TCLP: A type checker for CLP($\mathcal{X}$)

Emmanuel Coquery
Emmanuel.Coquery@inria.fr

Abstract

This paper is a presentation of TCLP: a prescriptive type checker for Prolog/CLP($\mathcal{X}$). Using parametric polymorphism, subtyping and overloading, TCLP can be used with practical constraint logic programs that may use meta-programming predicates, coercions between constraint domains (like $\mathcal{FD}$ and $\mathcal{B}$) and constraint solver definitions, including the CHR language. It also features type inference for variables and predicates, so the user can get rid of numerous type declarations.

1 Introduction

Traditionally, the class CLP($\mathcal{X}$) of constraint logic programs, introduced by Jaffar and Lassez [11], is untyped. One of the advantages of being untyped is programming flexibility. For example, `-/2` can be used as the classical arithmetic operator as well as a constructor for pairs. On the other hand, type checking allows the static detection of some programming errors, like for example calling a predicate with an illegal argument.

Several type systems have been created for (constraint) logic programming. The type system of Mycroft and O'Keefe [12, 15] is an adaptation of the Damas-Milner type system [6] to logic programming. It has been implemented in Gödel [10] and Mercury [19]. This type system uses parametric polymorphism, that is, parameters (*i.e.* type variables) are allowed as and in types. For example the type *list* has an argument to specify the type of elements occurring in the list. However this type system is not flexible enough to be used with meta-programming predicates, such as `arg/3`, `=../2` or `assert/1`.

Subtyping is a fundamental concept introduced by Cardelli [2] and Mitchell [14]. The power of subtyping resides in the subtyping rule which states that an expression of type τ can be used instead of an expression of type τ' provided that τ is a subtype of τ':

$$(Sub)\ \frac{U \vdash t : \tau\ ,\ \tau \leq \tau'}{U \vdash t : \tau'}$$

Subtyping can be used to deal with meta-programming by the introduction of a type *term* as a supertype of all types. For example, the subtype relation $list(\alpha) \leq term$, allows to type check the query `arg(N,[X|L],T)`, using the type $int \times term \times term \rightarrow pred$ for `arg/3`, although the second argument is a list. Subtyping can also be used for coercions between constraint domains. For example, it is possible to share variables between CLP($\mathcal{B}$), with type *boolean*, and CLP($\mathcal{FD}$), with type *int*, simply by adding the subtyping relation *boolean* $<$ *int*. This way $\mathcal{B}$ variables can be used with $\mathcal{FD}$ predicates.

Most of the type systems with subtyping that where proposed for constraint logic programs are descriptive type systems, *i.e.* they aim to describe the set of terms for which a predicate is true. On the other hand, there where only few prescriptive type systems with subtyping for logic programming [1, 7, 13, 16, 18]. Moreover, in these systems, subtyping relations between type constructors with different arities, as in $list(\alpha) < term$, are not allowed. Algorithms to deal with such subtyping relations, called non-structural non-homogeneous subtyping, can be found in [17, 20] in the case where the subtyping order forms a lattice, or in [4] for the case of quasi-lattices.

The combined use of subtyping and parametric polymorphism thus offers a great programming flexibility. Still, it can not address the first example given in this paper, that is `-/2` being viewed sometimes as the arithmetic operator and sometimes as a constructor of pairs (as in the predicate `keysort/2`). The solution to this problem resides in overloading. Overloading consists in assigning multiple types to a single symbol. This notion has already been used in numerous languages, such as C, to deal with multiple kinds of numbers in arithmetic operations. With overloading, `-/2` can have both type $int\_expr \times int\_expr \rightarrow int\_expr$ and type $\alpha \times \beta \rightarrow pair(\alpha, \beta)$.

In this paper, we describe TCLP, a type checker for Prolog/CLP($\mathcal{X}$), written in SICStus Prolog with Constraint Handling Rules (CHR) [9]. The type system of TCLP combines parametric polymorphism, subtyping and overloading in order to keep the flexibility of the traditionally untyped CLP($\mathcal{X}$) languages, yet statically detecting programming errors. Section 2 shows examples of how the type system takes advantage of these three features. Section 3 presents the type system of TCLP. In section 4, we describe the basic type declarations and output of TCLP, while section 5 shows how the type system can be extended to handle constraint solver programming, like new CLP($\mathcal{FD}$) constraints or CHR rules. Some benchmarks are presented in section 6 and section 7 concludes.

2 Motivating Examples

The aim of the TCLP type checker is to introduce a typing discipline in constraint logic programs in order to find programming errors, while offering enough flexibility for practical programming. That means dealing with Prolog/CLP($\mathcal{X}$) programming facilities like meta-programming or the simultaneous use of multiple constraint solvers. This goal is achieved using a combination of parametric polymorphism, subtyping and overloading. In the rest of this section, we give examples of how they are used in TCLP.

