TCLP: overloading, subtyping and parametric polymorphism made practical for constraint logic programming

Emmanuel Coquery and François Fages

Projet Contraintes, INRIA-Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France,
emmanuel.coquery@inria.fr, francois.fages@inria.fr

Abstract. This paper is a continuation of our previous work on the TCLP type system for constraint logic programming. Here we introduce overloading in TCLP and describe a new implementation of TCLP in the Constraint Handling Rules language CHR. Overloading, that is assigning several types to symbols, e.g. for integer and floating point arithmetic, makes it possible to avoid subtype relations like integer subtype of float, that are not faithful to the behavior of some predicates, e.g. unification between an integer and its float representation fails in ISO Prolog. We describe a new implementation of TCLP in Prolog and CHR where overloading is resolved by backtracking with the Andorra principle. Experimental results show that the new implementation of TCLP in CHR outperforms the previous implementation in CAML w.r.t. both runtime efficiency, thanks to simplifications by unification of type variables in CHR, and w.r.t. the percentile of exact types inferred by the TCLP type inference algorithm, thanks to overloading.

1 Introduction

The notion of subtyping is a fundamental concept introduced by Cardelli [2] and by Mitchell [12] in the context of functional languages, as another form of polymorphism complementing parametric polymorphism. The power of subtyping rests on the subsumption rule, that expresses the substitutivity of any expression of type τ wherever an expression of type τ' is expected, provided that τ is a subtype of τ':

$$(Sub)\ \frac{U \vdash t{:}\tau \quad \tau < \tau'}{U \vdash t{:}\tau'}$$

Overloading, also called ad hoc polymorphism, allows assigning several types to function or predicate symbols. Contrarily to subtyping, where all objects of some type τ have all supertypes $\tau' \geq \tau$, the different types assigned to overloaded symbols are specific to these symbols, that is why overloading is called ad hoc polymorphism. Arithmetic operations naturally apply to both integer and real numbers. For example, the addition + is naturally overloaded and can have types

$int \times int \rightarrow int$, $int \times float \rightarrow float$, $float \times int \rightarrow float$, $float \times float \rightarrow float$. In absence of overloading, the same set of types can sometimes be obtained by subtyping and constrained types [15]. For example + may be equivalently assigned type $\forall \alpha \leq float\ \alpha \times \alpha \rightarrow \alpha$, with the subtype relation $int \leq float$. Although elegant, this approach does not generalize well, and overloading generally provides a greater flexibility than subtyping.

In this paper we add overloading to the TCLP type system for constraint logic programming [6]. TCLP is a prescriptive type system which combines subtyping with parametric polymorphism, and provides algorithms for type checking and type inference for predicates. Parametric polymorphism, as introduced for Prolog in [13, 11] and in Gödel [10] and Mercury [17], allows typing homogeneous lists with a polymorphic type $list(\alpha)$ which can be instanciated to types for representing lists of integers, characters, list of lists of integers, etc. Overloading is considered in this setting in [5]. TCLP adds to this approach *subtyping* as a mean of typing metaprogramming predicates and automatic coercions between constraint domains. Metaprogramming predicates impose that all objects can be decomposed as terms, hence a type *term* is introduced as a supertype of all types. In particular, we have the subtype relations between type constructors of different arities like $list(\alpha) \leq term$.

Without subtyping, type inference is equivalent to solving a system of equalities between type expressions, which can be done by unification. With subtyping, type inference is equivalent to solving a system of *inequalities* between type expressions. Several algorithms for solving subtyping inequalities have been proposed in the literature. They can be classified along several axes:

- the domain of types: finite types [12, 9, 7], regular (recursive) types [1, 20, 15], or infinite types [19],
- the structure of types: lattices [1, 18, 15], quasi-lattices [16] posets with suprema [6], partial orders [7],
- the subtyping relation: structural extension of a subtyping relation on basic types, subtyping relation between different type constructors with the same arity [7], or between type constructors of different arities [15, 6].

In this paper we describe a new implementation of TCLP in Prolog and the Constraint Handling Rules language CHR [8]. In section 2, we review the algorithms involved in TCLP and show how overloading can be treated by backtracking with the Andorra principle. In section 3 we describe the implementation in CHR of Pottier's algorithm for solving subtype inequality constraints. In section 5 we propose TCLP types for ISO prolog, CLP($\mathcal{R}$), CLP($\mathcal{Q}$), CLP($\mathcal{FD}$) and CLP($\mathcal{B}$). In section 7 we report our evaluation results on 20 Sicstus Prolog libraries and on standard CLP programs. We show that the new implementation of TCLP in CHR outperforms the previous implementation in CAML w.r.t. both runtime efficiency, thanks to simplifications by unification of type variables in CHR, and w.r.t. the percentile of exact types inferred by the TCLP type inference algorithm, thanks to overloading. In particular we show that the Andorra principle suffices to deal efficiently with overloaded symbols in TCLP,

and that more sophisticated constraint programming techniques, like e.g. constructive disjunction, were not necessary to type check practical programs with overloading.

2 Adding overloading to the TCLP type system

2.1 TCLP Type checking

The typing rules of TCLP basically add the subtyping rule of Cardelli and Mitchell [2, 12] to the rules of Mycroft and O'Keefe [13]. By a simple transformation [6] we get the rules depicted in table 1 for deriving type judgments of the form $U \vdash$ typed expression where U is a typing for variables.

