domain-specific solver. The Chameleon system (Sulzmann & Wazny 2007) implements the CLP(X)-style inference scheme where the constraint domain is specifiable in terms of CHRs.

In general, the complexity of Hindley/Milner type inference is exponential (Kanellakis *et al.* 1991). Experience shows that type inference works well in practice. This observation is supported by some theoretical studies, for example, consider McAllester (2003). The approach defined in this article is highly practical and is implemented in the Chameleon (Sulzmann & Wazny 2007) system, where X is specifiable using CHRs (Frühwirth 1995).

Acknowledgments

We thank the reviewers for their helpful feedback on earlier drafts of this article.

APPENDIX A

Proof of Theorem 3 (Completeness of $\mathscr{W}$-style type inference)

We verify Theorem 3 by induction over the typing derivation. To ensure that the inductive proof will go through, we strengthen the statement (an idea that dates back to Damas & Milner 1982).

First, we introduce some notation. We write $C \vdash^i_X \Gamma' \preceq \Gamma$ if $\Gamma = [x_1 : \sigma_1, ..., x_n : \sigma_n]$ and $\Gamma' = [x_1 : \sigma'_1, ..., x_n : \sigma'_n]$ and for each $x : \sigma' \in \Gamma'$, $x : \sigma \in \Gamma$ we have that $C \vdash^i_X \sigma' \preceq \sigma$.

The completeness result follows from the following more general lemma.

> **Lemma 1** Let $C, \Gamma \vdash e : \sigma, \Gamma$ be realizable in C, $C'' \vdash^i_X \Gamma' \preceq \Gamma$ and $C'' \models_X C$. Then $\Gamma', e \vdash_W (C' \mid a)$ for some C', α such that $C'' \vdash^i_X (\forall \alpha. C' \Rightarrow \alpha) \preceq \sigma$ and $C'' \models_X \exists a. C'$.

Proof

Recall that Γ is realizable in C iff for each $x : \sigma \in \Gamma$ there exists a type t such that $C \vdash^i_X \sigma \preceq t$.

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

In the proof, we often omit parentheses by assuming that $\wedge$ binds tighter than $\exists$. Hence, $\exists a.C_1 \wedge C_1$ is a short form for $\exists a.(C_1 \wedge C_1)$.

The proof proceeds by induction over the derivation $C, \Gamma \vdash e : \sigma$. We omit cases (HMEq), (HM$\exists$ Intro), and (HMRec) for simplicity.

Case (HMVar) We find the following situation

$$C, \Gamma \vdash v : \sigma \quad (v : \sigma \in \Gamma)$$

Let us assume that σ is of the form $\forall \bar{a}.D \Rightarrow t$ and $v : \forall \bar{a}'.D' \Rightarrow t' \in \Gamma'$.

We find that

$$\Gamma', v \vdash_W (\exists \bar{a}'.b = t' \wedge D' \mid b)$$

We have to show that

$$C'' \vdash_X^i (\forall b.\exists \bar{a}'.b = t' \wedge D' \Rightarrow b) \leq (\forall \bar{a}.D \Rightarrow t)$$

which follows immediately from $C'' \vdash_X^i \Gamma' \leq \Gamma$ and the fact that

$$C'' \vdash_X^i (\forall b.\exists \bar{a}'.b = t' \wedge D' \Rightarrow b) \leq (\forall \bar{a}'.D' \Rightarrow t')$$

We yet need to verify that $C'' \models_X \exists b, \bar{a}'.b = t' \wedge D'$. The realizability assumption implies that $C \models_X \exists \bar{a}.D$. Hence, we also find $C'' \models_X \exists \bar{a}.D$ (1) because of $C'' \models_X C$ (by assumption). From

$$C'' \vdash_X^i (\forall b.\exists \bar{a}'.b = t' \wedge D' \Rightarrow b) \leq (\forall \bar{a}.D \Rightarrow t)$$

we can see that

$$C'' \wedge D \models_X \exists b, \bar{a}'.b = t' \wedge D' \quad (2)$$

We know that $\bar{a}$ does not appear in D'. Hence, from (2) we can conclude

$$C'' \wedge \exists \bar{a}.D \models_X \exists b, \bar{a}'.b = t' \wedge D' \quad (3)$$

From (1) and (3), we can finally conclude $C'' \models_X \exists b, \bar{a}'.b = t' \wedge D'$.

Case (HMAbs) We find the following situation:

$$\frac{C, \Gamma{+}{+}[x : t] \vdash e : t'}{C, \Gamma \vdash \lambda x.e : t \to t'}$$

We have that $C'' \vdash_X^i \Gamma' \leq \Gamma$ and $C'' \models_X C$. Then,

$$C'' \wedge \alpha = t \vdash_X^i \Gamma{+}{+}[x : \alpha] \leq \Gamma{+}{+}[x : t]$$

where α is fresh. Application of the induction hypothesis to the premise yields

$$\begin{gathered} \Gamma'{+}{+}[x : \alpha], e \vdash_W (C' \mid \alpha') \\ C'' \wedge \alpha = t \vdash_X^i (\forall \alpha'.C' \Rightarrow \alpha') \leq t' \quad (1) \\ C'' \wedge a = t \models_X \exists a'.C' \quad (2) \end{gathered}$$

for some constraint C' and type variable α'. Application of the (TIAbs) rule yields

$$\Gamma', \lambda x.e \vdash_W (\exists \alpha, \alpha'.(C' \wedge \alpha'' = \alpha \to \alpha') \mid \alpha'')$$