2.1 Prolog Examples

A first use of parametric polymorphism is the typing of structures that may be used with any type of data. For example, using the type $list(\alpha)$ for lists allows typing `[1,2]` with the type $list(int)$ and `['a','b']` with the type $list(char)$. A consequence is the use of polymorphic types for predicates manipulating these data structures in a generic way. For example, the type of the predicate `append/3` for concatenating lists is $list(\alpha) \times list(\alpha) \times list(\alpha) \rightarrow pred$. Of course, some other predicates may use non generic types when manipulating the data inside structures, like `sum_list/2` having type $list(int) \times int \rightarrow pred$.

Another use of parametric polymorphism is for constraints or predicates that can be used on any term, the best example being `=/2` with type $\alpha \times \alpha \rightarrow pred$. This type simply express that the two arguments of `=/2` must have the same type. Another example resides in term comparison predicates like `'@=<'/2`, which also has type $\alpha \times \alpha \rightarrow pred$.

On the other hand, predicates for manipulating terms cannot be typed using only parametric polymorphism. An example is the predicate `=../2` for decomposing terms. Indeed `T=..L` unifies `L` with the list constituted by the head constructor of `T` and the arguments of `T`. This means that `L` is an non-homogeneous list. Subtyping provides a solution for typing this predicate, through the introduction of the type *term* as the supertype of all types, that is for all types τ, $\tau \leq term$. Using the type $term \times list(term) \rightarrow pred$ for `=../2`, it is possible to type check a query like `[1] =.. ['.',1,[]]` with type $list(int)$ for `[1]`, *atom* for `'.'`, *int* for `1`, $list(\alpha)$ for `[]` and $list(term)$ for `['.',1,[]]`.

Subtyping is also interesting when typing programs that use dynamic predicates, using `assert/1`. The type of `assert/1` is $clause \rightarrow pred$ and the type of `':-'/2` is $pred \times goal \rightarrow clause$. This allows typing queries like `assert((p(X) :- X<1))`. However, without subtyping, queries like `assert(p(1))` are not correctly typed because `p(1)` would be typed *pred*, while `assert/1` expects the type *clause*. Using subtyping with $pred < clause$, `p(1)` is seen with the type *clause* and the query is well-typed.

The operator `-/2`, as showed in the introduction, provides a good example of the use of overloading, with types $int\_expr \times int\_expr \rightarrow int\_expr$, $float\_expr \times int\_expr \rightarrow float\_expr$, $int\_expr \times float\_expr \rightarrow float\_expr$, $float\_expr \times float\_expr \rightarrow float\_expr$ and $\alpha \times \beta \rightarrow pair(\alpha, \beta)$. This example shows the more classical overloading of `-/2` with respect to the different kinds of number as well as its use as a coding for pairs. In this case, subtyping can also be used to deal with the different kind of numbers, with $int\_expr < float\_expr$, using the type $\alpha \times \alpha \rightarrow \alpha, \alpha \leq float\_expr$. However, in the Prolog dialects that we considered, the unification `1=1.0` fails. This led us to choose overloading instead of subtyping for dealing with numerical expressions, thus making a clear distinction in types between integers and floats. An other example is `=/2`. It is used both as the equality constraint and to build pairs of the form `Name=Var` in an option of the predicate `read_term/3`. Thus is has both types $\alpha \times \alpha \rightarrow pred$ and $atom \times term \rightarrow varname$. Other examples include options shared by several different predicates or `','/2` used both as the conjunction and as a constructor for sequences.

2.2 Combining Constraint Domains

A first example is the combination of the Herbrand domain CLP($\mathcal{H}$) with an other domain, such as CLP($\mathcal{FD}$). Prolog is mainly used to handle data structures and for posting constraints. However there can be a stronger interaction when defining, e.g., predicates for labelling. The type used to represent $\mathcal{FD}$ is *int*, already present in the type hierarchy of CLP($\mathcal{H}$). This way $\mathcal{FD}$ variables can be also used as Prolog variables when needed.

Another interesting example is combining CLP($\mathcal{FD}$) and CLP($\mathcal{B}$). Indeed, variables can be shared between the two constraint solvers. This is possible when $\mathcal{B}$ is represented as the set $\{0,1\}$. In this case 0 and 1 have type *boolean* and *boolean* $<$ *int*. In this way, $\mathcal{B}$ variables can also be used with $\mathcal{FD}$ constraints.

A last example is reified constraints. This represent a combination of CLP($\mathcal{H}$), CLP($\mathcal{FD}$) and CLP($\mathcal{B}$). Constraints like `'#<=>'/2` accept other constraints as arguments. In order to handle these cases, $\mathcal{FD}$ constraints are typed with type *fd_constraint*. The subtype relations *fd_constraint* $<$ *pred* and *fd_constraint* $<$ *boolean_expr* allows these constraints to be used both in boolean expressions and as predicates in Prolog clauses.

