A Type Inference System for a Concurrent Logic Language

1M−8

Dongwook Shin
IIAS-SIS, Fujitsu Limited

Abstract

This paper proposes a type inference system for Guarded Horn Clauses, GHC, based on the notion of value and communication type. Value type is a type that a predicate can have, guaranteeing that a goal predicate of the value type does not raise type errors at run time. Communication type is a type under which two predicates communicate each other. These types are obtained by constraint solving and partial evaluation of a GHC program to some extent. We show that these types contribute to the early detection of errors in the software development.

1 Introduction

A type inference system is a system which detects the types of a given program without type information. It has been intensively studied for functional languages [Milner 78] and as a result, conceived as a useful software component which contributes to the early detection of programming errors.

During the last several years, type inference has also been studied in logic programming, in particular, Prolog [Mishra 84, Kanamori 85 and Zobel 87]. However there has been no attempt to infer types in concurrent logic languages.

This paper develops a type inference system of GHC which infers the type of each predicate according to the given partial type information. Two kinds of types are introduced here: value type and communication type. Value type is a type that a predicate can have, guaranteeing that a goal predicate of the type does not raise type errors at run time. Communication type is a type under which two predicates communicate each other. These types are obtained by constraint solving and partial evaluation of a GHC program to some extent. We show that these types contribute to early detection of errors in the software development.

2 Types

This section defines value type and communication type of GHC. First, this paper supports parametric polymorphism in value type. Thus, a type is able to involve a type variable. However, overloading is excluded, mainly because it makes the type inference much harder. For instance, *int* is a type, and $list(\alpha)$ is also a type, if α is a type variable and *list* is a type constructor.

Based on this, the value types of functions and predicates can be declared. If some of them are not declared, they are inferred by the methods in the following section. In particular, if the type of a predicate is not declared, it is, first, assumed as:

> pred p : $@p/n(1), \cdots, @p/n(n)$, where $@p/n(i)$ is a type variable symbol denoting the unknown type of the i-th argument of p/n

and the inference algorithm finds the appropriate type for each $@p/n(i)$.

In addition, three kinds of communication types between predicates are defined: one-shot communication (OS for short), stream-based lazy communication (SL for short) and stream-based eager communication (SE for short). OS type is a communication type taking place between two predicates (or processes) when a predicate gives a value to the other only once via a shared variable. SL type is a type indicating a communication pattern that a predicate sends a piece of information continuously via a stream, typically list, only when the other predicate is ready to process it. SE type is a type indicating a communication pattern that a predicate sends a piece of information continuously via a stream, typically list, immediately after the information is generated.

3 Value Type Inference as Constraint Solving

A value typing for a predicate or a clause associates a value type with each variable, functor and predicate. The value type annotations need not be provided by users because most general type annotations can be computed by the typing rules in [Hanus 89]. From the typing rule in [Hanus 89], we can get the *Most General Typed Clause* (*MGTC*) of each clause. From these *MGTC*s, the type of the predicate p/n is computed by Algorithm 1 with a set of constraint $\Psi_{p/n}$

Algorithm 1. type constraint algorithm for the predicate p/n

Input : typed clauses whose heads are p/n

Output : a set of equations for the types of p/n with its type constraint $\Psi_{p/n}$

1. For each $@p/n(i)$ which denotes the i-th argument type of p/n,

   $@p/n(i) = \text{unify}(\tau_i^1, \cdots, \tau_i^k)$, where τ_i^j is the type of the i-th argument of the j-th clause whose head is p/n.

2. For every typed predicate $q(s_1{:}\sigma_1, \cdots, s_k{:}\sigma_k)$, in the body part in an input typed clause, add a type constraint $(\sigma_1, \cdots, \sigma_k) = \sigma_{q/k}(@q/k(1), \cdots, @q/k(k))$ to $\Psi_{p/n}$.

For instance, let us consider the append program in Example 1 including preassumed types for list and equal predicate.

Example 1.
func []: $\rightarrow$ $list(\alpha)$
func [..|..]: α, $list(\alpha) \rightarrow list(\alpha)$
pred =: β, β
append([],X,Y) :- true | X = Y
append([H|T1],X,Y) :- true | Y = [H|T2],
append(T1,X,T2)

First, from these preassumed types, we get the following *MGTC*s;

$append([]{:}list(\alpha_1), X{:}\alpha_2, Y{:}\alpha_2)$:- true | $X{:}\alpha_2 = Y{:}\alpha_2$.

$append([H{:}\alpha_3 | T1{:}list(\alpha_3)]{:}list(\alpha_3), X{:}\alpha_4, Y{:}list(\alpha_3))$:- true | $Y{:}list(\alpha_3) = [H{:}\alpha_3 | T2{:}list(\alpha_3)]{:}list(\alpha_3)$, $append(T1{:}list(\alpha_3), X{:}\alpha_4, T2{:}list(\alpha_3))$.

Secondly, by Algorithm 1, the value type of append is computed as:

$@append/3(1) = \text{unify}(list(\alpha_1), list(\alpha_3))$,
$@append/3(2) = \text{unify}(\alpha_2, \alpha_4)$,
$@append/3(3) = \text{unify}(\alpha_2, list(\alpha_3))$,

with a constraint,

$\Psi_{p/n} = \{ (list(\alpha_3), \alpha_4, list(\alpha_3)) = \sigma(@append/3(1), @append/3(2), @append/3(3))\}$.

If a program does not have indirect recursion, the equations obtained by Algorithms 1 can be solved. For instance, the equations for *append* are solved with a type $(list(\alpha_1), list(\alpha_1), list(\alpha_1))$.

4 Communication Type

Communication types are inferred using the value types inferred in Section 3. If a variable is shared between two predicates and its type is a primitive type, then the communication type between them is OS. SL and SE could take place if there is a shared variable whose type is typically a list. The distinction between SE and SL are found by partial evaluation of a GHC program to some extent. We will not discuss this topic in detail owing to the limited space.

5 Conclusion

This paper proposes a type inference system for a concurrent logic language, GHC. Every type of a predicate or a functor is not necessarily predefined. Instead, the type system infers the undeclared types. Hence, it leaves out the burden that a programmer should declare the types and at the same time, it contributes to the early detection of errors in the software development.

References

[Hanus 89] M. Hanus, Polymorphic Higher-Order Programming in Prolog, Proceedings of the Sixth International Conference, 1989, pp. 382-397.

[Kanamori 84] T. Kanamori and K. Horiuchi, Type Inference in Prolog and Its Applications, ICOT TR-095, 1984.

[Milner 78] R. Milner, A Theory of Type Polymorphism in Programming, in: Journal of Computer and System Science, Vol. 17, 1978, pp. 348-375.

[Mishra 84] P. Mishra, Toward a Theory of Types in Prolog in: Proceedings of IEEE International Symposium on Logic Programming, 1984, pp. 289-298.

[Zobel 87] J. Zobel, Derivation of Polymorphic Types for Prolog Programs, Proc. of Fourth International Conference on Logic Programming, 1987, pp. 817-838.