Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 246 (2009) 183–198

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Extending Constructive Logic Negation with Types

Susana Munoz-Hernandez[1] ,[2]

*Babel Group,*
*Facultad de Informática, Universidad Politécnica de Madrid.*
*Campus de Montegancedo. Boadilla del Monte. Madrid-28660, Spain*

Juan José Moreno-Navarro [3]

*IMDEA-Software & Universidad Politécnica de Madrid.*
*Campus de Montegancedo. Boadilla del Monte. Madrid-28660, Spain*

**Abstract**

Negation has traditionally been a difficult issue in Logic Programming. Most of Prolog programmers have been restricted to use just a weak negation technique, like negation as failure.
Many alternative semantics were proposed for achieving constructive negation in the last 20 years, but no implementation was provided so far because of its exponential complexity and the difficulty for developing it. First effective implementations of constructive negation into standard Prolog compilers are available just recently, around 2003, provided by our previous works.
In this paper we present an extension of our implementations by introducing types in programs, thus improving usability as well as efficiency in some cases of our implementations of constructive negation. This can make constructive negation an interesting approach for its use in data bases querying, web search, filtered search, ontologies querying, coding rules, business rules, etc.
Thanks to the use of types, our constructive negation can provide concrete values as results, instead of constraints (as in our previous works). We provide details about the semantics and the implementation in our approaches of classical, finite constructive, and intensional negation. The paper also includes some practical examples additionally allowing for providing measurements of computational behavior.

*Keywords:* Logic Programming Implementation, Negation, Types, Constraint Logic Programming, Constructive Negation, Non-monotonic Reasoning.

## 1 Introduction

The beginning of logic is tied with that of scientific thinking. Its application for modeling human reasoning is clear as a programming language. But one of the main elements of logic, that is negation, is hardly represented in Logic Programming.

[1] This work is partially supported by the project DESAFIOS - TIN 2006-15660-C02-02 from the Spanish Ministry of Education and Science and project PROMESAS - S-0505/TIC/0407 from Comunidad de Madrid.

[2] Email: `susana@fi.upm.es`

[3] Email: `jjmoreno@fi.upm.es`

doi:10.1016/j.entcs.2009.07.022

### 1.1 Logic Programming and Negation

Negation is probably the most significant aspect of logic that was not included from the outset. Dealing with negation involves significant additional complexity. Nevertheless, the use of negation is very natural and plays an important role in many knowledge representation and reasoning systems, like web semantics, natural language processing, constraints management in databases, program composition, manipulation and transformation, coding rule checking, business rules, default reasoning, negative queries (search of false information), etc.

There are many ways of understanding and incorporating negation into Logic Programming, the problems really start at the semantic level, where the different proposals differ not only in the semantics but also as to expressiveness. Unfortunately, current Prolog[4] compilers support a very limited number of negation techniques: negation as failure under Fitting/Kunen semantics [9] (sound only under some circumstances usually not checked by compilers) which is a built-in in most Prolog compilers (Quintus, SICStus, Ciao, BinProlog, etc.), and the "delay technique" (applying negation as failure only *when* the variables of the negated goal become ground, which is sound but incomplete due to the possibility of floundering) which is present in Nu-Prolog, Gödel, and Prolog systems that implement delays (most of the above).

Among all proposals, constructive negation [5,21] (that we will call *classical* constructive negation) is probably one of the most promising because it has been proved to be sound and complete, and its semantics is fully compatible with the Prolog one. A previous paper [4] provided a simpler variant for negating goals that have a finite number of solutions (that we will call *finite* constructive negation). Another interesting approach, different to these ones, is the transformation proposed by Barbuti et all [2] that we will call *intensional* constructive negation. In the paper we will use these three approaches.

Attending to what we have expounded in this section, it is clear the interest for achieving a sound and complete implementation for these techniques. Constructive negation was, in fact, announced in early versions of the Eclipse Prolog compiler, but was removed from the latest releases. The reasons seem to be related to some technical problems with the use of coroutining (risk of floundering) and the management of constrained solutions. We are trying to fill a long time open gap in this area (remember that the original papers are from late 80s) facing the problem of providing a correct, effective and complete implementation, integrated into a standard Prolog compiler.