3 The Type System

3.1 CLP($\mathcal{X}$) Programs

CLP programs are built upon a denumerable set $\mathcal{V}$ of variables, a finite set $\mathcal{S}$ of symbols, given with their arity, a set $\mathcal{F} \subseteq \mathcal{S}$ of function symbols and a set $\mathcal{P} \subseteq \mathcal{S}$ of predicate and constraint symbols. $\mathcal{P}$ is supposed to contain the equality constraint symbol `=/2`. Terms are built upon $\mathcal{F} \cup \mathcal{V}$. An atom is of the form $p(t_1, \ldots, t_n)$, where $p/n \in \mathcal{P}$ and $t_1, \ldots, t_n$ are terms. A query is a finite sequence of atoms. When it is necessary to distinguish predicate atoms (built using a predicate symbol) and constraint atoms (built with a constraint symbol), queries are noted $c \mid \alpha$ where c is the constraint part of the query and α is the predicate part of the query. A clause is an expression $A \leftarrow Q$ where A is a predicate atom and Q is a query. A constraint logic program is a set of clauses and queries.

The execution model we consider for constraint logic programs is the CSLD rewriting relation :

Definition 1 *Let P be a CLP($\mathcal{X}$) program. The rewriting relation $\longrightarrow_{CSLD}$ over queries is defined as the smallest relation satisfying the following CSLD rule:*

$$\frac{\begin{array}{c}\mathtt{p}(N_1, \ldots, N_k) \leftarrow c' \mid A_1, \ldots, A_n \in \theta(P) \\ \mathcal{X} \models \exists (c \wedge M_1 = N_1 \wedge \ldots \wedge M_k = N_k \wedge c')\end{array}}{\begin{array}{l}c \mid \alpha, \mathtt{p}(M_1, \ldots, M_k), \alpha' \longrightarrow_{CSLD} \\ \qquad\qquad c, M_1 = N_1, \ldots, M_k = N_k, c' \mid \alpha, A_1, \ldots, A_n, \alpha'\end{array}}$$

where θ is a renaming of the clause with fresh variables.

3.2 Types

Types are (possibly infinite) terms built upon a signature of *type constructors*, denoted by κ, and *type variables* also called *parameters*, noted $\alpha, \beta, \ldots$. Types are noted τ and the set of types is noted $\mathcal{T}$. The subtyping order $\leq$ on types is induced by an order $<_{\mathcal{K}}$ on type constructors and a relation $\iota_{\kappa_1,\kappa_2}$ between the argument positions of each pair (κ_1, κ_2) of type constructors. For all type constructors κ_1, κ_2, $\iota_{\kappa_1,\kappa_2}$ is an injective partial function and $\iota^{-1}_{\kappa_1,\kappa_2} = \iota_{\kappa_2,\kappa_1}$. For all $\kappa_1 \leq_{\mathcal{K}} \kappa_2 \leq_{\mathcal{K}} \kappa_3$, $\iota_{\kappa_1,\kappa_3} = \iota_{\kappa_2,\kappa_3} \circ \iota_{\kappa_1,\kappa_2}$. For two types $\tau = \kappa(\tau_1, \ldots, \tau_m)$ and $\tau' = \kappa'(\tau'_1, \ldots, \tau'_n)$, $\tau \leq \tau'$ if and only if $\kappa \leq_{\mathcal{K}} \kappa'$ and for all $i, j \in \iota_{\kappa,\kappa'}$, $\tau_i \leq \tau'_j$. Moreover the type order is supposed to form a quasi-lattice, that is a partial order where the existence of a lower (resp. upper) bound to a non-empty set of types implies the existence of a greatest lower bound (resp. least upper bound) for this set. A type substitution is a mapping from type variable to types, extended the usual way into a mapping from types to types. A type substitution Θ is ground if for all type variable α, $\Theta(\alpha)$ is ground.

Ground types are interpreted as sets of terms, while non ground types are interpreted as mappings from ground substitutions to sets of terms. For example, the type $list(int)$ is interpreted as the set of the lists of integers, while the infinite type $list(list(\ldots))$ is interpreted as the set of lists that contain only lists that contain only lists ... [1]. The subtyping relation is interpreted as the inclusion of these sets of terms. A more formal description of types and of the subtyping relation can be found in [4].

To each functor f/n is associated a set $types(f/n)$ of *type schemes* of the form $\forall\alpha_1 \ldots \forall\alpha_n \tau_1 \times \ldots \times \tau_n \to \tau$, (abbreviated $\forall \tau_1 \times \ldots \times \tau_n \to \tau$), where $\{\alpha_1, \ldots, \alpha_n\}$ is the set of variables appearing in $\tau_1 \times \ldots \times \tau_n \to \tau$. We assume the existence of a particular type *pred* for the type of predicates: for all predicate and constraint symbols $p/n \in \mathcal{P}$, it is supposed that there is at least one type scheme $\forall \tau_1 \times \ldots \times \tau_2 \to \tau \in types(p/n)$ such that $\tau \leq pred$. On can note that some symbols may be overloaded both as function symbols and predicates symbols, such as `=/2` with types $\forall \alpha . \alpha \times \alpha \to pred$ and $atom \times term \to varname$.

