Type Constraint Solving for Parametric and Ad-hoc Polymorphism

Bart Demoen[1], María García de la Banda[2] and Peter J. Stuckey[3]

[1] Dept. of Computer Science, K.U.Leuven, Belgium
[2] Dept. of Computer Science, Monash University, Australia
[3] Dept. of Computer Science, University of Melbourne, Australia

Abstract. Unification has long been used as a mechanism for type checking and type inference for Hindley-Milner types in functional programming. The programmer defines the possible types, and the compiler uses unification to check and infer types for function definitions. In constraint logic programming it is natural to extend the functional programming case by allowing overloading of predicate and function definitions, that is, ad-hoc polymorphism. Mycroft and O'Keefe showed how to check predicate type declarations under these assumptions. In this paper, we show how to infer predicate types, by translating a constraint logic program with given types into a logic program over types. The program can then be used to check and infer the possible types for the predicates and variables appearing in the original program.

Since executing the translated program can be inefficient when there are highly disjunctive type definitions, we use methods of propagation based constraint solving and memoing to achieve efficient type inference.

1 Introduction

Unification has long been used as a mechanism for type checking and type inference of Hindley-Milner types [Milner, 1978]. For example, in functional languages the programmer is required to define the possible types, so that the compiler can use unification to check and infer types for functions. In this scheme, the type of a function is considered as a conjunction of tree equations over type constructors. This method neatly handles parametric polymorphism and higher-order types.

Mycroft and O'Keefe [Mycroft and O'Keefe, 1984] applied the same approach to (constraint) logic programs. Given type definitions and declaration of predicate types they check the predicate types are valid. Unlike the functional programming case they allow multiple types to be declared for functors and predicates. They also realised that this type checking can be expressed as a logic program. In this paper we extend this approach, allowing predicate types to be inferred rather than declared. We explicitly define a translation of a (constraint) logic program with type definitions to a logic program which can be used to infer

Proceedings of the Twenty Second Australasian Computer Science Conference, Auckland, New Zealand, January 18–21 1999. Copyright Springer-Verlag, Singapore. Permission to copy this work for personal or classroom use is granted without fee provided that: copies are not made or distributed for profit or personal advantage; and this copyright notice, the title of the publication, and its date appear. Any other use or copying of this document requires specific prior permission from Springer-Verlag.

types of predicates. We then investigate more robust approaches to executing the resulting program using constraint solving methods. Note that modern CLP languages (e.g. Mercury [Somogyi et al., 1996] and CIAO [Hermenegildo, 1994]) increasingly include facilities for types to be defined and used.

The constraint based view of typing is simply a rephrasing of the approach of Milner [Milner, 1978]: variables represent their own type (they become type variables) and each fragment of program text defines type constraints on the variables involved. For example, assuming that the (parametric) type of the *cons* function is $(T, \texttt{list}(T), \texttt{list}(T))$,[1] the equation $\texttt{[}Y\texttt{|}Ys\texttt{]} \texttt{ = } Xs$ defines the type constraint $Y = T' \wedge Ys = \texttt{list}(T') \wedge Xs = \texttt{list}(T')$ where T' is a new type variable. Each usage of *cons* inherits a new copy of this constraint: this corresponds to the *generic* types of Milner. Similarly the atom $\texttt{p(}X\texttt{,}Y\texttt{)}$ where predicate p has type $(\texttt{int}, \texttt{fn}(T, T))$ yields the constraint $X = \texttt{int} \wedge Y = \texttt{fn}(T'', T'')$. The type constraint of a conjunctive goal is obtained by conjoining the constraints arising from its literals. The type constraint of a rule is the constraint of its body, with the exception that recursive calls are constrained to have the same type as the head. This is Milner's *non-generic* type for identifiers which are not free in the body (the recursive literals). The type constraint of a predicate is obtained by conjoining the constraints defined by its rules.