(Var)	$\{x : \tau, \ldots\} \vdash x : \tau$	
(Func)	$\dfrac{U \vdash t_1 : \sigma_1 \;\; \sigma_1 \leq \tau_1\Theta \;\; \ldots \;\; U \vdash t_n : \sigma_n \;\; \sigma_n \leq \tau_n\Theta}{U \vdash f(t_1,\ldots,t_n) : \tau\Theta}$	Θ is a type substitution, $\tau_1 \ldots \tau_n \rightarrow \tau \in types(f)$
(Atom)	$\dfrac{U \vdash t_1 : \sigma_1 \;\; \sigma_1 \leq \tau_1\Theta \;\; \ldots \;\; U \vdash t_n : \sigma_n \;\; \sigma_n \leq \tau_n\Theta}{U \vdash p(t_1,\ldots,t_n) \;\; Atom_{\tau_1 \ldots \tau_n}}$	Θ is a type substitution, $\tau_1 \ldots \tau_n \in types(p)$
(Head)	$\dfrac{U \vdash t_1 : \sigma_1 \;\; \sigma_1 \leq \tau_1\Theta \;\; \ldots \;\; U \vdash t_n : \sigma_n \;\; \sigma_n \leq \tau_n\Theta}{U \vdash p(t_1,\ldots,t_n) \;\; Head_{\tau_1 \ldots \tau_n}}$	Θ is a renaming substitution, $\tau_1 \ldots \tau_n \in types(p)$
(Query)	$\dfrac{U \vdash A_1 \;\; Atom_{\tau_1^1 \ldots \tau_{m_1}^1} \;\; \ldots \;\; U \vdash A_n \;\; Atom_{\tau_1^n \ldots \tau_{m_n}^n}}{U \vdash A_1,\ldots,A_n \;\; Query}$	
(Clause)	$\dfrac{U \vdash Q \;\; Query \quad U \vdash A \;\; Head_{\tau_1 \ldots \tau_n}}{U \vdash A \leftarrow Q \;\; Clause_{\tau_1 \ldots \tau_n}}$	

Table 1. The overloaded TCLP typing rules.

A program is well typed if, for all clauses $p(t_1, \ldots, t_n) \leftarrow Q$, for all $\tau_1 \ldots \tau_n \in types(p)$, there exists a variable typing U such that $U \vdash p(t_1, \ldots, t_n) \leftarrow Q \; Clause_{\tau_1 \ldots \tau_n}$ can be derived from the rules in table 1.

The distinction between rules *Head* and *Atom* expresses the usual *definitional genericity* principle [11] which states that the type of a defining occurrence of a predicate (i.e. at the left of "$\leftarrow$" in a clause) must be equivalent up-to renaming to the assigned type of the predicate. The rule *Head* used for deriving the type of the head of the clause is thus not allowed to use substitutions other than variable renamings in the declared type of the predicate. The definitional genericity condition is useful to the subject reduction properties of TCLP [6]. Below we extend the result to overloaded symbols.

Proposition 1. *For any variable typing U, any type judgement R other than a Head or a Clause, and any type substitution Θ, if $U \vdash R$ then $U\Theta \vdash R\Theta$.*

Proof. By induction on the height of the derivation tree for $U \vdash R$.

Theorem 1. *(Subject reduction for CSLD resolution)*
Let P be a well-typed CLP($\mathcal{H}$) program, and Q be a well-typed query, i.e. $U \vdash Q$ Query for some variable typing U. If Q' is a CSLD-resolvent of Q, then there is a variable typing U' such that $U' \vdash Q'$ Query.

Proof. Let us assume without loss of generality that $Q = c|p(s), \bar{A}$, and that Q' is a CSLD-resolvent of Q with the program clause $p(t) \leftarrow d|\bar{B}$. Thus we have $Q' = c, d, s = t|\bar{A}, \bar{B}$.

As Q is well typed, we have $U \vdash c|p(s), \bar{A}$ *Query*. In particular, we have $U \vdash p(s)$ $Atom_\tau$ for some type τ. And as the program is well typed, there exists a variable typing U'', renamed appart from U, such that $U'' \vdash p(t) \leftarrow d|\bar{B}$ $Clause_\tau$. Since $U \vdash p(s)$ $Atom_\tau$, we have $U \vdash s : \tau\Theta$ for some substitution Θ.

Now let $U' = U \cup U''\Theta$. By proposition 1, we have $U''\Theta \vdash d|\bar{B}$ *Query*, thus $U' \vdash c, d, |\bar{A}, \bar{B}$ *Query*. What remains to be shown is $U' \vdash s = t$ $Atom_{\alpha\times\alpha}$.

Since $U'' \vdash p(t)$ $Head_\tau$, we have $U'' \vdash t : \tau$. Hence, by proposition 1, $U''\Theta \vdash t : \tau\Theta$. Therefore, we have $U \vdash s : \tau\Theta$ and $U''\Theta \vdash t : \tau\Theta$, from which we conclude $U' \vdash s = t$ $Atom_{\alpha\times\alpha}$.

Without overloading, the TCLP typing rules are deterministic, i.e. the syntax of the expression at hand determines the rules to apply. Therefore type checking in TCLP basically amounts to collecting all subtype inequalities along a derivation of the expression, and checking their satisfiability in the structure of types. We refer to [6] for more details on the type checking algorithm and to section 3 for the solving of subtyping constraints.

2.2 Overloading by backtracking under the Andorra principle

Overloaded symbols make the typing rules non deterministic as they may have several types, i.e. different possible types for their arguments or their result. The Andorra principle, introduced for the parallelization of Prolog one decade ago [4], is the well-known principle that consists in delaying the execution of choice points until the time where all deterministic goals have been executed. We shall see that this simple control strategy, at the heart of constraint programming, is sufficient to efficiently handle overloaded symbols in TCLP. The idea is that the context of an expression containing overloaded symbols usually provides sufficient information to disambiguate the type of overloaded symbols. Hence, by simply delaying choice points, the type information coming from the context suffices to determine the type of overloaded symbols. If this is not sufficient, then the different types can be enumerated by backtracking under the Andorra principle and some simple principle of looking ahead. The algorithm for enumerating the types of overloaded symbols proceeds as follows:

1. the types of overloaded symbol occurrences are checked w.r.t. the current store and all unfeasible types are eliminated, if all types of a symbol occurrence are eliminated it is a failure,
2. if some symbol occurrence has only one type left, the corresponding subtyping constraints are added to the store,
3. the first two steps are iterated until a fixpoint is reached.
4. then a non resolved symbol is chosen, and each possible type is tried by iterating the whole procedure by backtracking.

The first two steps are particularly efficient as they often let the context disambiguate the type of overloaded symbols. The result of this treatment of overloading in type checking mode is a success if one typing makes the subtyping constraints solvable, and a failure if all types fail. The result in type inference mode is an enumeration of inferred types.