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

where α'' is a new type variable. We first show that

$$C'' \vdash^i_X (\forall\alpha''.\exists\alpha,\alpha'.(C' \wedge \alpha'' = \alpha \to \alpha') \Rightarrow \alpha'') \preceq t \to t'$$

The above is equivalent to $C'' \models_X \exists a'', a', a.C' \wedge a'' = a \to a' \wedge a'' = t \to t'$ (3).

From (1), we can conclude that

$$C'' \wedge a = t \models_X \exists a'.C \wedge a' = t'$$

which implies that

$$\phi(C'' \wedge a = t) \models_X \phi(\exists a'.C \wedge a' = t') \quad (4)$$

where $\phi = [t/a]$. We can assume that $a \notin fv(C'')$. Hence, $\phi(C'' \wedge a = t) = C''$. We write $=$ to denote logical equivalence among constraints. The constraint $\phi(\exists a'.C \wedge a' = t')$ is equivalent to $\exists a, a'.C \wedge a' = t' \wedge a = t$. We simply represent substitution via existential quantification. Hence, from (4) we can conclude

$$C'' \models_X \exists a, a'.C \wedge a' = t' \wedge a = t$$

which implies (3) by introducing the "intermediate" variable a''.

It remains to verify that

$$C'' \models_X \exists a, a', a''.C' \wedge a'' = a \to a'$$

From (2) via a similar reasoning as above, we can conclude

$$C'' \models_X \exists a, a'.C \wedge a = t$$

which implies (by weakening) $C'' \models_X \exists a, a'.C$. Variable a'' does not appear in C. Hence, we can conclude that

$$C'' \models_X \exists a, a', a''.C \wedge a'' = a \to a'$$

and we are done.

Case (HMApp) We have the following situation:

$$\frac{C, \Gamma \vdash e_1 : t_1 \to t_2 \qquad C, \Gamma \vdash e_2 : t_1}{C, \Gamma \vdash e_1\ e_2 : t_2}$$

Application of the induction hypothesis to the left and right premise yields

$$\begin{array}{cc} \Gamma', e_1 \vdash_W (C_1 \mid \alpha_1) & \Gamma', e_2 \vdash_W (C_2 \mid \alpha_2) \\ C'' \vdash^i_X (\forall\alpha_1.C_1 \Rightarrow \alpha_1) \preceq t_1 \to t_2 & C'' \vdash^i_X (\forall\alpha_2.C_2 \Rightarrow \alpha_2) \preceq t_1 \\ C'' \models_X \exists a_1.C_1 & C'' \models_X \exists a_2.C_2 \end{array} \quad (A\,1)$$

for some constraints C_1, C_2 and type variables α_1, α_2. We can assume that the set of freshly generated type variables in $\Gamma', e_1 \vdash_W (C_1 \mid \alpha_1)$ and $\Gamma', e_2 \vdash_W (C_2 \mid \alpha_2)$ are disjoint.

Application of the (TIApp) rule yields

$$\Gamma', e_1\ e_2 \vdash_W (\exists\alpha_1,\alpha_2.(C_1 \wedge C_2 \wedge \alpha_1 = \alpha_2 \to \alpha_3) \mid \alpha_3)$$

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

where α_3 is a fresh type variable.

From A 1, we an conclude that

$$C'' \models_X \exists a_1.(C_1 \wedge a_1 = t_1 \rightarrow t_2) \quad C'' \models_X \exists a_2.(C_2 \wedge a_2 = t_1) \tag{A 2}$$

which yields

$$C'' \vdash^i_X (\forall \alpha_3.\exists \alpha_1, \alpha_2.(C_1 \wedge C_2 \wedge \alpha_1 = \alpha_2 \rightarrow \alpha_3) \Rightarrow \alpha_3) \preceq t_2$$

From A 2, we can conclude that

$$C'' \models_X \exists a_1, a_2.C_1 \wedge C_2 \wedge a_1 = t_1 \rightarrow t_2 \wedge a_2 = t_1$$

Recall that a_1 does not appear in C_2 and a_2 does not appear in C_1. The above implies

$$C'' \models_X \exists a_1, a_2, a_3.C_1 \wedge C_2 \wedge a_1 = a_1 \rightarrow a_3 \wedge a_2 = t_1 \wedge a_3 = t_2$$

and via weakening we obtain

$$C'' \models_X \exists a_1, a_2, a_3.C_1 \wedge C_2 \wedge a_1 = a_1 \rightarrow a_3$$

Thus, we are done.