We (as [Mycroft and O'Keefe, 1984]) extend Milner's framework by allowing overloading of predicates and functions, that is, ad-hoc polymorphism. In order to do this we must consider the type of a predicate or function as a disjunctive constraint formula over types, one disjunct for each ad-hoc polymorphic definition. This modification allows us to handle ad-hoc polymorphism in a straightforward way. Assume, for example, that we overload the + functor for use on both ints and floats. We can then define its type constraint as $T_+ = (\texttt{int}, \texttt{int}, \texttt{int}) \vee T_+ = (\texttt{float}, \texttt{int}, \texttt{float}) \vee T_+ = (\texttt{int}, \texttt{float}, \texttt{float}) \vee T_+ = (\texttt{float}, \texttt{float}, \texttt{float})$. Now, consider the following program (left) for summing a list of numbers, and the type constraints defined by each of its rules (right):

`sumlist(L, S) :-`	$T_{\texttt{sumlist}} = (L, S) \wedge$
`    [S] = L.`	$(S, \rho(T_{nil}), L) = \rho'(T_{cons}) \wedge$
`sumlist(L,S) :-`	
`    [X|Xs] = L,`	$(X, Xs, L) = \rho''(T_{cons}) \wedge$
`    X + S1 = S,`	$(X, S1, S) = \rho'''(T_+) \wedge$
`    sumlist(Xs, S1)`	$(Xs, S1) = T_{\texttt{sumlist}}.$

where $T_{nil} = (\texttt{list}(T))$, $T_{cons} = (T, \texttt{list}(T), \texttt{list}(T))$ and ρ, ρ', ρ'' and ρ''' are renamings which rename to new disjoint variables. Note how *generic* uses are renamed to use new variables, while the *non-generic* recursive call is replaced by an equation which equates the type of the predicate (the head variables) with the type of the recursive call. Then, if we conjoin the constraints defined by each rule, place the resulting constraint in DNF and project away variables, we obtain the (disjunctive) answer constraint: $T_{\texttt{sumlist}} = (\texttt{list}(\texttt{int}), \texttt{int}) \vee T_{\texttt{sumlist}} = (\texttt{list}(\texttt{float}), \texttt{float})$.

[1] We use a relational form of type declarations

In the terminology of type systems for logic programs, the type system we describe is a prescriptive type that adds extra information to the program. For example, the unique type for `append` is $(\texttt{list}(T), \texttt{list}(T), \texttt{list}(T))$ which disallows some of its success set. The key property of a Milner type system is that type correct programs do not go wrong (have type incorrect execution) and hence types can be completely compiled away. This continues to hold for type correct programs where types may be disjunctive.

Type checking of logic programs is performed by several systems like those described in [Mycroft and O'Keefe, 1984; Rouzaud and Nguyen-Phuong, 1992] and [Meyer, 1996]. However, none of the implemented systems infer predicate types nor deal with higher-order predicates in a complete way. Type inference is performed by several abstract interpretation frameworks: without an attempt to cover literature completely, we mention [Hentenryck et al., 1995; Gallagher and de Waal, 1994; Janssens and Bruynooghe, 1992]. However, in these frameworks the type definitions themselves are derived from the program clauses. We assume given type definitions and we check and/or infer predicate typings using only these type declarations: in this respect, our work is closer to [Codish and Demoen, 1994] which also assumes the set of possible types to be given to the analysis. The closest in spirit to our type checking/inference approach is the Mercury system [Somogyi et al., 1996]: the Mercury compiler contains a type checker that also infers types for local predicates missing a pred-declaration and it uses only the user provided types, i.e. it does not infer new type definitions. However, Mercury's implementation differs substantially from ours: it uses a fixpoint algorithm that essentially infers types bottom-up by successive approximation. This works very well in practice, but it can have performance problems for exceptional cases and also for badly typed programs. In particular, the bottom-up approach can lead to non-terminating inference. Our approach is based on a program transformation; it applies constraint technology and selective memoing: together these techniques resulted in an implementation with high performance even if the transformed program is executed under Prolog, which is several times slower than Mercury. The final difference with Mercury is that by treating recursive calls non-generically, we eliminate the need for a fixed point computation.