2.3 TCLP type inference

In a prescriptive type system, type reconstruction algorithms are useful to omit type declarations in programs, and still check the typability of the program by the possibility or not to infer the omitted types [11]. In TCLP, a predicate can accept any argument of a type below the declared type of the predicate. Therefore when inferencing the type of a predicate from the clauses defining the predicate, it is always possible to infer the most general, yet not informative, type. In particular if there is a type *term* which a supertype of any type, the typing of any predicate with type *term* is always a valid typing.

For these reasons, type inference in TCLP is heuristic. First, a minimum type is inferred for the predicate according to the type of the arguments found in the defining clauses of the predicate. Then a heuristic type is inferred according to the bounds of the types of the arguments found in the defining clauses. That heuristic type is finally made polymorphic by trying to replace unbounded types by type parameters. We refer to [6] for the details of the TCLP type inference algorithm.

When adding overloading, the TCLP type inference algorithm becomes non deterministic. Several inferred types can be enumerated according to the different types for the overloaded symbols. In the experimental results reported below, only the first inferred type is considered.

3 Solving subtype inequalities

The hard part of the TCLP implementation is in the solving of subtype inequalities. Only recently algorithms have been found for solving subtype relations between symbols of different arities, like $list(\alpha] \leq term$, and their decidability in type structures more general than lattices is still an open problem [7].

The solving of subtype inequalities is the following problem:
Input: a system of subtype relations $\bigwedge_{i=1}^{n} \tau_i \leq \tau'_i$

Output: does there exist a substitution Θ such that $\bigwedge_{i=1}^{n} \tau_i\Theta \leq \tau'_i\Theta$? A better output is to exhibit a minimal or maximal solution. A *minimal* (resp. *maximal*) is a solution Θ such that for any solution Θ' there exists a substitution Θ'' such that $\forall \alpha \in V\ \alpha\Theta\Theta'' \leq \alpha\Theta'$ (resp. $\forall \alpha \in V\ \alpha\Theta\Theta'' \geq \alpha\Theta'$).

3.1 TCLP types

TCLP deals with a structure of partially ordered terms, called poterms, for representing types with variables (parameters) and subtyping. For our purpose in this paper, it is convenient to make some changes in the assumptions described for TCLP in [6]. Here it will be simpler to first consider the solving of subtyping inequalities over infinite (regular) types instead of finite types. Proposition 4 below shows that in the context of TCLP type inference the solving over finite types is equivalent to solving over infinite types. Moreover we shall assume that the set of types ordered by the subtyping relation is a lattice. On the other hand, we shall relax the "arity decreasing" assumption made in [6].

Let $\mathcal{K}$ be a finite set of type *constructors* containing the symbols $\bot$ and $\top$. With each symbol $K \in \mathcal{K}$, an arity $m \geq 0$ is associated, the symbol with its arity is noted K/m. Let $\mathcal{U}$ be a countable set of *type variables*, also called *parameters*, denoted by $\alpha,\ \beta, ...$ An *infinite type* τ is an infinite term formed over $\mathcal{K}$ and $\mathcal{U}$, i.e. a partial function from strings of integers to symbols, $\tau : (\mathcal{N}^+)^* \to \mathcal{K} \cup \mathcal{U}$, such that i) $\mathrm{dom}(\tau)$ is non-empty and prefix-closed, ii) if $\tau(w) = K/n \in \mathcal{K}$ then $\{w0, ..., wn\} \subset dom(\tau)$ iii) if $\tau(w) = \alpha \in \mathcal{U}$ then $wi \notin \mathrm{dom}(\tau)$ for any $i \in \mathcal{N}^+$. The subterm of τ at $w \in \mathrm{dom}(\tau)$ is the type $\tau/w = \lambda w'.\tau(ww')$. An infinite type is regular if it contains a finite number of subterms. A finite type is a type with a finite domain. We denote $\mathcal{T}$ the set of regular types over $\mathcal{K}$ and $\mathcal{U}$.

The set of type variables in a type τ is denoted by $V(\tau)$. The set of ground types $\mathcal{G}$ is the set of regular types containing no variable. A *flat type* is a finite type of the form $K(\alpha_1, \ldots, \alpha_m)$, where $K \in \mathcal{K}$ and the α_i are distinct parameters.

Now, an order $\leq_{\mathcal{K}}$ is assumed on type constructors such that $(\mathcal{K}, \leq_{\mathcal{K}}, \bot, \top)$ forms a lattice. Moreover, we assume that with each pair $K/m \leq_{\mathcal{K}} K'/m'$, a partial injective mapping between arguments $\iota_{K,K'} : \{1, \ldots, m\} \to \{1, \ldots, m'\}$ is associated such that $\iota_{K,K''} = \iota_{K,K'} \circ \iota_{K',K''}$ whenever $K \leq_{\mathcal{K}} K' \leq_{\mathcal{K}} K''$. These assumptions mean that the arguments of comparable constructors are mapped consistently with $\leq_{\mathcal{K}}$. We also assume that if $K''/n = glb(K, K')$ then $\mathrm{dom}(\iota_{K'',K}) \cup \mathrm{dom}(\iota_{K'',K'}) = [1, n]$, that is greatest lower bounds do not introduce new parameters. Similarly, if $K''/n = lub(K, K')$ then $\mathrm{range}(\iota_{K,K''}) \cup \mathrm{range}(\iota_{K',K''}) = [1, n]$. The order on type constructors is extended to a *covariant subtyping order* $\leq$ on infinite types. The order $\leq$ is defined as the intersection of the following preorders:

- $\leq_0 = \mathcal{T} \times \mathcal{T}$,
- for any $k \in \mathcal{N}$, let $\tau \leq_{k+1} \tau'$ holds if and only if
  - either $\tau, \tau' \in \mathcal{U}$ and $\tau = \tau'$
  - or $\tau(\epsilon) \leq_{\mathcal{K}} \tau'(\epsilon)$ and $\forall i \in dom(\iota(\tau(\epsilon), \tau'(\epsilon)))\ \tau/i \leq_k \tau'/\iota(\tau(\epsilon), \tau'(\epsilon))(i)$
- $\leq = \bigcap_{k \in \mathcal{N}^+} \leq_k$.