3.3 Well Typed Programs

The typing rules of TCLP, given in Table 1, allow to deduce type judgment of the form $U \vdash$ typed expression, where U is a *typing environment*, that is a mapping from $\mathcal{V}$ to $\mathcal{T}$. A clause $p(t_1, \ldots, t_n) \leftarrow Q$ is *well-typed* if for all type schemes $\forall \tau_1 \times \ldots \times \tau_n \to \tau \in types(p/n)$ with $\tau \leq pred$, there exists a typing environment U such that $U \vdash p(t_1, \ldots, t_n) \leftarrow Q\ Clause_{\tau_1 \times \ldots \times \tau_n}$. A program is *well-typed* if all its clauses are well-typed. A query Q is *well-typed* if there exists a typing environment U such that $U \vdash Q\ Query$.

Basically, the type system of TCLP adds the subtyping rule [2, 14] to the rules of Mycroft and O'Keefe [15]. Overloading is handled in the side condition of rules (*Func*), (*Atom*) and (*Head*) by considering all possible type schemes for each occurrence of overloaded symbols. The type annotations appearing in the rules (*Head*) and (*Clause*) are used to keep track of the type used for the head of the clause. The distinctions between rules (*Head*) and (*Atom*) express the principle of *definitional genericity* [12], that the type of the head of a clause must be equivalent up-to renaming to the type of the predicate defined by this clause. This condition of definitional genericity is useful for the correctness properties ("subject reduction") of the type system [3, 8]. The rule (*Head*), used for typing heads of clauses, thus allows only renaming substitutions of the type declared for the predicate.

Theorem 1 (subject reduction) *[3] Let P be a well-typed program and Q a well typed query, i.e. $U \vdash Q\ Query$ for some typing environment U. If $Q \longrightarrow_{CSLD} Q'$ then there is a typing environment U' such that $U' \vdash Q'\ Query$.*

[1] this does not mean that the terms in this set are infinite: for example `[]`, `[[]]` and `[[],[]]` are in this set.

$$(Var) \qquad \{x:\tau,\ldots\} \vdash x:\tau$$

$$(Func) \qquad \frac{U\vdash t_1{:}\sigma_1 \;\; \sigma_1\leq\tau_1\Theta \;\; \ldots \;\; U\vdash t_n{:}\sigma_n \;\; \sigma_n\leq\tau_n\Theta}{U\vdash f(t_1,\ldots,t_n){:}\tau\Theta}$$

where Θ is a type substitution
and $\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \mathit{types}(f/n)$

$$(Atom) \qquad \frac{U\vdash t_1{:}\sigma_1 \;\; \sigma_1\leq\tau_1\Theta \;\; \ldots \;\; U\vdash t_n{:}\sigma_n \;\; \sigma_n\leq\tau_n\Theta}{U\vdash p(t_1,\ldots,t_n) \;\; \mathit{Atom}}$$

where Θ is a type substitution
and $\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \mathit{types}(p/n)$, with $\tau \leq \mathit{pred}$.

$$(Head) \qquad \frac{U\vdash t_1{:}\sigma_1 \;\; \sigma_1\leq\tau_1\Theta \;\; \ldots \;\; U\vdash t_n{:}\sigma_n \;\; \sigma_n\leq\tau_n\Theta}{U\vdash p(t_1,\ldots,t_n) \;\; \mathit{Head}_{\tau_1\times\ldots\times\tau_n}}$$

where Θ is a renaming substitution
and $\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \mathit{types}(p/n)$, with $\tau \leq \mathit{pred}$.

$$(Query) \qquad \frac{U\vdash A_1 \;\; \mathit{Atom} \;\; \ldots \;\; U\vdash A_n \;\; \mathit{Atom}}{U\vdash A_1,\ldots,A_n \;\; \mathit{Query}}$$

$$(Clause) \qquad \frac{U\vdash Q \;\; \mathit{Query} \qquad U\vdash A \;\; \mathit{Head}_{\tau_1\times\ldots\times\tau_n}}{U\vdash A \leftarrow Q \;\; \mathit{Clause}_{\tau_1\times\ldots\times\tau_n}}$$

Table 1: The TCLP typing rules with overloading.

It is worth noting that the CSLD resolution is an abstract execution model, which proceeds only by constraint accumulation. The theorem above does not hold for more concrete execution models that perform substitution steps. Let us consider the predicates `p/1` and `q/1`, with $\mathit{int} \rightarrow \mathit{pred} \in \mathit{types}(\texttt{p/1})$ and $\mathit{byte} \rightarrow \mathit{pred} \in \mathit{types}(\texttt{q/1})$. Let us suppose that `p/1` is defined by `p(500)`. The query `p(X),q(X)` is well typed with `X` : *byte*. A step of CSLD resolution produces the query `X=500,q(X)`. A substitution step produces the query `p(500)`, which is not well typed since `500` does not have type *byte*. This can be viewed as a weakness of the type system, but we believe this is the price to pay for flexibility. Moreover, it is possible to keep the type of variables at run-time in order to get stronger subject reduction theorem [8] for an execution model that performs substitution steps.