2 Translating a CLP program with type definitions

In the following we use Mercury syntax for defining types. A *type constructor* f is a functor of arity n. A *type expression* is a type variable v or a term of the form $f(t_1, \ldots, t_n)$ where f is an n-ary type constructor, and $t_1, \ldots, t_n$ are type expressions. A *type definition* for f is of the form

`:- type` $f(v_1, \ldots, v_n)$ `--->` $(f_1(t_{11}, \ldots, t_{1m_1}); \cdots ; f_k(t_{k1}, \ldots, t_{km_k}))$.

where $v_1, \ldots, v_n$ are distinct type variables, $f_1, \ldots f_k$ are distinct functors, and $t_{11}, \ldots, t_{km_k}$ are type expressions, involving at most the variables $v_1, \ldots, v_n$.

Mercury also allows hidden type definitions and equational type definitions which are handled by our approach but we omit these due to lack of space.

Example 1. Consider the following type definitions[2] for lists, pairs, nodes (for representing a graph, a node has an identifier and a list of adjacent node identifiers), arithmetic expressions, and functional applications.

```
:- type list(T) ---> ([] ; [T|list(T)]).
:- type pair(T) ---> (T - T).
:- type node(T) ---> (T-list(T)).
:- type expr ---> (var(pair(int)) ; expr * expr ; expr - expr ; - expr).
:- type fn(I,O) ---> (I --> O).
```

Translating type definitions: A type definition defines types for the functors appearing in the definition. The set of types for a functor defines a type constraint that applies whenever the functor is used in a program, which is encoded by the atom `func_f`, by our translation. The translation of a type definition of the form

`:- type` $f(v_1, \ldots, v_n)$ `--->` $(f_1(t_{11}, \ldots, t_{1m_1}); \cdots; f_k(t_{k1}, \ldots, t_{km_k}))$.

results in the rule `func_f`$_i(t_{i1}, \ldots, t_{im_i}, f(v_1, \ldots, v_n))$ for each functor f_i in the definition.

Functors may appear in several type definitions (contrary to the functional programming case, but also allowed by [Mycroft and O'Keefe, 1984]), thus resulting in predicates with multiple rules: one for each type definition in which they appear. Note that, to obtain legitimate Prolog programs, we replace non-alphabetic functor names by a descriptive name beginning with an underscore.

Example 2. The translation of definitions in Example 1 results in the following rules for `-` (minus):

```
func__minus(T,T,pair(T)).
func__minus(T,list(T),node(T)).
func__minus(expr,expr,expr).
func__minus(expr,expr).
```

Translating program rules: We assume programs rules are normalized and fully flattened, that is, each literal except equality has only variables as arguments, and each equation involves at most one functor occurrence. It is straightforward to convert any program to this form.

Let us first restrict our attention to programs without mutually recursive predicates. A (non-recursive) literal $p(v_1, \ldots, v_n)$ is translated to $pred\_p(v_1, \ldots, v_n)$. The equation $f(v_1, \ldots, v_n) = v$ is translated to $func\_f(v_1, \ldots, v_n, v)$. The equation $v_1 = v_2$ is translated as itself. A (directly recursive) literal $p(v_1, \ldots, v_n)$ in a rule with head $p(w_1, \ldots, w_n)$ is translated as $v_1 = w_1, \ldots, v_n = w_n$. This is the *non-generic* instance where the call must have the same type as the head. The i^{th} rule for predicate p, with head $p(w_1, \ldots, w_n)$ creates a rule with head `rule`i`_p`$(w_1, \ldots, w_n)$ and body given by the translation of the body. Finally, the constraints defined by the k rules of a predicate are conjoined by adding the rule `pred_p(V1,...,Vn) :- rule1_p(V1,...,Vn), ..., rule`k`_p(V1,...,Vn).`

[2] We write functors in infix notation where this is usual.