One can check that $\leq$ is an ordering relation and that on ground types, $(\mathcal{G}, \leq, \bot, \top)$ forms a lattice [15].

Contravariant type constructors, where the ordering relation for comparing some of their arguments is reversed, are not considered in this paper. Therefore, if *int* $\leq_{\mathcal{K}}$ *float* for some basic types *int* and *float* then we have *list*(*int*) $\leq$ *list*(*float*) and *list*(*float*) $\not\leq$ *list*(*int*). We also have *list*(*float*) $\not\leq$ *list*(α) as the subtyping order does not include the instantiation preorder. Intuitively, a ground type represents a set of expressions, and the subtyping order between ground types corresponds to set inclusion. Parametric types do not directly support this interpretation, their parameters denote unknown types, like logical variables.

3.2 Solving subtype inequalities

We consider systems of subtype inequalities between variables and flat types, that is types of the form $\alpha \leq \beta$, $K(\alpha_1, ...\alpha_n) \leq \alpha$ or $\alpha \leq K(\alpha_1, ...\alpha_n)$. Non flat types can be represented in this form by introducing new variables and inequalities between these variables and the type they represent.

Proposition 2. *[1, 20] A system of subtype inequalities in a lattice of regular (or infinite) types is satisfiable iff it is decomposable with the following rules:*

Trans $\Sigma,\ \alpha \leq \beta,\ \beta \leq \gamma \longrightarrow \Sigma,\ \alpha \leq \beta,\ \beta \leq \gamma,\ \alpha \leq \gamma$
if $\alpha \leq \gamma \notin \Sigma$ and $\alpha \neq \gamma$.

Clash $\Sigma,\ K(\alpha_1, ..., \alpha_m) \leq \alpha,\ \alpha \leq \beta,\ \beta \leq K'(\alpha'_1, ..., \alpha'_n) \longrightarrow$ false
if $K \not\leq_{\mathcal{K}} K'$.

Dec $\Sigma,\ K(\alpha_1, ..., \alpha_m) \leq \alpha,\ \alpha \leq \beta, \beta \leq K'(\alpha'_1, ..., \alpha'_n) \longrightarrow$
$\Sigma,\ K(\alpha_1, ..., \alpha_m) \leq \alpha,\ \alpha \leq \beta, \beta \leq K'(\alpha'_1, ..., \alpha'_n),\ \{\alpha_i \leq \alpha'_{\iota(i)}\}_{i \in \mathrm{dom}(\iota)}$
if $K \leq_{\mathcal{K}} K'$, $\iota = \iota_{K,K'}$ and $\{\alpha_i \leq \alpha'_{\iota(i)}\}_{i \in \mathrm{dom}(\iota)} \not\subset \Sigma \cup \{\alpha \leq \beta\}$.

Exhibiting minimal and maximal solutions necessitates some extra work. For the sake of presentation, we assume that the initial system to be solved, Σ_0 over variables V_0, is first completed by introducing new variables γ_S and δ_S for each non empty subset S of V_0, and by adding the inequalities $\gamma_S \leq \alpha$ and $\alpha \leq \delta_S$ for all variables $\alpha \in S$. We also assume that the system is completed by adding the inequality $\alpha \leq \alpha$ for each variable α. Given a system Σ and a set of variables S we define the variable $\gamma(S, \Sigma) = \gamma_{\{\alpha \in V_0 \mid \exists \beta \in S\ \ \beta \leq \alpha \in \Sigma\}}$ and similarly $\delta(S, \Sigma) = \delta_{\{\alpha \in V_0 \mid \exists \beta \in S\ \ \alpha \leq \beta \in \Sigma\}}$.

Proposition 3. *[15] In a system of subtype inequalities simplified with the additional rules below, the identification of all parameters to their lower bound lb(α) (resp. upper bound ub(α)) provides a minimum solution (resp. maximum solution).*

(Glb) $\Sigma,\ \alpha \leq K(\alpha_1,...,\alpha_m),\ \alpha \leq \beta,\ \beta \leq K'(\alpha'_1,...,\alpha'_n) \longrightarrow$
$\Sigma,\ \alpha \leq K''(\alpha''_1,...,\alpha''_l),\ \alpha \leq \beta,\ \beta \leq K'(\alpha'_1,...,\alpha'_n),\ \Sigma'$
if $K'' \neq K$ or $\Sigma' \not\subset \Sigma \cup \{\alpha \leq \beta\}$,
where $K'' = \mathrm{glb}(K, K')$, $\iota = \iota_{K'',K},\ \iota' = \iota_{K'',K'}$,
$\alpha''_k = \gamma(\{\alpha_{\iota(k)}, \alpha'_{\iota'(k)}\}, \Sigma \cup \{\alpha \leq \beta\})$ for all $1 \leq k \leq l$,
$\Sigma' = \{\alpha''_k \leq \alpha_{\iota(k)}\}_{k \in \mathrm{dom}(\iota)} \cup \{\alpha''_k \leq \alpha'_{\iota'(k)}\}_{k \in \mathrm{dom}(\iota')}$

(Lub) $\Sigma,\ K(\alpha_1,...,\alpha_m) \leq \alpha,\ \alpha \leq \beta,\ K'(\alpha'_1,...,\alpha'_n) \leq \beta \longrightarrow \ldots$
$\Sigma,\ K(\alpha_1,...,\alpha_m) \leq \alpha,\ \alpha \leq \beta,\ K''(\alpha''_1,...,\alpha''_l) \leq \beta,\ \Sigma'$
if $K'' \neq K'$ or $\Sigma' \not\subset \Sigma \cup \{\alpha \leq \beta\}$,
where $K'' = \mathrm{lub}(K, K')$, $\iota = \iota_{K,K''},\ \iota' = \iota_{K',K''}$,
$\alpha''_k = \delta(\{\alpha_{\iota^{-1}(k)}, \alpha'_{\iota'^{-1}(k)}\}, \Sigma \cup \{\alpha \leq \beta\})$ for all $1 \leq k \leq l$,
$\Sigma' = \{\alpha_i \leq \alpha''_{\iota(i)}\}_{i \in \mathrm{dom}(\iota)} \cup \{\alpha'_j \leq \alpha''_{\iota'(j)}\}_{j \in \mathrm{dom}(\iota')}$