Case (HM∀ Elim) We have the following situation:

$$\frac{C, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow t \qquad C \models_X [\bar{t}/\bar{\alpha}]D}{C \wedge D, \Gamma \vdash e : [\bar{t}/\bar{\alpha}]t}$$

Application of the induction hypothesis yields

$$\begin{array}{c} \Gamma', e \vdash_W (C' \mid \alpha) \\ C'' \vdash^i_X (\forall \alpha.C' \Rightarrow \alpha) \preceq (\forall \bar{\alpha}.D \Rightarrow t) \\ C'' \models_X \exists a.C' \end{array}$$

for some constraint C' and type variable α. It immediately follows that

$$C'' \vdash^i_X (\forall \alpha.C' \Rightarrow \alpha) \preceq [\bar{t}/\bar{\alpha}]t$$

which establishes the induction step.

Case (HM∀ Intro) We have the following situation:

$$\frac{C \wedge D, \Gamma \vdash e : t \qquad \bar{\alpha} \notin fv(C) \cup fv(\Gamma)}{C \wedge \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}.D \Rightarrow t}$$

W.l.o.g. $\bar{\alpha} \notin fv(\Gamma', C'')$. We have that $C'' \wedge D \models_X C \wedge D$. Application of the induction hypothesis yields

$$\begin{array}{c} \Gamma', e \vdash_W (C' \mid \alpha) \\ C'' \wedge D \vdash^i_X (\forall \alpha.C' \Rightarrow \alpha) \preceq t \\ C'' \wedge D \models_X \exists a.C' \end{array}$$

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

We can conclude that

$$C'' \vdash_X^i (\forall \alpha.C' \Rightarrow \alpha) \preceq (\forall \bar{\alpha}.D \Rightarrow t)$$

and $C'' \wedge \exists \bar{a}.D \models_X \exists a.C'$ (we existentially quantify over $\bar{a}$ on both sides, note that $\bar{a}$ do not appear in C'), which establishes the induction step.

Case (HMLet) We have the following situation:

$$\frac{C, \Gamma \vdash e : \sigma \qquad C, \Gamma{+}{+}[f : \sigma] \vdash e' : t'}{C, \Gamma \vdash \mathsf{let}\ f = e\ \mathsf{in}\ e' : t'}$$

We apply the induction hypothesis to the left premise and obtain

$$\begin{gathered} \Gamma', e \vdash_W (C_1 \mid \alpha_1) \\ C'' \vdash_X^i (\forall \alpha_1.C_1 \Rightarrow \alpha_1) \preceq \sigma \\ C'' \models_X \exists a_1.C_1 \quad (1) \end{gathered}$$

for some constraint C_1 and type variable α_1. We conclude that

$$C'' \vdash_X^i \Gamma'{+}{+}[f : (\forall \alpha_1.C_1 \Rightarrow \alpha_1)] \preceq \Gamma{+}{+}[f : \sigma]$$

Thus, we are in the position to apply the induction hypothesis to the right premise which yields

$$\begin{gathered} \Gamma'_x{+}{+}[f : (\forall \alpha_1.C_1 \Rightarrow \alpha_1)], e' \vdash_W (C_2 \mid \alpha_2) \\ C'' \vdash_X^i (\forall \alpha_2.C_2 \Rightarrow \alpha_2) \preceq t' \quad (2) \\ C'' \models_X \exists a_2.C_2 \end{gathered}$$

for some constraint C_2 and type variable α_2. Application of rule (TILet) yields

$$\Gamma', \mathsf{let}\ f = e\ \mathsf{in}\ e' \vdash_W ((\exists \alpha_1.C_1) \wedge C_2 \mid \alpha_2)$$

We have to show that

$$C'' \vdash_X^i (\forall \alpha_2.((\exists \alpha_1.C_1) \wedge C_2) \Rightarrow \alpha_2) \preceq t'$$

The above is equivalent to $C'' \models_X \exists a_2.(\exists \alpha_1.C_1) \wedge C_2 \wedge a_2 = t$. Note that a_2 does not appear in C_1 and a_1 does not appear in C_2. Hence, it is sufficient to show that $C'' \models_X \exists a_1.C_1$ and $C'' \models_X \exists a_2.C_2 \wedge a_2 = t$. The first statement follows from (1) and the second statement follows from (2). Thus, we are done. □

APPENDIX B

Soundness and completeness of CLP(X)-style type inference

First, we verify soundness. In preparation, we slightly generalize the $\sim$ relation among CLP(X) rules P_E, environments E, Γ, and Γ_λ. We assume that

$$Termfv([x_1 : \sigma_1, ..., x_n : \sigma_n]) = \{x_1, ..., x_n\}$$

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

We define $P_E, E \sim \Gamma, \Gamma_\lambda$ iff

1. Γ_λ only consists of simple types, $Termfv(\Gamma) = E$, $fv(\Gamma) \subseteq fv(\Gamma_\lambda)$,
2. For each $(f : \forall \bar{a}.D \Rightarrow t) \in \Gamma$ we have that $\bar{a} = fv(D,t) - fv(\Gamma_\lambda)$ and $f(t,l) \wedge l = [t_1,...,t_n] \rightarrowtail^*_{P_E} D'$ where $\Gamma_\lambda = [x_1 : t_1,...,x_n : t_n]$ and $\models_X \bar{\exists}_{fv(\Gamma_\lambda,t)}.D \leftrightarrow \bar{\exists}_{fv(\Gamma_\lambda,t)}.D'$.