Example 3. The following figure illustrates the translation (right) of a program (left) which defines a simple chained lookup mechanism.

```
member(X, T1) :-
      [X|_] = T1.
member(X, T2) :-
      [_|R] = T2,
      member(X,R).

deref(V, Binds, DV) :-
      V-V1 = T3,
      member(T3, Binds),
      deref(V1, Binds, DV).
deref(V, _, T4) :-
      V = T4.
```

```
pred_member(X,L) :-
      rule1_member(X,L),
      rule2_member(X,L).
rule1_member(X, T1) :-
      func__cons(X, _, T1).
rule2_member(X, T2) :-
      func__cons(_,R,T2),
      X = X, R = T2.

pred_deref(V,B,DV) :-
      rule1_deref(V,B,DV),
      rule2_deref(V,B,DV).
rule1_deref(V, Binds, DV) :-
      func__minus(V,V1,T3),
      pred_member(T3, Binds),
      V1 = V, Binds = Binds, DV = DV.
rule2_deref(V, _, T4) :-
      V = T4.
```

The translation of mutually recursive predicates is somewhat more difficult. Milner's inference scheme requires all mutually recursive calls of the same predicate to share the same type. We achieve this by a program transformation which makes the appropriate variables available.

Let us assume (for simplicity) that for each predicate p the heads of its rules are identical (with arguments $\bar{x}_p$ say), and the rest of the variables in each rule are disjoint from those in any other program rule. Consider the strongly connected component $\{p_1, \dots, p_n\}$ in the predicate call graph. In the first step of the translation, each rule for predicate p_i of the form $p_i(\bar{x}_{p_i}) :- L_1, \dots, L_m$ is replaced by the rule $p_i(\bar{x}_{p_1} \dots \bar{x}_{p_i} \dots \bar{x}_{p_n}) :- L_1, \dots, L_m$, and the new rule $p_i(\bar{x}_{p_i}) :- p_1(\bar{x}_{p_1} \dots \bar{x}_{p_n}), \dots, p_n(\bar{x}_{p_1} \dots \bar{x}_{p_n})$ is added. After this, the types of the mutually recursive heads are always available. In the final step, any mutually recursive call is replaced by equations unifying it with the recursive head.

The special handling of recursive calls guarantees that the translated program is non-recursive. Hence, any evaluation of them is finite.

Type inference: The translated program can be used for type inference. To infer the types for a predicate p of arity n we simply execute the goal $pred\_p(V_1, \dots, V_n)$. Note that we need to use execution with occurs check since otherwise, the translated programs can act erroneously.

Example 4. Consider the program of Example 3. The goal `pred_deref(V,B,DV)` has two answers: $V = DV \wedge B = list(pair(DV))$, and $V = expr \wedge B = list(expr) \wedge DV = expr$. Thus, there are two possible (polymorphic) types for `deref`. The goal `pred_member(X,L)` has the unique answer $L = list(X)$. Note that we can also use the program to infer types of variables in rules.

One of the principal motivations of Milner style types is that, when used, type correct programs are ensured not to go wrong. This result continues to hold in the disjunctive framework. For any well-typed initial goal G for program P we can show that every goal appearing in a derivation of G is also well-typed.

Translating predicate type declarations: Rules for predicates may be accompanied by a type declaration. During modular compilation, predicates whose rules are given in other modules are not available to the type checker. Instead, only the predicate type declarations will be available.

A *predicate type declaration* for predicate p with arity n is of the form `:- pred` $p(t_1, \ldots, t_n)$, were $t_1, \ldots, t_n$ are type expressions. The translation of the the type declaration above results in a predicate defined by the rule `decl_p(`t_1`,...,`t_n`)`.

In languages like Mercury, the same predicate may be assigned different types (usually in different modules), and type inference is used to determine which version of the predicate should be called. This allows another form of overloading.

Given a predicate type declaration for a predicate `p`, the translation of atoms for `p` can be modified to make use of the information from the predicate type declaration `decl_p`, rather than the information from the rules `pred_p`. In particular, recursive calls for predicates `p` with type declarations can be translated using a *generic* type constraint, rather than a *non-generic* constraint. This is a relaxation of the Milner scheme also used in functional languages like Haskell and Miranda. There are two good reasons to allow this. First, the type declaration contains more accurate information given by the programmer (if it is wrong this will be discovered while checking it). It may be that it is stricter than the type that can be inferred from the rules of `p`. Second, the resulting programs are simpler, since `decl_p` is defined only in terms of type constructors, whereas `pred_p` is in general more complicated.