A system of subtype inequalities Σ is *acyclic* if there exists a ranking function on type variables $r : \mathcal{U} \to \mathcal{N}$ such that if $\sigma \leq \tau \in \Sigma$, $\alpha \in V(\sigma)$ and $\beta \in V(\tau)$ then $r(\alpha) < r(\beta)$. In [6] it is shown that the systems of subtype inequalities for TCLP type checking and type inference are acyclic, moreover:

Proposition 4. *[6] An acyclic system of inequalities is satisfiable over finite types if and only if it is satisfiable over regular types.*

The simplification rules given in this section are at the heart of TCLP algorithms for type checking and type inference. The next section describes their implementation in the Constraint Handling Rules language CHR [8].

4 Implementation of TCLP in CHR

4.1 Representing the subtype lattice

The subtype lattice can be described with three predicates `tclp_le(T1,T2)` `tclp_glb(T1,T2,GLB)` and `tclp_lub(T1,T2,LUB)`, for defining subtyping relations between type constructors, greatest lower bounds and lowest upper bounds respectively. These predicates can use the constraints `:<`,`tclp_vGLB` and `tclp_vLUB` defined in the following sections for expressing subtyping constraints on arguments. The user enters high level descriptions of the order and the program generates the dynamic clauses for `tclp_le/2`, `tclp_glb/3` and `tclp_lub/3`, corresponding to these declarations. For example, the following type declarations with the (implicit) subtype relations:

```
:- type int.
:- type list(A).
:- order int < term.
:- order list(A) < term.
```

generates the following clauses (the clauses for `tclp_lub` are symmetrical) :

```
tclp__le(list(_),term).
tclp__le(list(T1),list(T2)) :- T1 :< T2.
tclp__glb(int,term,int).
tclp__glb(term,list(T),list(T)).
tclp__glb(list(T1),list(T2),list(T3)) :- tclp__vGLB(T1,T2,T3).
```

4.2 Representing type variables

The set of simplification rules given in section 3 could be translated quite directly in CHR. However, for efficiency reasons, it is preferable to introduce for each type variable α a data structure `tclp__parameter(A,UB,USet,LSet,LB)` which encapsulates its current upper and lower bounds $ub(\alpha)$, $lb(\alpha)$, and the list $USet$ (resp. $LSet$) of type variables in the right hand side (resp. left hand side) of an inequality with α in the system. The `tclp__update` (`loset`, `hiset`, `lobound`) constraints are used to trigger changes in `tclp__parameter`.

```
tclp__update_hibound(X,Hibound) ,
tclp__parameter(X, HiboundX, HisetX, LosetX, LoboundX)
       <=> tclp__glb(HiboundX, Hibound, NewHibound),
           tclp__parameter(X, NewHibound, HisetX, LosetX, LoboundX).
tclp__update_hiset(X, Hiset) ,
tclp__parameter(X, HiboundX, HisetX, LosetX, LoboundX)
       <=> list_to_ord_set(HisetX, SHisetX),
           ord_union(Hiset, SHisetX, NewHiset),
           tclp__parameter(X, HiboundX, NewHiset, LosetX, LoboundX).
tclp__update_hiset_s([X|L],Hiset) :- tclp__update_hiset(X, Hiset),
                                     tclp__update_hiset_s(L, Hiset).
```

4.3 Type inequalities

The constraint `X :< Y` represents a subtyping constraint between `X` and `Y`. The predicates `tclp__transup` and `tclp__transinf` transform non flat types into flat ones.

```
X :< Y <=> var(X),var(Y) | tclp__var_ineq(X,Y).
X :< Y <=> nonvar(X),var(Y) | tclp__transinf(X,XFlat) , tclp__lo(XFlat,Y).
X :< Y <=> var(X), nonvar(Y) | tclp__transup(Y,YFlat) , tclp__hi(X,YFlat).
tclp__var_ineq(X,X) <=> true.
tclp__parameter(X, HiboundX, HisetX, LosetX, LoboundX),
tclp__parameter(Y, HiboundY, HisetY, LosetY, LoboundY) \
tclp__var_ineq(X,Y) <=> true |
           list_to_ord_set(HisetX,SHisetX),
           ( ord_member(Y,SHisetX) -> true ;
             ( list_to_ord_set(LosetX, SLosetX),
               ord_add_element(SLosetX,X,Loset),
               list_to_ord_set(HisetY,SHisetY),
               ord_add_element(SHisetY,Y,Hiset),
               tclp__update_hiset_s(Loset, Hiset),
               tclp__update_loset_s(Hiset, Loset),
               tclp__update_hibound_s(Loset, HiboundY),
               tclp__update_lobound_s(Hiset, LoboundX),
               tclp__le(LoboundX, HiboundY) ) ).
tclp__parameter(X, HiboundX, HisetX, LosetX, LoboundX) \
tclp__hi(X,Hibound) <=> tclp__update_hibound_s([ X | LosetX ], Hibound),
                        tclp__le(LoboundX,NewHiboundX).
```

There is also a rule to treat the case where a type variable has two constraints `tclp__parameter`, which happens when one unifies two type variables. The rule is similar to the one for treating an inequality between two variables.

4.4 Computing GLBs and LUBs

The computation of the GLB (resp. LUB) of two flat types is done with declarations `tclp__glb` (resp. `tclp__glb`). Below we describe the computation of greatest lower bounds of two type variables with the predicate `tclp__vGLB`. The `tclp__vLUB` for least upper bounds is symmetrical.