3.4 Type Checking

The typing rules of Table 1 are syntax directed. Without overloading, the type checking algorithm, given a typing environment U and the type of symbols, basically collects subtype inequalities along the derivation of the expression to check and then check the satisfiability of collected subtyping constraints, using the algorithm described in [4]. This type checking algorithm can be extended to infer a typing environment U for which the expression is well-typed, simply by replacing the type of variables appearing in the expression to type check by parameters. Then checking the satisfiability of the resulting subtyping constraint system determines the existence of a typing environment U.

Overloading introduces non-determinism in the rules (*Func*) and (*Atom*). For type checking expressions, subtype inequalities are first collected along the derivation by replacing the type of overloaded symbols by type variables. Then the possible typings for each occurrence of overloaded symbols are enumerated by checking the satisfiability of the subtype constraint system. In order to remain efficient, the enumeration proceeds with the Andorra principle. This principle, first introduced for the parallelization of Prolog [5], consists in delaying choice points until time where all deterministic goals have been executed. This strategy proves to be sufficient to deal with overloading in TCLP, mainly because in most cases the type information coming from the context of an expression is sufficient to disambiguate the type of overloaded symbols in this expression.

The type checking algorithm used in TCLP is simply the combination of the type inference for variables with the enumeration of possible types for overloaded symbols.

3.5 Type Inference For Predicates

In a prescriptive type system, type reconstruction can be used to omit type declarations and still type check the program by inferring the type of undeclared predicates using their defining clauses [12], if it exists, and raising an error otherwise. Since in TCLP, a predicate can accept any argument of a subtype of the type of declared predicate, the type $term \times \ldots \times term \rightarrow pred$ is always a possible type. Because this type is not very informative, we use a heuristic type inference algorithm [8]. Basically it tries to combine the different type informations taken from the functors and variables appearing the head of the defining clauses to deduce a more informative type. In the presence of overloaded symbols, several heuristic types can be found by enumerating the possible types for these symbols. The current implementation uses only the first one in the typing of the remaining part of the program. This choice was made to avoid the multiplication of overloaded predicates. The enumeration proceeds by first choosing the last declared type for each overloaded symbol. This enumeration strategy proves to be right most of the time, because the last declared type for an overloaded symbol usually correspond to the currently defined predicate.

4 Standard Use Of TCLP

4.1 Type Declarations

We now introduce the concrete syntax of TCLP type declarations. These declarations take the form of Prolog directives. They can be placed either in the program source or in a separated file with the suffix `.typ`. They consist in type constructor declarations, type order declarations and type scheme declarations.

Type constructor declarations are done using any one of the following two syntaxes:

`:- type` t/n.   `:- type` $t(A_1, \ldots, A_n)$.

Both directives declare a a type constructor t with n arguments. For example the type constructor *list* can be declared by

```
:- type list/1.
```

Type order declarations are done using the directive `order`:

:- order $t(A_1,\ldots,A_m)$ < $u(B_1,\ldots,B_n)$.

which declares that $t <_{\mathcal{K}} u$. The relation $\iota_{t,u}$ is deduced from the variables appearing as arguments: if $A_i = B_j$ then $(i,j) \in \iota_{t,u}$. For example:

```
:- order assoc(A,B) < tree(B).
```

declares that $assoc <_{\mathcal{K}} tree$ and that $\iota_{assoc,tree} = \{(2,1)\}$.

The syntax for declaring type schemes is:

:- typeof $f(t_1,\ldots,t_n)$ is t.

where t_i and t are types. This declares that the type scheme $\forall t_1 \times \ldots \times t_n \rightarrow t$ is in $types(f/n)$. For example:

```
:- typeof append(list(A),list(A),list(A)) is pred.
```

declares that $\forall \alpha. list(\alpha) \times list(\alpha) \times list(\alpha) \rightarrow pred \in types($`append/3`$)$. Overloaded symbols simply have several declarations (one per type scheme).

Type constructor and type scheme syntax can also be combined:

```
:- type list(A) is [ [] , [ A | list(A) ] ].
```

is syntactic sugar for

```
:- type list/1.
:- typeof [] is list(A).
:- typeof [ A | list(A) ] is list(A).
```

In addition to explicit declarations, TCLP implicitly adds default declarations. For every declared type constructor κ, the declaration that $\kappa <_{\mathcal{K}} term$ is added to ensure that it is still a supertype of all types. Numbers are implicitly declared to have either type *byte*, *int* or *float*. All non-numeric constants are declared to have type *atom* except for characters, which are declared to have type *char* with $char <_{\mathcal{K}} atom$. Still, thanks to overloading, non-numeric constants may also have other types corresponding to their use in specific situations. For example, `write/0` has both the type *atom* and the type *io_mode*. Using these types, the following query, for opening a file named "write" in writing mode, is well typed: `open(write,write,Stream)`, the first occurrence of write being typed as *atom* and the second as *io_mode*. Finally any functor f/n that has no declared type scheme has the default type scheme $term \times \ldots \times term \rightarrow term$.