Example 5. Consider the original program in Example 3 and the following predicate type declarations for `deref` and `member`:

```
:- pred deref(T, list(pair(T)), T).
:- pred member(T,list(T)).
```

The modified part of the translated program is:

```
decl_deref(T,list(pair(T)),T)).
decl_member(T,list(T)).

rule1_deref(V, Binds, DV) :- func__minus(V,V1,T3),
      decl_member(T3, Binds), decl_deref(V1, Binds, DV).
```

Note how in the translation of `deref` we use the type declaration predicate `decl_deref` rather than the translation for recursive predicates.

We can use type inference to create predicate type declarations. Given an answer to the goal $pred\_p(V_1, \ldots, V_n)$ of the form $V_1 = t_1 \wedge \cdots \wedge V_n = t_n$, we can define a predicate type declaration for p as `:- pred` $p(t_1, \ldots, t_n)$. This allows us to build an inference methodology which works bottom up. We use the inferred types for predicates in the leaves of the call graph, and use these results to infer types for the predicates higher up in the call graph. This process can avoid the evaluation of the `pred_p` rules for predicates which call p. Instead, the simpler `decl_p` rules are used.

Example 6. In the translated program of Example 3, we can determine the types for `member` independently of `deref`. Converting this to a predicate declaration we obtain `:- pred member(X, list(X))`. If the rules for `deref` use this definition, we obtain the same answers as before, for the goal `pred_deref(V,B,DV)` but with a smaller derivation tree.

Type checking: Predicate type declarations define a superset of the intended meaning of the predicate. Thus, each local predicate type declaration must be consistent with the type constraints determined by the predicate rules, and it must constrain the types at least as much as the predicate rules do. For simplicity we assume only one type declaration for each local predicate. The predicate type declaration `:- pred` $p(t_1, \ldots, t_n)$ is *correct* if `decl_p(`$v_1, \ldots, v_n$`)` $\rightarrow$ `pred_p(`$v_1, \ldots, v_n$`)`. Hence to check the program we simply need to execute the goal `forall [V1, ..., Vn] decl_p(V1, ..., Vn) => pred_p(V1, ..., Vn)`. In practice, using Prolog we can write a simpler check making use of the non-logical features.

3 More efficient constraint solving

The advantage of translating a type program into a logic program is that we can then use any logic programming system as a constraint solver for the disjunctive type constraints (since they are Herbrand constraints). Unfortunately, this evaluation approach may not be efficient in some cases.

Example 7. Consider the following program, where + is overloaded, as defined in the introduction:

```
what(O) :- A + B = C, D + E = F, G + H = I, J + K = L,
           C + F = M, I + L = N, M + N = O.
```

Evaluating the goal `pred_what(O), int = O` using a logic programming system explores 256 choices to find that the only solution is $O =$ `int`.

As illustrated in the above example, the logic programming evaluation can be very inefficient for our particular needs, with an exponential worst case. It seems impossible to avoid this worst case in general, because the problem of finding a correct type for a program with ad-hoc polymorphism is NP-hard, since we can reduce 3-SAT to this problem.

Even though the worst case is unavoidable, our interest is in type checking real programs. Constraint programming [Marriott and Stuckey, 1998] is routinely used to solve NP-hard problems and techniques used in this field are applicable to our problem. We will show in this section how we can represent disjunctive information about types using ideas from finite domain representation [Hentenryck, 1989], and use generalized propagation [Le Provost and Wallace, 1993] to efficiently ensure that constraints are satisfied.

Type Variables with Domains: Finding a solution to the type constraints can be seen as a constraint satisfaction problem. The possible types are given by the Herbrand Universe of the type constructors. As a result, there is an infinite number of possible types for any variable. For this reason, finite domains are not directly usable in the form provided by current CLP(FD) systems. However, a lazy representation allows us to use the essential ideas in finite domains.

We associate with every type variable a domain of possible values. Initially, this is given by the finite set of type constructors defined in the program. Given the type definitions in Example 1, the initial domain of a type variable X would be $\{\texttt{list}(T_1), \texttt{pair}(T_2), \texttt{node}(T_3), \texttt{expr}, \texttt{fn}(T_4, T_5)\}$ where $T_1, \ldots, T_5$ are new type variables. Conceptually, the type variables $T_1, \ldots, T_5$ have an initial domain of a similar form. This would give an infinite number of variables. This problem can be avoided by representing unconstrained type variables without producing an initial domain. For efficiency, and because we handle unconstrained variables differently in any case, we allow the binding of one variable to another (as in Prolog). This means all further references to the bound variable, reference the variable it is bound to.