It was just during the last years [15,14] when we have provided effective implementations of some constructive negation techniques[5]. Here in this paper we improve the expressiveness and usability of our implementations of classical, finite and intensional constructive negation by including types.

[4] We understand Prolog as depth-first, left to right implementation of SLD resolution for Horn clause programs, ignoring, in principle, side effects, cuts, etc.

[5] More details about differences in between the constructive negation techniques can be found at [15,14]

### 1.2 Type Systems for Logic Programming

Introducing type systems for the checking of types in logic programming dates back to the initial papers of Mishra [13] and Mycroft-O'Keefe ([17]). Since then, there has been a number of proposals providing notions of types and typing in Logic Programming (see [18] and [11] for general surveys).

There is a distinction between the effects of typing in other programming languages (e.g. functional programming) and logic programming. In strongly-typed languages, *well-typed programs cannot go wrong* (in the sense of well-definiteness of expression [12]), while in logic programming *ill-typed programs will fail*.

Meyer [11] classifies the proposals in three classes:

- *Types for proving partial correctness*, i.e. basically providing static type checking.
- *Types as constraints*, type constraint enhance the expressiveness of the language and sometimes can be exploited in the implementation.
- *Types as approximations*. Type declarations can be seen as approximations for the set of atoms with are intended to be true. Declaration can express a necessary condition or even a sufficient condition.

The type language in a type system decides which sets of terms are types. To be useful, the set of types should be closed under set intersection, set union and set complement operations. The decision problems such as the emptiness of a type, inclusion of a type in another, and equivalence of two types should be decidable. Regular types, i.e. those described by regular term grammars, satisfy these conditions and have been widely used as types in Logic Programming.

It is also possible to distinguish the typing proposals from the semantics point of view: I) *Prescriptive typing*, where the semantics depend on clauses and type declarations, e.g. [1], II) *Descriptive typing*. Semantics are independent on type declarations. Deriving descriptive types from a program (this process is also called "type inference" or "type analysis"), essentially means finding, at compile-time, an approximate description of the values that program variables can take at run-time. Descriptive types can be inferred that approximate various semantics of a logic program: declarative semantics e.g. [6,22], or operational semantics e.g. [7,19].

Of course, it is worth mentioning also strongly-typed logic languages such as Gödel or Mercury. In both cases the programmer is required to declare the types for functions and predicates. Moreover, sub-typing is often not permitted. In contrast, in most of the proposals above based on regular types, writing type definitions is optional and sub-typing is allowed.

We have started this paper with an introduction to negation (and constructive negation) in Logic Programming. The remainder of the paper is organized as follows. Section 2 describes additional semantics foundations of our proposal. Section 3 discusses some implementation issues related to implement types in a Prolog system. Section 4 presents the structure of the prototypes implemented for classical, finite and intensional constructive negation with types. A set of illustrative examples are provided Section 5 comparing the original techniques with the modified techniques using types. And finally in section 6 we will provide some experimental results, conclusions and we will discuss briefly some future work.

## 2 Semantics Foundations

### 2.1 Regular Types and Type Declarations

Regular types can be defined by regular grammars. Terminal symbols are basic types: we introduce base types and corresponding base type symbols, like `int`, `num`, `char` etc., denoting respectively sets of integers, all numbers, characters, etc. Grammar rules generate combined types probably including constructors (maybe 0-ary, i.e. constants). Types can be defined in several syntactic manners. For our purpose we will restrict themselves to use Horn clauses, although they can be also automatically generated from other syntax. E.g. the type for list of int can be defined by

```
typeList ([]).
typeList ([X|L]) :- int (X), typeList (L).
```

that comes from the rule

$$typeList \longrightarrow [\;] \;\; ++ \;\; [int|typeList]$$

where we use the symbol ++ in between alternative constructors in a type declaration. For simplicity we do not deal with polymorphism, but it can be easily included by using additional parameters in the predicates defining types.