4.2 TCLP Invocation

TCLP can be used either as a stand-alone executable (by typing `tclp file.pl` in the shell) or as a library for SICStus Prolog. When invoked, TCLP determines and loads a standard type library, usually named `stdlib.typ`. This library contains the type definitions and types for built-in predicates of the selected Prolog dialect, currently either ISO, GNU or SICStus Prolog. In the case of SICStus Prolog, type files for each library are automatically loaded when encountering the corresponding `use_module` directive.

When invoked on a source file, TCLP prints the types inferred for undeclared predicates using the syntax for type scheme declarations. This allows to reuse the types inferred by TCLP for type checking other libraries or same file after some modifications. For example, the type inference of the predicate `append/3`:

```
append([],L,L).
append([X|L],L2,[X|R]) :- append(L,L2,R).
```

produces the following output:

```
:- typeof append(list(A),list(A),list(A)) is pred.
```

If a type error is encountered, TCLP prints it and exits immediately. Here we give examples of ill-typed queries and clauses with the error message displayed by TCLP:

- Illegal type for an argument

  ```
  :- X is Y << 3.5 .

  ! Incompatible type : 3.5 has type float but is
    required to have type int_expr
  ```

- No type can be found for a variable

  ```
  :- length(N,L), member(a,L).

  ! Incompatible types for L : int and list(top)
  ```

- Violation of definitional genericity

  ```
  :- typeof p(list(A)) is pred.
  p([1]).

  ! Incompatible type : 1 has type byte but is
    required to have type A
  ```

- Error on an overloaded symbol

  ```
  :- X is 3 << (2 - 3.5).

  ! Can't find a good type for (-)/2
  ```

5 Advanced Definitions

An interesting feature of TCLP is the possibility to extend the typing rules. The aim is the type checking of phrases that are similar to clauses from the type checking point of view. This extension uses declarations that specify how these phrases must be cut into sets of heads and bodies. The heads are type checked using rules similar to the (*Head*) rule and the bodies are type checked as queries. Note, however, that a new subject reduction theorem must be proved in order to ensure the correctness of the system thus obtained. We show two examples of type system extensions, one for primitive CLP($\mathcal{FD}$) constraints definitions in SICStus Prolog and another for the CHR language [9].

5.1 CLP($\mathcal{FD}$) Primitive Constraints

In SICStus, primitive constraints can be declared using `'+:'/2`, `'-:'/2`, `'+?'/2` and `'-?'/2`. In order to type check these declarations one may want to introduce new typing rules. This is achieved using the declaration

```
:- tclp__define_clause_op(BinOp,Type).
```

where `BinOp` is the binary operator that separates the head and the body and `Type` is the type of the head. For example, the declaration

```
:- tclp__define_clause_op('+:',fd_constraint).
```

adds the following typing rule:

$$\frac{U \vdash H \ \mathit{Head}' \qquad U \vdash B \ \mathit{Query}}{U \vdash H \ \texttt{+:} \ \ B \ \mathit{Clause}}$$

where $U \vdash H \ \mathit{Head}'$ is derived using the rule (Head'), which differ from (Head) only by the side condition: $\tau \leq \mathit{pred}$ becomes $\tau \leq \mathit{fd\_constraint}$ in (Head').

5.2 CHR Rules

There are three kinds of CHR rule: C `==>` Q (propagation rule), C `<=>` Q (simplification rule) and $C_1 \setminus C_2$ `<=>` Q (simpagation rule, *i.e.* both a simplification rule and a propagation rule). C, C_1 and C_2 are sequences of CHR constraints. Q is either a query or Q_1 `|` Q_2 where Q_1 and Q_2 are queries. In order to handle these rules, the declaration `tclp__define_clause/5` is used. We refer to the TCLP documentation for the precise syntax of these declarations. The declarations for CHR rules are given in appendix A. Here we give the typing rule for propagation rules (other rules are similar). Type judgment of the form $U \vdash H \ \mathit{Head}''$ are derived from a rule (Head'') similar to (Head) excepted that the side condition $\tau \leq \mathit{pred}$ is replaced by $\tau \leq \mathit{chr\_constraint}$.

$$\frac{U \vdash H_1 \ \mathit{Head}'' \ \ldots \ U \vdash H_n \ \mathit{Head}'' \ U \vdash B \ \mathit{Query}}{U \vdash H_1, \ldots, H_n \ \texttt{<=>} \ B \ \mathit{Clause}}$$