Constraints for Type Variables with Domains: Since we can normalize and flatten the rules in the type constraint program we can assume that the constraints appearing in the translated type program have one of two forms: $X = Y$ and $f(X_1, \ldots, X_n) = Y$. When they are reached in a derivation, the procedures equals(X, Y) and fequals$(f(X_1, \ldots, X_n), Y)$, respectively, are called. These two procedures are defined below.

```
equals(X, Y)
  if free(X) then bind(X,Y)
  elseif free(Y) then bind(Y,X)
  else
    DX := domain_of(X)
    DY := domain_of(Y)
    D := domain_intersect(DX,DY)
    if D ≠ ∅ then
      set_domain_of(X,D)
      bind(Y,X)
    else return false
  return true

fequals(f(X1,...,Xn), Y)
  let V be a new variable
  set_domain_of(V,{f(X1,...,Xn)})
  return equals(Y,V)
```

```
domain_intersect(D1, D2)
  D := ∅;
  for each e1 ≡ f(t1,...,tn) ∈ D1
    if exists e2 ≡ f(s1,...,sn) ∈ D2 and
       ffequals(e1,e2) then
       D := D ∪ {f(t1,...,tn)}
  return D

ffequals(f(t1,...,fn), f(s1,...,sn))
  in := true; i := 1
  while in ∧ i ≤ n
    in := in ∧ equals(ti, si)
    i := i + 1;
  return in
```

The procedure free succeeds if the variable is unconstrained. The procedure bind binds the first argument to the second. The procedure domain_of returns the current domain of a variable, while set_domain_of updates the current domain of the first argument to be the second argument. All these procedures are assumed to work with an implicit backtrackable state, in particular whenever a call to equals, fequals or ffequals returns *false* then all the changes to the state made within its execution are undone.

Handling disjunctive constraints: The aim of the constraint solver is to avoid using choice when handling disjunctive constraints. Instead, any predicate in the translated program with more than one defining rule is treated as a (propagation) constraint. Propagation constraints are used to reduce the domains of the variables involved in it without creating choices. Every time the variables involved in a propagation constraint are modified, the propagation constraint is invoked to (possibly) further reduce the domains of the variables involved.

The algorithm to invoke a propagation constraint is illustrated below. It simply executes the goal, collects all answers, and then performs the union of the possible domains that arise in each answer.

```
propagate(p(v_1, ..., v_n))
    for i = 1 to n
        D_i := ∅
    while next_answer(p(v_1, ..., v_n))
        for i = 1 to n
            N_i := domain_of(v_i)
            D_i := domain_union(D_i, N_i)
    for i = 1 to n
        if D_i = ∅ then return false
        set_domain_of(v_i, D_i)
    return true
```

```
domain_union(D_1, D_2)
    D := ∅
    for each f(t_1, ..., t_n) ∈ D_1
        if exists f(s_1, ..., s_n) ∈ D_2 then
            D := D ∪ {f(v_1, ..., v_n)}
            where v_1, ... v_n are new variables
            if |D_1| = 1 ∧ |D_2| = 1 then
                for i = 1 to n
                    if ¬ free(s_i) ∧ ¬ free(t_i) then
                        D_s = domain_of(s_i)
                        D_t = domain_of(t_i)
                        DU = domain_union(D_s,D_t)
                        set_domain_of(v_i,DU)
        else D := D ∪ {f(t_1, ..., t_n)}
    for each f(s_1, ..., s_n) ∈ D_2
        if not exists f(t_1, ..., t_n) ∈ D_1 then
            D := D ∪ {f(s_1, ..., s_n)}
    return D
```

The propagate procedure finds all the answers to an atom $p(v_1, \ldots, v_n)$ given the current domains for each v_i, and unions the resulting domains using domain_union. It assumes next_answer returns *true* with the state set to that of a successful derivation for the atom for each possible successful derivation, and then returns *false* while resetting the state to that existing on the call to propagate. The domain_union procedure unions two domains. If the result is a singleton set, then the procedure unions the domain of each argument individually, otherwise the union is only made on the principal functors of the domains. Note that this is a strict generalization of the usual case for finite domains.