Type declarations for Prolog predicates can be made through "assertions". Assertions [20] can be used for including information on Prolog programs about intended or inferred calls and success patterns. There are several possible forms of assertions (entry, call states, correctness properties, ...) but for typing information "success" assertions are enough. Assertions (specifically what we call success assertions) are of the form:

$$\text{success} \;\; P(X) : \;\; Pre \Longrightarrow Post$$

$P(X)$ is a predicate descriptor, i.e., it has a predicate symbol as main functor and all arguments are distinct free variables, and $Pre$ and $Post$ are pre- and postconditions respectively. For our purposes it is sufficient to consider that $Pre$ and $Post$ correspond to formulas relating variables of $X$. The meaning of assertions is twofold. First, the precondition $Pre$ expresses properties which should hold in calls to $P$[6]. Second, the postcondition $Post$ expresses properties which should hold on termination of a successful computation of $P$, provided that $Pre$ holds on call.

Type declarations can be included as success assertions. For example:

```
:- success length(L,N) : => int (N), typeList(L).
```

[6] Usually preconditions are not used for typing, except when polymorphism is included.

### 2.2 Negating Predicates with Type Assertions

The distinction made above about prescriptive vs. descriptive typing could be overcome when a setting of type declarations by assertions is used. Assertions can be either included by the user (prescriptive typing) or detected by the compiler (by using static analysis and abstract interpretation techniques, like those reported in [6,22,7,19]).

Therefore, we assume that type assertions are part of the program. With respect to the semantics of a Prolog predicate $P$ with a type assertion:

:- success P ($\overline{X}$) : => *typedec*($\overline{X}$).
P ($\overline{t_1}$) :- B$_1$.
$\vdots$
P ($\overline{t_n}$) :- B$_n$.

it is equivalent to include the type constraints in the body of clauses:
P ($\overline{t_1}$) :- $\overline{X} = \overline{t_1}$, *typedec*($\overline{X}$), B$_1$.
$\vdots$
P ($\overline{t_n}$) :- $X = \overline{t_n}$, *typedec*($X$), B$_n$.

Notice that we are talking about equivalence in the declarative semantics, although the operational behavior of both programs can differ.

Remember that our final goal is to handle negation in logic programming. For this goal this semantic interpretation is quite adequate. The logical semantics of predicate $P$ is: $P(X) \longleftrightarrow typedec(X) \wedge \bigvee_1^n (X = \overline{t_i} \wedge B_i)$
so, when we negate it we get:
$\neg P(X) \longleftrightarrow \neg typedec(X) \vee (typedec(X) \wedge \neg \bigvee_1^n (X = t_i \wedge B_i))$

Therefore when we evaluate $\neg P$ we should obtain two kind of answers: i) type declaration are wrong, or ii) we can assume type declarations for negating clauses of $P$.

The first answer is irrelevant (can be skipped when typing is prescriptive) and the second one allows for implementing our constructive methods, using typing information for generating values for negating predicates. In this sense, we are using *types as constraints*.

## 3 Implementation Foundation

The Ciao system [8] is a programming environment for developing Prolog programs.

It is important to note that, in Ciao libraries there is a distinction between modules and packages. *Modules* are regular libraries that one can import from any Prolog program, their code (or the code of some of their predicates) is added to the code of the program that imports them. *Packages* are libraries that define a transformation function for the code of the program that imports them. This transformation function generates an output code that is a expanded program that

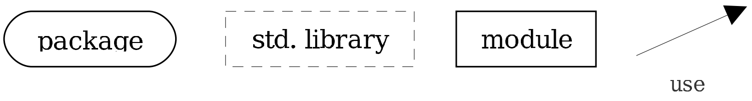

Fig. 1. Notation of packages and modules for figures.

can add clauses, remove clauses or modify the clauses of the original program. Our approach is a combination of built-in and own modules and packages. In Figure 1 we can see the notation that we use in the following sections (to represent packages, standard libraries, modules, and the relation that tells which packages/modules use to which other packages/modules).