The second item states that we can compute the types in Γ by running the CLP(X) program P_E on the goal $f(t,l) \wedge l = [t_1,...,t_n]$. In the result D', we may have references to irrelevant type variables, which we can project away as stated by $\models_X \bar{\exists}_{fv(\Gamma_\lambda,t)}.D \leftrightarrow \bar{\exists}_{fv(\Gamma_\lambda,t)}.D'$. Implicitly, we make use of Theorem 1, which ensures that the logical meaning of the resulting constraint D' is equivalent to $f(t,l) \wedge l = [t_1,...,t_n]$ with respect to P_E.

In the upcoming soundness proof we make use of the following Weakening Lemma which is Lemma 13 in Sulzmann (2000).

Lemma 2 (Weakening) Let $C, \Gamma \vdash e : \sigma$ such that $C' \vdash^i_X \sigma \preceq \sigma'$ and $C' \models_X C$. Then, $C', \Gamma \vdash e : \sigma'$.

The above lemma says that expression e is still derivable under a stronger constraint but weaker type.

We verify soundness of the CLP(X)-style type inference scheme.

Theorem 4 (Soundness of CLP(X)-style type inference)
Let $P_E, E \sim \Gamma, \Gamma_\lambda$ and $E, \Gamma_\lambda, e \vdash_{Cons} (G \mid t)$ and $E, \Gamma_\lambda, e \vdash_{Def} P$ such that $G \rightarrowtail^*_{P \cup P_E} D$. Then, $D, \Gamma{+}{+}\Gamma_\lambda \vdash e : t$.

Proof
The proof proceeds by structural induction over e. We only show some of the more interesting cases.

Case (CGVar-f) and (RGVar): We have that

$$\frac{f \in E \quad t, l \text{ fresh}}{E, [x_1 : t_1, \ldots, x_n : t_n], f \vdash_{Cons} (f(t,l) \wedge l = [t_1, \ldots, t_n] \mid t)}$$

$$E, \Gamma, f \vdash_{Def} \emptyset$$

By assumption, $(f : \forall \bar{a}.D \Rightarrow t) \in \Gamma$ we have that $\bar{a} = fv(D,t) - fv(\Gamma_\lambda)$ and

$$f(t,l) \wedge l = [t_1,...,t_n] \rightarrowtail^*_{P_E} D'$$

where $\models_X \bar{\exists}_{fv(\Gamma_\lambda,t)}.D \leftrightarrow \bar{\exists}_{fv(\Gamma_\lambda,t)}.D'$ (1). Hence,

$$D, \Gamma{+}{+}\Gamma_\lambda \vdash f : t$$

by application of typing rules (HMVar) and (HM$\forall$ Elim). Another (HM$\exists$ Intro) application step leads to

$$\bar{\exists}_{fv(\Gamma_\lambda,t)}.D, \Gamma{+}{+}\Gamma_\lambda \vdash f : t$$

From (1) and Lemma 2, we can conclude that

$$\bar{\exists}_{fv(\Gamma_\lambda,t)}.D', \Gamma{+}{+}\Gamma_\lambda \vdash f : t$$

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

Recall that $C \models_X \exists a.C$ for any constraint C and variable a. Hence, by another application of the Lemma 2, we find that

$$D', \Gamma{+}{+}\Gamma_\lambda \vdash f : t$$

and we are done.

Case (CGAbs) and (RGAbs): We have that

$$\frac{E, \Gamma_\lambda{+}{+}[x : t_1], e \vdash_{Cons} (G \mid t_2) \quad t_1 \text{ fresh}}{E, \Gamma_\lambda, (\lambda x.e) \vdash_{Cons} (G \mid t_1 \to t_2)}$$

$$\frac{E, \Gamma_\lambda{+}{+}[x : t_1], e \vdash_{Def} P \quad t_1 \text{ fresh}}{E, \Gamma_\lambda, (\lambda x.e) \vdash_{Def} P}$$

W.l.o.g. we can assume that both rules share the same fresh type variable t_1. By assumption $G \rightarrowtail^*_{P_E \cup P} D$. Application of the induction hypothesis to e yields

$$D, \Gamma{+}{+}\Gamma_\lambda{+}{+}[x : t_1] \vdash e : t_2$$

We apply the typing rule (HMAbs) and find that

$$D, \Gamma{+}{+}\Gamma_\lambda \vdash \lambda x.e : t_1 \to t_2$$

and we are done.