Example 8. Consider the goal `pred_what(O),int = O`. The execution of goal `func_plus(A,B,C)` inside propagate returns four answers, which are unioned to give domain {`float`, `int`} for variables A–C. After the execution of `pred_what(O)` each of the `func_plus` atoms has been executed similarly and determined the domain of {`float`, `int`} for the variables A–O. The `func_plus` atoms remain as propagation constraints. The evaluation of `int = O` reduces the domain of O to {`int`}, which means the constraint `func_plus`(M, N, O) must be re-executed to determine if there is any more information. There is now only one answer, and the domains of M and N are set to {`int`}. These in turn cause re-execution of other delayed constraints, until all the variables have domain {`int`}. In total 56 choices are examined to solve the problem.

Labelling: The constraint solver is incomplete. That is, after propagation quiesces so that no variables domain is changed if any of the remaining propagation constraints are reexecuted, the type constraints may not have an answer.

Example 9. Consider the program

```
:- pred p(int,float,float).
:- pred p(float,int,int).
q(X,Y,Z) :- p(X,Y,Z), p(Y,X,Z).
```

The propagation constraint solver on the goal `pred_q(X,Y,Z)` obtains the domain {`int`, `float`} for each variable, but no typing for `q` exists. The propagation constraints corresponding to the `p` atoms cannot infer any more information but are mutually incompatible.

In order to guarantee an answer we need to enforce that the propagation constraints hold. In finite domains, the labelling method is often used. Simply labelling each variable by assigning it one of its domain of values may not work, because there can be an infinite depth of type expression. Instead, we label only the variables whose domain is not the complete set of declared types: since unconstrained type variables are represented in the implementation as free variables this is straightforward.

Example 10. For the translation of the program above, the goal `pred_q(X,Y,Z)` finished with each variable having domain {`float`, `int`}, and two delayed propagation constraints `decl_p(X,Y,Z)` and `decl_p(Y,X,Z)`. Setting X to `int` wakes the two delayed constraints. The first sets Y and Z to `float` and the second fails. Similarly, setting X to `float` causes failure. Hence, no types for `q` are inferred.

4 Implementation and Evaluation

It is reasonably straightforward to implement the constraint solver described in the previous section using the meta-programming and attributed variable features of modern Prologs. A type variable has an attached domain and list of propagation constraints. When the domain is updated the propagation constraints are re-evaluated. The backtrackable state is automatically maintained by the system. The simple algorithms presented in the previous section are optimized in a number of ways in the implementation.

The largest class of typed constraint logic programs we are aware of are Mercury programs. Thus, our 10 benchmarks come from the modules in the Mercury library and compiler. Among them, we choose the 5 largest files in number of predicates to check and the 5 largest in number of literals per predicate. To these

				All			Half		Other		None		
Benchmark	Lits	Preds	Translate	Solv	LP	Merc	Solv	LP	Solv	LP	Solv	LP	Merc
typecheck	8	160	520 (43-3)	112	56	1350	186	96	196	84	336	212	1710
make_hlds	11	116	500 (47-3)	284	142	1430	470	212	458	214	862	488	1750
code_info	5	169	490 (27-0)	180	80	1020	318	152	300	154	452	234	1660
prog_io	3	115	410 (6-1)	118	68	500	422	376	276	106	700	594	1570
llds_out	12	113	480 (56-5)	114	62	1140	214	114	182	92	362	198	1290
optimize	35	6	240 (2-0)	6	4	390	10	6	10	6	12	10	350
tree234	26	36	120 (103-2)	26	22	610	42	32	50	38	68	50	710
mdchk_unify	26	15	250 (8-0)	52	30	590	86	34	72	38	114	58	850
ite_gen	26	4	230 (0-0)	10	6	150	10	4	16	6	18	10	500
jumpopt	25	10	220 (6-0)	30	20	530	72	40	48	28	88	46	390
boyer	8	18	80 (47-5)	10	6	80	20	12	22	14	30	16	220
what	14	1	10 (0-0)	7	87	220	7	87	16	12393	16	12392	42420

Table 1. Translation and Checking and inferring times

real-life modules, we added the well-known boyer benchmark (in Mercury) and the artificial "what" benchmark in which the same clause occurs twice.