To compute the GLB of `X` and `Y`, we distinguish four cases according to whether `X` and `Y` are original or introduced type variables. Introduced variables are recognized by the fact that they are introduced with a constraint of the form `tclp__original_up(X,Origs)`, where `Origs` is the set of original variables above introduced variable `X`.

```
tclp__original_up(X,OrigX), tclp__original_up(Y,OrigY) \ tclp__vGLB(X,Y,GLB)
       <=> list_to_ord_set(OrigX,SOrigX), list_to_ord_set(OrigY,SOrigY),
           ord_union(SOrigX,SOrigY,OrigGLB),
           tclp__GLBVar(OrigGLB,GLB).
tclp__original_up(X,OrigX) \ tclp__vGLB(X,Y,GLB)
       <=> list_to_ord_set(OrigX,SOrigX),
           ord_add_element(SOrigX,Y,OrigGLB),
tclp__original_up(X,OrigX) \ tclp__vGLB(Y,X,GLB) <=> tclp__vGLB(X,Y,GLB).
tclp__vGLB(X,Y,GLB) <=> list_to_ord_set([X,Y],OrigGLB),
                        tclp__GLBVar(OrigGLB,GLB).
tclp__GLBVar(OrigGLB,GLB) :-
        chr:findall_constraints(tclp__original_up(_,_), AllOrigs),
        ( find_VAR(AllOrigs, OrigGLB, GLB) -> true;
          ( tclp__original_down(GLB,[]),
            tclp__original_up(GLB,OrigGLB),
            tclp__parameter(GLB,term,[],[],bottom),
            tclp__link_up(GLB,OrigGLB) ) ).
```

The predicate `tclp__GLBVar(OrigGLB,GLB)` is true when `GLB` is the type variable introduced for the set of original variables `OrigGLB`. The predicate `find_VAR(AllOrigs, Origs, Var)`, where `AllOrigs` is the list of all constraints of the form `tclp__original_up` and `Origs` is a set of original variables, looks up `Origs` in `AllOrigs` and unifies `Var` with the corresponding variable in the constraint `tclp__original_up`. Otherwise it fails, which means that no type variable was introduced for the set `Origs`. The predicate `tclp__link_up(Var,List)` puts the constraint `Var :< X` for all `X`s in `List`.

4.5 Overloading

The algorithm of section 2.2 for solving overloaded symbols is implemented in CHR. Occurrences of overloaded symbols are given an unknown type of the form $\alpha_1 \times ... \times \alpha_n \rightarrow \alpha$, called an abstract type scheme, on which type checking

constraints are accumulated. The following rule *reduce* basically implements the filtering step 1 of the algorithm, and may raise the *failure* rule or the *instanciate* rule (step 2). The labeling rule (step 4) is not detailed.

```
reduce @ abstract_type(Id, AbstractScheme) \
   multi_type(Id,ConcreteSchemes), do_reduce(N,Total)
   <=> filter_schemes(AbstractScheme, ConcreteSchemes, NewConcreteSchemes),
       multi_type(Id, NewConcreteSchemes),
       ( length(ConcreteSchemes, Length), length(NewConcreteSchemes, Length)
       -> N1 is N+1, do_reduce(N1,Total)
       ; do_reduce(N,Total) ).
failure @ abstract_type(Id, _), multi_type(Id,[]) <=> fail.
instanciate @ multi_type(Id,[ ConcreteScheme ]),
              abstract_type(Id, AbstractScheme)
              <=> apply_scheme(ConcreteScheme, AbstractScheme).
labeling @ label(Id), multi_type(Id, Types),
           abstract_type(Id, AbstractScheme, F/N, Location)
           <=> apply_one_scheme(AbstractScheme,Types),
               reduce_all(Remaining),
              (Remaining=0 -> true; label_functor).
```

5 TCLP types for ISO Prolog

5.1 Type structure

Figure 5.1 depicts the TCLP type structure we propose for ISO Prolog. This type structure is completed in a lattice by adding a bottom type $\perp$ below all types. This type $\perp$ is an empty type and is thus considered as an error type in TCLP [6].

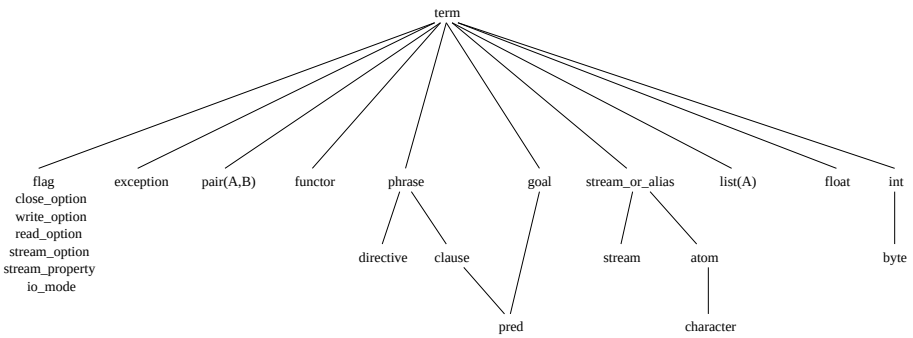

Fig. 1. TCLP type structure for ISO Prolog.

Metaprogramming predicates in ISO prolog basically impose that every object can be decomposed as a term. This is treated in TCLP by subtyping with a type *term* at the top of the lattice of types. Since bytes are integers with a restricted value, we have $byte < int$. However we do not have a subtype relation between $int \not< float$. This choice is motivated by the fact that, in ISO Prolog, there is no implicit coercion from integers to floats, in particular a unification like `1=1.0` fails in ISO Prolog. Another subtype relation is introduced for allowing coercions from characters to atoms. The subtyping relations between *stream*, *atom* and *stream_or_alias* are motivated by the fact that a stream can be replaced by an alias, i.e. an atom. The type *pred* is the type of predicates, that is heads of clauses as well as occurrences in the body of a clause or in a directive. For this reason, predicates can be viewed both with the type *clause* (for clauses with an empty body) and with type *goal*. This justifies the relations $pred < clause$ and $pred < goal$.

Parametric types are introduced for lists, with type $list(\alpha)$, and for pairs, with type $pair(\alpha, \beta)$. We thus have subtype relations $list(\alpha) < term$ and $pair(\alpha, \beta) < term$, i.e. subtype relations between type constructors of different arities which is responsible for the difficulty of subtype constraint solving in TCLP.