Case (CGApp) and (RGApp):

$$\frac{E, \Gamma_\lambda, e_1 \vdash_{Cons} (G_1 \mid t_1) \quad E, \Gamma_\lambda, e_2 \vdash_{Cons} (G_2 \mid t_2) \quad t \text{ fresh}}{E, \Gamma_\lambda, e_1\ e_2 \vdash_{Cons} (G_1 \wedge G_2 \wedge t_1 = t_2 \to t \mid t)}$$

$$\frac{E, \Gamma_\lambda, e_1 \vdash_{Def} P_1 \quad E, \Gamma_\lambda, e_2 \vdash_{Def} P_2}{E, \Gamma_\lambda, e_1\ e_2 \vdash_{Def} P_1 \cup P_2}$$

By assumption $G_1 \wedge G_2 \wedge t_1 = t_2 \to t \rightarrowtail^*_{P_E \cup P_1 \cup P_2} D$. Function symbols in goal G_1 only appear in $P_E \cup P_1$ and function symbols in goal G_2 only appear in $P_E \cup P_2$. Hence, we can conclude that $G_1 \rightarrowtail^*_{P_E \cup P_1} D_1$ (1) and $G_2 \rightarrowtail^*_{P_E \cup P_2} D_2$ (2) for some D_1 and D_2 such that $D \models_X D_1 \wedge D_2 \wedge t_1 = t_2 \to t$ (3).

On the basis of (1) and (2), we can apply the induction hypothesis to the left and right premise, which yields

$$D_1, \Gamma{+}{+}\Gamma_\lambda \vdash e_1 : t_1$$
$$D_2, \Gamma{+}{+}\Gamma_\lambda \vdash e_2 : t_2$$

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

From (3) and the Weakening Lemma, we conclude that

$$D, \Gamma ++ \Gamma_\lambda \vdash e_1 : t_1$$
$$D, \Gamma ++ \Gamma_\lambda \vdash e_2 : t_2$$

From (3) and application of rule (HMEq), we conclude that

$$D, \Gamma ++ \Gamma_\lambda \vdash e_1 : t_2 \to t$$

We are in the position to apply rule (HMApp), which leads to

$$D, \Gamma ++ \Gamma_\lambda \vdash e_1\ e_2 : t$$

and we are done.

Case (CGLet) and (RGLet): We have that

$$\frac{E \cup \{f\}, \Gamma_\lambda, e_2 \vdash_{Cons} (G \mid t) \quad \Gamma_\lambda = [x_1 : t_1, \ldots, x_n : t_n] \quad a,l \text{ fresh}}{E, \Gamma_\lambda, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Cons} (G \wedge f(a,l) \wedge l = [t_1, ..., t_n] \mid t)}$$

$$\frac{E, \Gamma_\lambda, e_1 \vdash_{Cons} (G' \mid t') \quad \Gamma_\lambda = [x_1 : t_1, \ldots, x_n : t_n] \quad l,r \text{ fresh} \quad E, \Gamma_\lambda, e_1 \vdash_{Def} P_1 \quad E \cup \{f\}, \Gamma, e_2 \vdash_{Def} P_2 \quad P = P_1 \cup P_2 \cup \{f(t', l)\ \text{:-}\ G' \wedge l = t_1 : ... : t_n : r\}}{E, \Gamma_\lambda, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Def} P}$$

By assumption we find that $G \rightarrowtail^*_{P_E \cup P} D$. Because of the (anonymous) call to f (we refer here to the constraint $f(a,l) \wedge l = [t_1, ..., t_n]$) there exists a subderivation $G' \rightarrowtail^*_{P_E \cup P_1} D'$ (1) where $D \models_X \bar{\exists}_{fv(\Gamma_\lambda)}.D'$ (2).

On the basis of (1), we can apply the induction hypothesis to e_1, which yields

$$D', \Gamma ++ \Gamma_\lambda \vdash e_1 : t'$$

Then, we apply the typing rule (HM$\exists$ Intro) and obtain

$$\bar{\exists}_{fv(\Gamma_\lambda, t')}.D', \Gamma ++ \Gamma_\lambda \vdash e_1 : t'$$

Next, we apply typing rule (HM$\forall$ Intro) and find

$$\exists \bar{a}.\bar{\exists}_{fv(\Gamma_\lambda, t')}.D', \Gamma ++ \Gamma_\lambda \vdash e_1 : \forall \bar{a}.\bar{\exists}_{fv(\Gamma_\lambda, t')}.D' \Rightarrow t' \quad (3)$$

where $\bar{a} = fv(D', t') - fv(\Gamma_\lambda)$.

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

We set

$$P_{E\cup\{f\}} = P_E \cup \{f(t,l) \text{ :- } G \wedge l = t_1 : \ldots : t_n : r\} \cup P_1$$

and immediately find that

$$P_{E\cup\{f\}}, E \cup \{f\} \sim \Gamma{+}{+}[f : \forall fv(D',t') - fv(\Gamma_\lambda).D' \Rightarrow t'], \Gamma_\lambda$$

We are in the position to apply the induction hypothesis to e_2 and obtain that

$$D, \Gamma{+}{+}\Gamma_\lambda{+}{+}[f : \forall fv(D',t') - fv(\Gamma_\lambda).D' \Rightarrow t'] \vdash e_2 : t \quad (4)$$

W.l.o.g., we assume that t' is a fresh variable. Then, from (2), we can conclude that

$$D \models_X \exists \bar{a}.\bar{\exists}_{fv(\lambda_\lambda,t')}.D' \quad (5)$$

From (3), (5), and the Weakening Lemma, we obtain that

$$D, \Gamma{+}{+}\Gamma_\lambda \vdash e_1 : \forall \bar{a}.\bar{\exists}_{fv(\Gamma_\lambda,t')}.D' \Rightarrow t'$$