Type inference can still be used with the new type system, as shown in the following example. This example consists in a constraint solver for finding greatest common divisor and was taken from the CHR web page. The CHR rules:

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=> N=<M | L is M mod N, gcd(L).
```

produce the following output in TCLP

```
:- typeof gcd(int) is chr_constraint.
```

6 Experimental Evaluation

The performance of the system has been evaluated on a GNU/Linux 2.4 system with an Intel Pentium 4 CPU at 2 GHz, 256 Mb of RAM using SICStus 3.9.1 and a preliminary version of TCLP 0.4. Running times for 16 SICStus Prolog libraries are shown in Table 2. The first column indicates the name of the library. The remaining column are divided in two groups: the first group indicates running times when using pure type checking, that is without type inference for predicates, while the second group indicates running times using type inference for all predicates that are not exported by the library. Each group contains three columns. The first one, *Overld*, is the time consumed to solve ambiguous overloaded symbols. The second one, *T.check* indicates the type checking time (including type inference in the case of the second group). The last column, *Total*, indicates the running total time, including loading type libraries and building the resulting type order.

	Pure type checking			With predicate type inference		
File	Overld	T.check	Total	Overld	T.check	Total
arrays	0.18 s	0.80 s	2.52 s	0.19 s	1.00 s	2.73 s
assoc	0.52 s	2.16 s	3.89 s	0.87 s	3.88 s	5.61 s
atts	0.75 s	1.92 s	3.72 s	1.35 s	3.26 s	5.04 s
bdb	0.84 s	3.14 s	6.08 s	1.07 s	4.19 s	7.06 s
charsio	0.07 s	0.40 s	1.99 s	0.09 s	0.43 s	2.00 s
clpr	29.10 s	47.05 s	49.68 s	97.60 s	142.49 s	145.76 s
fastrw	0.05 s	0.20 s	1.83 s	0.12 s	0.32 s	1.98 s
heaps	0.49 s	1.87 s	3.58 s	1.51 s	5.50 s	7.24 s
jasper	0.32 s	0.98 s	3.21 s	0.48 s	1.36 s	3.52 s
lists	0.96 s	1.86 s	3.44 s	1.23 s	2.63 s	4.24 s
ordsets	0.89 s	2.35 s	3.92 s	3.64 s	7.33 s	8.92 s
queues	0.12 s	0.44 s	2.14 s	0.17 s	0.55 s	2.26 s
sockets	0.82 s	1.83 s	4.02 s	0.77 s	2.12 s	4.15 s
terms	0.44 s	1.32 s	2.90 s	0.54 s	1.72 s	3.31 s
trees	0.27 s	0.79 s	2.47 s	0.32 s	1.17 s	2.89 s
ugraphs	7.39 s	14.17 s	16.28 s	11.20 s	31.97 s	34.04 s

Table 2: Running times

Running times prove that TCLP is fast enough to be used in practice, the worst time being obtained for the clpr library which represents about 4400 lines of code and 527 inferred predicates. When running on small files, most of the running time is used to compute all data structures related to TCLP declarations. These computations usually take 2 to 3 s depending on declarations that are specific to each library. The time used to solve overloaded symbols is very low, usually less than 50% (68% in the worst case) of the total type checking time, thanks to the enumeration strategy. The overhead of type inference w.r.t. pure type checking can be explained by the fact that pure type checking considers the program clauses one by one, while type checking with predicate type inference considers clauses grouped by strongly connected components of the call graphs, which leads to considerably larger subtyping constraint systems and to a higher number of overloaded symbols to be treated at once.

7 Conclusion

We presented TCLP, a prescriptive type checker for Prolog/CLP($\mathcal{X}$), which can be used with practical constraint logic programs. Thanks to parametric polymorphism, subtyping and overloading, it can type check queries and goals using generic data structures, term decomposition and meta-programming predicates, overloaded symbols such as `'-'/2`, or the combination of multiple constraint solvers including reified constraints. The possibility to extend the type system, makes it possible to use TCLP for constraint solver programming like extending CLP($\mathcal{FD}$) with new constraints or using the CHR language. TCLP features type inference for variables and for predicates, so the user can get rid of numerous type declarations. The experimental evaluation of TCLP on 16 SICStus Prolog libraries, including CLP($\mathcal{R}$), proved that the type checker is fast enough to be used in practice. For these reasons, we believe that TCLP is a good tool for type checking constraint logic programs.

As future work, we intend to develop a formalization of the extensions of the type system. We also want to extend TCLP to other Prolog dialects such as, e.g., Ciao Prolog or SWI Prolog.

Availability TCLP is distributed under the GNU Lesser General Public License, and is available as sources, binaries for Linux/x86 and MacOSX or as a library for SICStus Prolog. An online demo can be found on the TCLP web site:
`http://contraintes.inria.fr/~coquery/tclp`

References

[1] C. Beierle. Type inferencing for polymorphic order-sorted logic programs. In *12th International Conference on Logic Programming ICLP'95*, pages 765–779. The MIT Press, 1995.

[2] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[3] E. Coquery and F. Fages. TCLP: overloading, subtyping and parametric polymorphism made practical for constraint logic programming. Technical report, INRIA Rocquencourt, 2002.

[4] E. Coquery and F. Fages. Subtyping constraints in quasi-lattices. In P. Pandya and J. Radhakrishnan, editors, *Proceeding of the 23rd Conference On Foundations Of Software Technology And Theoretical Computer Science*, LNCS. Springer, 2003.

[5] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model. In *Proceedings of the 8th International Conference on Logic Programming ICLP'91*, pages 443–456. MIT Press, 1991.

[6] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[7] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming ESOP'88*, LNCS, pages 79–93. Springer-Verlag, 1988.

[8] F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, November 2001.

[9] T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[10] P. Hill and J. Lloyd. *The Gödel programming language.* MIT Press, 1994.

[11] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 1987 Symposium on Principles of Programming Languages POPL'87*, pages 111–119, 1987.

[12] T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–217. MIT Press, 1991.

[13] G. Meyer. Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Technical report, Informatik Berichte 200, Fern Universitat Hagen, 1996.

[14] J. Mitchell. Coercion and type inference. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages POPL'84*, pages 175–185, 1984.

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

[16] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[17] F. Pottier. Simplifying subtyping constraints: a theory. *To appear in Information and Computation*, 2002.

[18] G. Smolka. Logic programming with polymorphically order-sorted types. In *Algebraic and Logic Programming ALP'88*, number 343 in LNCS, pages 53–70. J. Grabowski, P. Lescanne, W. Wechler, 1988.

[19] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[20] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium SAS'96*, number 1145 in LNCS, pages 349–365, 1996.

A TCLP Declarations For CHR Rules

We use an auxiliary Prolog file, `chrcore.pl`, to decompose CHR rules in sets of heads and bodies. The predicate `chr__heads/3` decomposes a sequence of heads into a list of Head-Location-Type triplets, while the predicate `chr__clauses/4` breaks a rule into a body and a list of heads. In the last clause, the type *chr_constraint* is specified, which leads TCLP to use the rule (*Head*$''$).

The predicate `user:arg_location/2` is predefined in TCLP and is used to provide the location of the different parts of the rule in the program source code to TCLP, mainly for reporting errors in the right place.