For the interested reader, a complete implementation is provided at: `http://babel.ls.fi.upm.es/~susana/code/negation/types/`

It includes a complete CIAODE framework (in a file "CiaoDE-1.13.0-7228.tar") that contains the release 1.13 of Ciao Prolog, the release 1.2 of CiaoPP and the release 2.0.38 of LPDoc. The 21.4 release of Emacs for Linux ("emacs-21.4a.tar") is also available in the same address.

The types supported by Ciao are regular types. Nevertheless, we have had to develop some additional modules to provide constructivity to the type system. For this purpose we have used the assertion system.

Ciao type checking is implemented by the package `assertions` (see [20] and chapter 53 of the Ciao reference manual [3]). The compiler returns an error if the type of the value that is assigned to a variable does not correspond to the type declared for it. But there is no generation of valid values in case of looking for values for the variable. This is what we have to add to the implementation of assertions of Ciao Prolog.

Indeed, the Ciao compiler ignores the assertions. If we want to analyze types or detect errors we have to use the Ciao pre-processor (ciaopp). With our additional module for types we will use it directly with the compiler.

*success* assertions take this form in Ciao:

```
:-success Goal=>PostCond
```

saying that if a goal *Goal* is successful, then their arguments should be of the type declared in *PostCond*.

This type information can be used by the pre-compiler for analysis, but we are interested in using this information at run-time (execution time) by the compiler. That is why we need to incorporate it into the program. We achieve this with some new packages, *gen_types* and *intneg_types*, that we have implemented, which expand the original code adding for each type declaration a clause that contains the useful information. Particularly, the function *add_types*/3 generates a new predicate *pred_check_types*/$n$ for each predicate *pred*/$n$. We use these new predicates to check all types of the arguments of *pred*/$n$ according to the information that was provided in the original program using the *success* assertion.

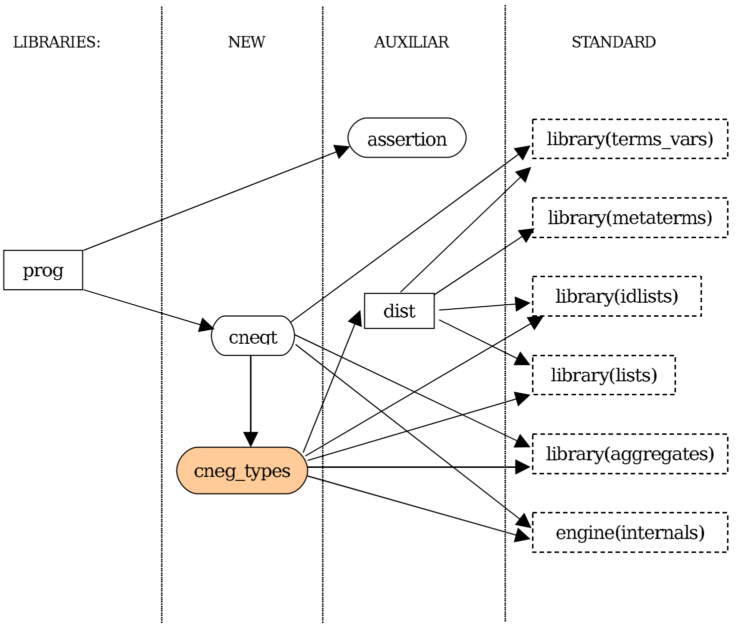

Fig. 2. Packages and modules that are used to implement classical constructive negation with types.

## 4 Constructive Negation Implementation with Types

### 4.1 Classical Constructive Negation

As we already said, one of the most interesting techniques for implementing constructive negation was proposed by Chan [5] and refined later [15]. We have taken the implementation provided by [15] (particularly the predicate *cneg*/1) as basis to develop a new predicate *cnegt*/1 that implements classical constructive negation with types. This predicate is implemented in the package *cnegt.pl* that also uses the package *gen_types*.