Together with (4), we conclude by application of typing rule (HMLet) that

$$D, \Gamma{+}{+}\Gamma_\lambda \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t$$

and we are done. □

Next, we consider completeness. For convenience, we will make use of a slightly different formulation of rule (HMLet) from Figure 1. We combine the rule for quantifier introduction with the rule for let statements.

$$\text{(HMLet')}\quad \frac{\begin{array}{c} D'', \Gamma'' \vdash e_1 : t'' \\ \bar{a} = fv(D'',t'') - fv(\Gamma'') \quad \sigma = \forall \bar{a}.D'' \Rightarrow t'' \\ D', \Gamma''{+}{+}[f : \sigma] \vdash e_2 : t' \end{array}}{(\exists \bar{a}.D'') \wedge D', \Gamma'' \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t'}$$

In essence, the above rule corresponds to the inference rule (TILet) from Figure 2. It should be clear that we can replace rules (HM∀ Intro) and (HMLet) by (HMLet') without changing the set of typable programs.

The completeness result follows from the following lemma. As in case of Lemma 1, we provide a slightly stronger statement than necessary so that the induction will go through.

Lemma 3 Let $P_E, E \sim \Gamma', \Gamma_\lambda$ and $fv(\Gamma) \subseteq fv(\Gamma_\lambda)$ and $\vdash \Gamma' \preceq \Gamma$ and $D', \Gamma{+}{+}\Gamma_\lambda \vdash e : t'$. Then, $E, \Gamma_\lambda, e \vdash_{Cons} (G \mid t)$ and $E, \Gamma_\lambda, e \vdash_{Def} P$ for some goal G, type t and CLP(X) program P such that $D' \models_X \bar{\exists}_{fv(\Gamma_\lambda,t')}.D \wedge t = t'$ where $G \rightarrowtail^*_{P_E \cup P} D$.

Proof

The proof proceeds by structural induction. We only show the case for let-defined functions.

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

Case (HMLet'): We have that

$$\frac{D'', \Gamma{+}{+}\Gamma_\lambda \vdash e_1 : t'' \quad \bar{a} = fv(D'', t'') - fv(\Gamma_\lambda) \quad \sigma = \forall \bar{a}.D'' \Rightarrow t'' \quad D', \Gamma{+}{+}\Gamma_\lambda{+}{+}[f : \sigma] \vdash e_2 : t'}{(\exists \bar{a}.D'') \wedge D', \Gamma{+}{+}\Gamma_\lambda \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t'}$$

where we assume that $\Gamma_\lambda = [x_1 : t_1, ..., x_n : t_n]$.

Application of the induction hypothesis to the left premise yields

$$E, \Gamma_\lambda, e_1 \vdash_{Cons} (G_1 \mid t_1') \quad E, \Gamma_\lambda, e_1 \vdash_{Def} P_1$$

such that $D'' \models_X \bar{\exists}_{fv(\Gamma_\lambda, t'')}.D_1 \wedge t'' = t_1'$ (1) where $G_1 \rightarrowtail^*_{P_E \cup P_1} D_1$ (2). We can conclude that

$$\vdash^i_X (\forall fv(D_1, t_1') - fv(\Gamma_\lambda).D_1 \Rightarrow t_1') \preceq (\forall fv(D'', t'') - fv(\Gamma_\lambda).D'' \Rightarrow t'')$$

We have that

$$P_{E \cup \{f\}}, E \cup \{f\} \sim \Gamma'{+}{+}[f : \forall fv(D_1, t_1') - fv(\Gamma_\lambda).D_1 \Rightarrow t_1'], \Gamma_\lambda$$

where $P_{E \cup \{f\}} = P_E \cup \{f(t_1', l)\ \text{:-}\ G_1 \wedge l = t_1 : ... : t_n : r\} \cup P_1$. Notice that $P_{E \cup \{f\}}$ includes P_1, hence, G_1 will be reduced to D_1.

We can then apply the induction hypothesis to e_2, which yields

$$E \cup \{f\}, \Gamma_\lambda, e_2 \vdash_{Cons} (G \mid t) \quad E \cup \{f\}, \Gamma_\lambda, e_2 \vdash_{Def} P_2$$

such that $D' \models_X \bar{\exists}_{fv(\Gamma_\lambda, t')}.D \wedge t' = t$ (3) where $G \rightarrowtail^*_{P_{E \cup \{f\}} \cup P_2} D$ (4).

Application of the rules (CGLet) and (RGLet) yields

$$\frac{E, \Gamma_\lambda, e_2 \vdash_{Cons} (G \wedge f(a, l) \wedge l = [t_1, ..., t_n] \mid t)}{E, \Gamma_\lambda, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Def} P}$$

where $P = P_1 \cup P_2 \cup \{f(t_1', l)\ \text{:-}\ G_1 \wedge l = t_1 : ... : t_n : r\}$.