5.2 Metaprogramming and subtyping

The type *term* is used for term manipulation predicates, and can be used to decompose any object, thanks to the subtype relation between any type and *term*. Term manipulation predicates thus have the following types in TCLP: `functor` : $term \times atom \times int \rightarrow pred$, `arg` : $int \times term \times term \rightarrow pred$, `=..` : $term \times list(term) \rightarrow pred$, `copy_term` : $\alpha \times \alpha \rightarrow pred$. Note that, as *term* is the top element of the type lattice, each occurence of *term* in the type of a predicate can be equivalently replaced by a fresh type variable, e.g. `arg` : $int \times \alpha \times \beta \rightarrow pred$.

The type *clause* provides the possibility to type check dynamic predicate declarations using the following type declarations: `clause` : $pred \times goal \rightarrow pred$, `asserta` : $clause \rightarrow pred$, `assertz` : $clause \rightarrow pred$, `retract` : $clause \rightarrow pred$, `abolish` : $functor \rightarrow pred$. One should ckeck however that the head condition (see section2) is satisfied in order to ensure the property of subject reduction at run-time [6]. The symbol $:-$ for constructing clauses is thus type checked in TCLP with a special rule that checks that its first argument is a *Head* not just an *Atom*, according to table 1.

The dynamic types of objects can be tested with ISO Prolog predicates `var/1`, `atom/1`, `integer/1`, `float/1`, `atomic/1`, `compound/1`, `nonvar/1` and `number/1`. These predicates are typed in TCLP with type $term \rightarrow pred$, as they are applicable to any ISO Prolog object. This marks a fundamental difference between a prescriptive type system like TCLP and a descriptive type system which would instead type `float` : $float \rightarrow pred$, as descriptive types are in fact an approximation of the success set.

5.3 Arithmetic and overloading

Typing arithmetic expressions involves a large amount of overloading, because of the interaction between *int* and *float*. We thus have the following types for arithmetic operations: `+,-,*` : $int \times int \rightarrow int$ and `+,-,*,/` : $float \times int \rightarrow float$, $int \times float \rightarrow float$ and $float \times float \rightarrow float$.

Note that the ordering of the rules is important as far as only the first inferred type is considered as the result of the type inference algorithm. Note also that pairs in ISO Prolog are constructed with the same symbol minus `-/2`. The type `-` : $\alpha \times \beta \rightarrow pair(\alpha, \beta)$ is thus also added to the types of `-/2`.

The experimental results reported below show that, despite the combinatorial nature of these overloaded type declarations, the handling of overloading in TCLP does not produce a combinatorial explosion and remains efficient in practical programs.

5.4 Options

Many system predicates in ISO Prolog come with a set of terms describing either properties or options, e.g. `open/3` comes with `read`, `write` and `append`. We choose to associate a type to each set of options, that gives precise typings, e.g. `open` : $atom \times io\_mode \times stream \rightarrow pred$. We use subtyping when a set of options is completely included in another set of options, e.g. $stream\_option < stream\_property$, otherwise we use overloading.

Many options are atomic and are thus overloaded with type *atom*. In the previous version of TCLP without overloading, a common subtype between options and the type *atom* was introduced, but since these types have different uses, overloading is preferred.

6 TCLP types for CLP($\mathcal{R}$,$\mathcal{Q}$,$\mathcal{FD}$,$\mathcal{B}$)

In TCLP, the constraint domain of CLP($\mathcal{R}$) can be typed with the same type *float* as Prolog. Similarly the constraint domain of CLP($\mathcal{FD}$) can be typed with type *int*. The boolean domain of CLP($\mathcal{B}$) is a subset composed of values 0 and 1 of the finite domain $\mathcal{FD}$ composed of values 0 and 1. A new domain type *boolean* is thus introduced as a subtype of *int*, $boolean < int$. In CLP($\mathcal{Q}$) the constraint domain of rational numbers with infinite precision is typed a special type *rat*.

The inferred TCLP types for classical CLP($\mathcal{FD}$) examples are the expected ones, like `queens` : $int \times list(int) \rightarrow pred$ etc. On the other hand, on many CLP($\mathcal{R}$) examples, the first inferred type is *int* instead of *float*, since in these examples the arithmetic expressions involve simple operations with integer constants only.

7 Evaluation

File	Without overloading					With overloading		
	Type Checking		Type Inference		%exact	Type Check	Type Inf.	%exact
	CAML	CHR	CAML	CHR		CHR	CHR	
arrays.pl	2.2 s	2.1 s	11.9 s	3.9 s	23%	2.5 s	3.2 s	68%
assoc.pl	5.3 s	6.0 s	40.1 s	13.6 s	68%	5.2 s	13.5 s	91%
atts.pl	7.4 s	5.5 s	77.5 s	12.4 s	64%	6.4 s	15.8 s	91%
bdb.pl	23.6 s	20.2 s	41.1 s	17.4 s	64%	16.1 s	21.7 s	66%
charsio.pl	1.3 s	1.0 s	2.4 s	1.3 s	33%	0.8 s	3.8 s	74%
clpb.pl	24.3 s	22.7 s	1827.3 s	224.8 s	n/a	18.4 s	204.9 s	n/a
clpr.pl	304.45 s	445.1 s	3958.41 s	566 s	n/a	n/a	n/a	n/a
fastrw.pl	0.4 s	0.5 s	0.7 s	0.7 s	100%	0.4 s	0.6 s	100%
heaps.pl	3.5 s	4.2 s	43.3 s	17.4 s	71%	3.5 s	16.4 s	97%
jasper.pl	7.4 s	2.7 s	12.0 s	3.9 s	84%	2.3 s	3.0 s	84%
lists.pl	3.5 s	3.8 s	16.2 s	6.6 s	98%	3.5 s	7.6 s	98%
ordsets.pl	4.1 s	5.2 s	199.4 s	44.8 s	97%	4.1 s	49.2 s	97%
queues.pl	0.6 s	0.7 s	4.1 s	1.5 s	75%	0.6 s	1.3 s	96%
sockets.pl	6.8 s	3.9 s	15.4 s	5.3 s	68%	3.0 s	4.3 s	92%
random.pl	0.9 s	1.0 s	4.1 s	1.0 s	55%	0.9 s	0.9 s	58%
terms.pl	2.5 s	2.6 s	308.7 s	4.3 s	77%	2.5 s	4.4 s	77%
trees.pl	1.4 s	1.6 s	12.6 s	3.2 s	31%	1.4 s	3.0 s	75%
ugraphs.pl	48.2 s	25.3 s	274.2 s	353.5 s	67%	21.1 s	350.2 s	67%
clpfd.pl	24.3 s	34.8 s	59.6 s	154.0 s	n/a	33.1 s	140.1 s	n/a

Table 2. Performance on Sicstus Prolog libraries.