We tested the approach using 4 different versions of each benchmark: All, where all type declarations are included; Half, where a random half are omitted; Other, where the other half are omitted; and None, where all are omitted.

Table 1 gives an idea of the complexity of the benchmarks (number of literals per predicate and total number of predicates) and shows the time (in milliseconds) spent in the translation of the None benchmark version (which is the most complex) and (between brackets) the number of simply–mutually recursive cycles that had to be broken during the translation. Translation times for other verions: All, Half, Other, are close to that of None. As usual, the time spent in I/O is not added. The transformation program is written in SICStus Prolog 3.1#5, the translated benchmarks were run on a SPARCstation 20 under Solaris 2.5. The time in milliseconds is shown to check each (local) type declaration and to infer all types for predicates with rules but no type declarations. We give results for the propagation based solver (Solv), the logic programming system as a solver (LP), and the time spent by the Mercury type checker (Merc). In the first two cases we used ECL^{i}PS^e 3.5. For Mercury, we used the Melbourne implementation, release 0.7 with Boehm garbage collector; each benchmark module.m, was compiled with "mmc -v -S .... module.m".

When comparing the timings between our system and Mercury, one has to take into account that Mercury computes and stores types for all the variables in a program, while we infer types for the variables only implicitly. Also, Mercury reports type errors at the level of the variables, while our system does it at the level of predicates. Furthermore, when an error is found—or an ambiguity—Mercury gives up, i.e. it doesn't enter a new cycle in its fixpoint computation, while our system continues the inference for other parts of the program. Finally, we by no means claim to implement exactly the same type inference as Mercury does, but we might be closer to what it is supposed to implement. The comparison with Mercury is only important in that it sets a yardstick with which to measure new implementations for performing the same task.

The figures suggest that our translation schema leads to very efficient type checking and inference. Surprisingly, the LP system performs very well and most often better than the solver, despite the fact that there is quite a bit of overloading of functors in the Mercury modules. However, the solver is always within a small constant factor of the LP system and it performs much more reliably for highly ambiguous cases. Our implementation is more than a toy: it deals with almost all features of the full-blown language Mercury including: modules, functions, ad-hoc polymorphism, parametric polymorphism, higher-order predicates, etc. Currently, the system is being used by others for type checking/inferencing their Prolog programs according to the Mercury model.

References

Codish, M. and Demoen, B. (1994). Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proc. of Static Analysis Symposium*, number 864 in LNCS, pages 281–297. Springer-Verlag.

Gallagher, J. and de Waal, A. (1994). Fast and precise regular approximations of logic programs. In *Proc. of 11th International Conference on Logic Programming*, pages 599–616.

Hall, C., Hammond, K., Peyton Jones, S., and Wadler, P. (1994). Type classes in Haskell. In *European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*. Springer Verlag.

Hentenryck, P. Van (1989). *Constraint Satisfaction in Logic Programming*. MIT Press.

Hentenryck, P. Van, Cortesi, A., and Charlier, B. Le (1995). Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22:179–209.

Hermenegildo, M. (1994). Some methodological issues in the design of CIAO - a generic, parallel, concurrent constraint system. In *Proc. of Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag.

Janssens, G. and Bruynooghe, M. (1992). Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13:205–258.

Le Provost, T. and Wallace, M. (1993). Generalized constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16:319–359.

Marriott, K. and Stuckey, P.J. (1998). *Programming with Constraints: an Introduction*. MIT Press.

Meyer, G. (1996). Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Technical report, FernUniversität Hagen.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

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

Rouzaud, Y. and Nguyen-Phuong, L. (1992). Integrating modes and subtypes into a Prolog type-checker. In *In Proc., Joint Int. Conf. and Symp. on Logic Programming*, pages 85–97.

Somogyi, Z., Henderson, F., and Conway, T. (1996). The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64.