We yet need to verify that $G \wedge f(a, l) \wedge l = [t_1, ..., t_n] \mid t \rightarrowtail^*_{P_E \cup P} D'''$ for some D''' such that $(\exists \bar{a}.D'') \wedge D' \models_X \bar{\exists}_{fv(\Gamma_\lambda, t')}.D''' \wedge t' = t$. From (2), we can conclude that

$$\begin{array}{ll} & f(a, l) \wedge l = [t_1, ..., t_n] \\ \rightarrowtail_{P_E \cup P} & [a/t_1']G_1 \wedge l = t_1 : ... : t_n : r \wedge l = [t_1, ..., t_n] \\ \rightarrowtail^*_{P_E \cup P} & [a/t_1']D_1 \wedge l = t_1 : ... : t_n : r \wedge l = [t_1, ..., t_n] \end{array}$$

and therefore from (4) we can conclude that

$$G \wedge f(a, l) \wedge l = [t_1, ..., t_n] \mid t \rightarrowtail^*_{P_E \cup P} D \wedge [a/t_1']D_1 \wedge l = t_1 : ... : t_n : r \wedge l = [t_1, ..., t_n]$$

From (1) and (3), we can conclude that $\exists \bar{a}.D'' \models_X \bar{\exists}_{fv(\Gamma_\lambda, t')}.D_1$ and $D' \models_X \bar{\exists}_{fv(\Gamma_\lambda, t')}.D \wedge t' = t$. Constraints D and D_1 only share variables in $fv(\Gamma_\lambda, t')$. Hence, we can conclude that $(\exists \bar{a}.D'') \wedge D' \models_X \bar{\exists}_{fv(\Gamma_\lambda, t')}.D_1 \wedge D \wedge t' = t$. Variable t_1' does not appear in $fv(\Gamma_\lambda, t')$ and $\exists l, r.l = t_1 : ... : t_n : r \wedge l = [t_1, ..., t_n]$ is a true statement. Hence, we can conclude that

$$(\exists \bar{a}.D'') \wedge D' \models_X \bar{\exists}_{fv(\Gamma_\lambda, t')}.D \wedge [a/t_1']D_1 \wedge l = t_1 : ... : t_n : r \wedge l = [t_1, ..., t_n]$$

and we are done. □

https://doi.org/10.1017/S0956796807006569 Published online by Cambridge University Press

References

Aiken, A. & Wimmers, E. L. (1992) Solving systems of set constraints. In *Seventh IEEE Symposium on Logic in Computer Science, Santa Cruz, CA*. Los Alamitos, CA: IEEE Computer Society Press, pp. 320–340.

Alves, S. & Florido, M. (2002) Type inference using constraint handling rules. *Electr. Notes Theor. Comput. Sci.* **64**.

Birkedal, L. & Tofte, M. (2001) A constraint-based region inference algorithm. *Theor. Comput. Sci.* **258**(1–2), 299–392.

Coquery, E. & Fages, F. (2002) TCLP: Overloading, subtyping and parametric polymorphism made practical for CLP. In *Proc. of ICLP '02*, vol. 2401. Berlin: Springer-Verlag.

Damas, L. & Milner, R. (1982) Principal type-schemes for functional programs. In *Proc. of POPL'82*. New York: ACM Press, pp. 207–212.

Demoen, B., García de la Banda, M. & Stuckey, P. J. (1999) Type constraint solving for parametric and ad-hoc polymorphism. In *Proc. of the 22nd Australian Computer Science Conference*. Berlin: Springer-Verlag, pp. 217–228.

Dietzen, S. & Pfenning, F. (1991) A declarative alternative to "assert" in logic programming. In *Proc. of ISLP'91*, pp. 372–386.

Eo, H., Lee, O. & Yi, K. (2003) Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Generation Comput.* **22**(1), 1–36.

Fordan, A. & Yap, R. H. C. (1998) Early projection in CLP(R). In *CP '98: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*. London, UK: Springer-Verlag, pp. 177–191.

Frühwirth, T. (1995) Constraint handling rules. In *Constraint Programming: Basics and Trends*. LNCS. Berlin: Springer-Verlag, pp. 90–107.

Fuh, Y.-C. & Mishra, P. (1990) Type inference with subtypes. *Theor. Comput. Sci.* **73**, 155–175.

Glynn, K., Stuckey, P. J. & Sulzmann, M. (2000) Type classes and constraint handling rules. In *Workshop on Rule-Based Constraint Reasoning and Programming*. Available at: http://xxx.lanl.gov/abs/cs.PL/0006034. Accessed August 2007.

Henglein, F. (1992) *Simple Closure Analysis*. DIKU Semantics Report D-193. University of Copenhagen.

Henglein, F. (1993) Type inference with polymorphic recursion. *Trans. Programming Lang Syst.* **15**(1), 253–289.

Henkin, L., Monk, J. D. & Tarski, A. (1971) *Cylindric Algebra*. Amsterdam: North-Holland Publishing Company.

Jaffar, J. & Lassez, J-L. (1987) Constraint logic programming. In *Proc. of POPL'87*, pp. 111–119.

Jaffar, J., Maher, M., Marriott, K. & Stuckey, P.J. (1998) The semantics of constraint logic programs. *J. Logic Programming* **37**(1–3), 1–46.