We compare the performances of two versions of TCLP. The first one, coded in Objective Caml, uses the subtyping constraint solving library Wallace [14] by F.Pottier. The second one, coded in Sicstus Prolog, uses the CHR implementation subtyping constraints described in section 4. For the latter implementation, we also compare the typings with and without overloading.

The benchmarks are composed of 20 Sicstus Prolog libraries and of a Prolog implementation of CLP($\mathcal{FD}$). The first column gives the CPU time for type checking of both versions in CAML and CHR. The second column gives the CPU time for type inference. The third column indicates the percentile of inferred types which are identical to the (authors') intended types. The last three columns display these results for the CHR implementation using a different type structure and type declarations with overloading. This allows us to estimate the impact of overloading both in terms of runtime efficiency and in terms of the performance of the heuristics used for inferring types.

The significant increase of the percentile of exact types inferred with overloading can be explained by the more precise typings provided by overloaded type declarations. In particular for arithmetic, in the version of TCLP without overloading, the typing with *float* was always inferred, whereas in the version of TCLP with overloading, the typing with $\int$ is inferred when possible. The remaining differences between the heuristically inferred types and the intended types in some examples are mainly due, on the one hand, to the permissive typing of equality `=/2` : $\alpha \times \alpha \rightarrow$ *pred* which, when instanciated with type *term*,

does not provide communication between the types of its arguments [6], and on the other hand, to the fact that only the first inferred type is considered.

One can notice that the times for type checking (resp. type inference) are close whenever they are done with or without overloading. On the other hand, although the type checking times between CAML and CHR implementations are close, the CHR implementation runs significantly faster for type inference. The gain of efficiency on the CHR version of TCLP is explained by the capability of the CHR subtyping solver to *unify type variables*, while the CAML implementation does not perform such unifications. When two type variables $T1$ and $T2$ have to be unified, the CAML implementation adds the inequalities $T2 \leq T1$, $T1 \leq T2$ to the store. In CHR, unification is done by the rule `type_ident @ V::T1  V::T2 <=> T1=T2.` Since the complexity of Trifonov and Smith decomposition (rules in proposition 2) is $O(n^3)$, simplification by unification of type variables permits a significant speed-up on examples which contain several occurrences of a same type variable.

The benchmark results show also that the practical cost of overloading is low. This can be explained by the efficiency of the Andorra and looking ahead principles in this case and, for a smaller part, by the removal of some subtype relations from the type structure used with overloading.

8 Conclusion

The TCLP type system with overloading is a practical system for typing Prolog and constraint logic programs. We have shown that the addition of overloading to subtyping and parametric polymorphism is necessary to properly type arithmetic predicates, and to deal with some overloaded symbols like minus which denotes both subtraction and pairs in Prolog.

Type checking and type inference in TCLP involve the solving of complex subtype inequality constraints. We have described an implementation of Pottier's algorithm in CHR which surprisingly outperformed the original implementation in CAML, thanks to some simplifications by unification of type variables which are natural to implement in CHR.

In the new implementation of TCLP in Prolog and CHR, overloading is implemented by backtracking with the Andorra principle. We have shown that this simple strategy is very efficient on large programs such as the Sicstus Prolog implementation of CLP($\mathcal{R}$) for example. We have proposed TCLP types for ISO Prolog and constraint logic programs, and used these types for typing the Sicstus Prolog libraries and classical constraint logic programs.

As for future work, we plan to acquire more practical experience from the users of TCLP [3] and extend TCLP to other languages. We plan also to use the backtracking capabilities of the new Prolog-CHR implementation of type constraints to experiment the solving of subtype inequality constraints in more general structures than lattices (quasi-lattices, partial orders) for which the decidability of subtype constraint satisfaction is an open problem [7, 15].

References

1. R.M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
2. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.
3. E. Coquery. `Tclp`: a generic type checker for constraint logic programs, October 2000. `http://pauillac.inria.fr/~coquery/tclp/`.
4. 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.
5. B. Demoen, M. Garcia de la Banda, and P.J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of the 22nd Australian Computer Science Conference*, pages 217–228, january 1999.
6. F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1, November 2001.
7. A. Frey. Satisfying subtype inequalities in polynomial space. In *Proceedings of the 4th International Static Analysis Symposium SAS'97*, number 1302 in LNCS, 1997.
8. 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.
9. Y.C. Fuh and P. Mishra. Type inference with subtypes. In *Proc. ESOP'88*, number 300 in LNCS, pages 94–114, 1988.
10. P. Hill and J. Lloyd. *The Gödel programming language*. MIT Press, 1994.
11. 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.
12. 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.
13. A. Mycroft and R.A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23:295–307, 1984.
14. F. Pottier. `Wallace`: an efficient implementation of type inference with subtyping, February 2000. `http://pauillac.inria.fr/~fpottier/wallace/`.
15. F. Pottier. Simplifying subtyping constraints: a theory. *To appear in Information and Computation*, 2002.
16. 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.
17. 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.
18. J. Tiuryn. Subtype inequalities. In *Proc. 7th IEEE symposium on logic in computer science LICS'92*, pages 308–315, 1992.
19. J. Tiuryn and M. Wand. Type reconstruction with recursive types and atomic subtyping. In *Proceedings of the 22nd Australian Computer Science Conference*, pages 217–228, january 1999.

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.