```
myappend([],X,X).
myappend([X|L],L2,[X|R]) :- myappend(L,L2,R).

%% rule decomposition
chr__clause((HeadsDef <=> Body), Location,
            Heads, [ Body - BodyLoc ]) :-
        user:args_location(Location,[HeadsLoc, BodyLoc]),
        chr__heads(HeadsDef, HeadsLoc, Heads).
chr__clause((HeadsDef ==> Body), Location,
            Heads, [ Body - BodyLoc ]) :-
```

```
        user:args_location(Location,[HeadsLoc, BodyLoc]),
        chr__heads2(HeadsDef, HeadsLoc, Heads).

%%  sequence of heads to list
chr__heads((H1\H2), Location, Heads) :- !,
        user:args_location(Location,[L1,L2]),
        chr__heads2(H1,L1,Heads1),
        chr__heads2(H2,L2,Heads2),
        myappend(Heads1, Heads2, Heads).
chr__heads(H,L,Hds) :-
        chr__heads2(H,L,Hds).

chr__heads2((H1,H2), Location, Heads) :- !,
        user:args_location(Location,[L1,L2]),
        chr__heads2(H1,L1,Heads1),
        chr__heads2(H2,L2,Heads2),
        myappend(Heads1, Heads2, Heads).
chr__heads2(H,L,[H-L-chr_constraint]).
```

The following code comes from the type declaration file for the CHR library, `chr.typ`. The first directive loads the code from `chrcore.pl`. The two last directives define, given a rule and its location, a list of heads and a list of bodies, using `chr__clause/4` from `chrcore.pl`. Using these directives, TCLP will decompose CHR rules in sets of heads and bodies, heads being type checked with the rule (*Head''*), while bodies are type checked as queries. The difference between the second and the third directive is that the second directive discards the name of rules (names are given to rules in CHR using the notation *Name* : `@` *Rule*).

```
%% load prolog code for parsing CHR rules
:- tclp__load_prolog(tclplib('sicstus/chrcore.pl')).

%% the declarations simply consist in the call to predicates
%% defined in chrcore.pl
:- tclp__define_clause((_ @ Rule), Location, Heads, Bodies,
                       (user:args_location(Location,
                                           [_,RuleLoc]),
                           chr__clause(Rule, RuleLoc,
                                       Heads, Bodies))).
:- tclp__define_clause(Rule, Location, Heads, Bodies,
                       chr__clause(Rule, Location,
                                   Heads, Bodies)).
```