Kanellakis, P. C., Mairson, H. G. & Mitchell, J. C. (1991) Unification and ML-type reconstruction. In *Computational logic - Essays in Honor of Alan Robinson*. Cambridge, Mass.: MIT Press, pp. 444–478.

Kennedy, A. J. (1996) *Type Inference and Equational Theories*. Tech. rept. LIX/RR/96/09. LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France.

Lakshman, T. L. & Reddy, U/ S. (1991) Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Proc. of ISLP'91*. Cambridge, Mass.: MIT Press, pp. 202–217.

Lee, O. & Yi, K. (1998) Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Programming Lang. Syst.*, **20**(4), 707–723.

Liang, C. (1997) Let-polymorphism and eager type schemes. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*. Springer-Verlag, pp. 490–501.

Maher, M. (1988) Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proc. 3rd Logic in Computer Science Conference*, pp. 348–357.

McAllester, D. A. (2003) Joint RTA-TLCA invited talk: A logical algorithm for ML type inference. In *Proc. of RTA'03*. LNCS, Vol. 2706. Berlin: Springer-Verlag, pp. 436–451.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **17**, 348–375.

Mitchell, J. (2002) *Concepts of Programming Languages*. Cambridge University Press.

Müller, M. (1994) A constraint-based recast of ML-polymorphism. In *8th International Workshop on Unification*. Also available as Technical Report 94-R-43, Université de Nancy.

Mycroft, A. & O'Keefe, R. (1984) A polymorphic type system for Prolog. *Artif. Intelligence* **23**, 295–307.

Nadathur, G. & Miller, D. (1988) An overview of λprolog. In *Fifth International Conference and Symposium on Logic Programming*, *Bowen, K. & Kowalski, R. (eds)*. MIT Press.

Odersky, M., Sulzmann, M. & Wehr, M. (1999) Type inference with constrained types. *Theory Pract. Object Syst.*, **5**(1), 35–55.

Palsberg, J. & Smith, S. (1996) Constrained types and their expressiveness. *ACM Trans. Programming Lang. Syst.* **18**(5), 519–527.

Peyton Jones, S. (ed). (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Pottier, F. (1998) A framework for type inference with subtyping. In *Proc. of ICFP'98*. ACM Press, pp. 228–238.

Pottier, F. & Rémy, D. (2005) The essence of ML type inference. *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed). Cambridge, Mass.: MIT Press, Chap. 10, pp. 389–489.

Rémy, D. (1993) Type inference for records in a natural extension of ML. In *Theoretical Aspects of Object-Oriented Programming. Types, Semantics and Language Design*, Chapter 3. Gunter, C. A. & Mitchell, J. C. (eds), MIT Press.

Robinson, J. A. (1965) A machine-oriented logic based on the resolution principle. *J. ACM* **12**, 23–41.

Stuckey, P. J. & Sulzmann, M. (2005) A theory of overloading. *ACM Trans. Programming Lang. syst. (TOPLAS)* **27**(6), 1–54.

Stuckey, P. J., Sulzmann, M. & Wazny, J. (2003a) The Chameleon type debugger. In *Proc. of Fifth International Workshop on Automated Debugging (AADEBUG 2003)*. Computer Research Repository. Available at: http://www.acm.org/corr/.

Stuckey, P. J., Sulzmann, M. & Wazny, J. (2003b) Interactive type debugging in Haskell. In *Proc. of Haskell'03*. New York: ACM Press, pp. 72–83.

Stuckey, P. J., Sulzmann, M. & Wazny, J. (2004) Improving type error diagnosis. In *Proc. of Haskell'04*. New York: ACM Press, pp. 80–91.

Stuckey, P. J., Sulzmann, M. & Wazny, J. (2006) Type processing by constraint reasoning. *Proc. of APLAS'06*. LNCS, Vol. 4279, Berlin: Springer-Verlag, pp. 1–25.

Sulzmann, M. (2000) *A General Framework for Hindley/Milner Type Systems With Constraints*. Ph.D. thesis, Department of Computer Science, Yale University.

Sulzmann, M. (2001) A general type inference framework for Hindley/Milner style systems. In *Proc. of FLOPS'01*. LNCS, Vol. 2024. Berlin: Springer-Verlag, pp. 246–263.

Sulzmann, M., Müller, M. & Zenger, C. (1999) *Hindley/Milner Style Type Systems in Constraint Form.* Research Report ACRC-99-009. University of South Australia, School of Computer and Information Science.

Sulzmann, M., Odersky, M. & Wehr, M. (1997) Type inference with constrained types. In *FOOL4: 4th Int. Workshop on Foundations of Object-Oriented Programming Languages.*

Sulzmann, M. & Wazny, J. (2007) *Chameleon.* Available at http://www.comp.nus.edu.sg/~sulzmann/chameleon. Accessed August 2007.

Urban, C., Pitts, A. M. & Gabbay, M. J. (2004) Nominal unification. *Theor. Comput. Sci.* **323**(1-3), 473–497.

Wadler, P. & Blott, S. (1989) How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*. New York: ACM Press, pp. 60–76.

Zenger, C. (1999) *Indizierte Typen.* Ph.D. thesis, Universität Karlsruhe, Germany.