If a Prolog program, e.g. *prog.pl*, requires to use classical constructive negation with types, it just has to use the package *cnegt.pl* as well as the standard package *assertions.pl* by adding this line[7]:

```
:- module(prog,_,[assertions,.(cnegt)]).
```

In Figure 2 we can see the tree showing the dependencies of the packages and modules that are needed. We distinguish between new developed libraries, auxiliary libraries (already developed when implementing constructive negation) and the standard ones.

### 4.2 Finite Constructive Negation

Finite constructive negation [4] is a negation technique similar to the classical constructive negation but the frontier that is negated is the last one. It was the first

[7] It is Ciao modules syntax : $-module(module\_name, exported\_preds, imported\_packages)$. The "_" means that all predicates defined into the module *prog.pl* are exported and the notation .() means that the package *cnegt.pl* is in the same directory (path) that *prog.pl*.

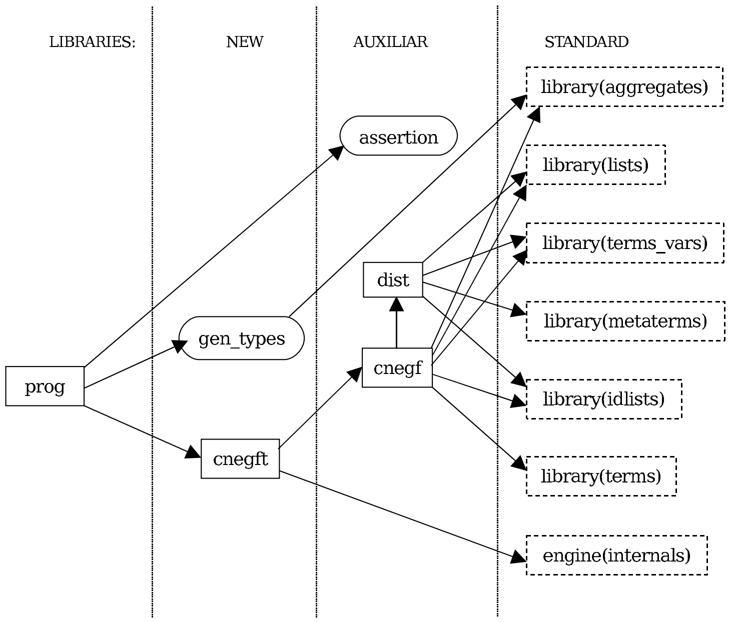

Fig. 3. Packages and modules which are used to implement the finite constructive negation with types.

proposal by Chan. That is, first all solutions are obtained and then the disjunction of all of them is negated. Obviously, this technique can only be used if the goal that is negated has a finite number of solutions. In our implementation we are sure about this fact by using finiteness analysis [16].

We have provided an implementation for this technique in [15] with the predicate *cnegf*/1. On top of this predicate we have defined a new predicate *cnegft*/1 that implements finite constructive negation with types. This predicate is implemented in the module *cnegft.pl*. Prolog program that use this kind of negation, should also use the package *gen_types* that we have also developed.

In order to be used by a Prolog program it has to import the predicate *cnegft*/1 from module *cnegft.pl* and the standard package *assertions.pl* and the package *gen_types.pl* that we have developed:

```
:- use_module(cnegft,[cnegft/1]).
:- use_package([assertions]).
:- use_package(.(gen_types)).
```

In Figure 3 we can see the tree of dependencies of the packages and modules that are needed. We differentiate again the source of libraries.

## 4.3 Intensional Constructive Negation

Intensional constructive negation [2]is a negation technique stemming from a different approach than the previous ones. It is based on a transformational approach of the source program to obtain a program that also includes the negative counterpart of each predicate. This transformation is performed at compilation time and execution is, in general (and in particular for complex goals), much more efficient

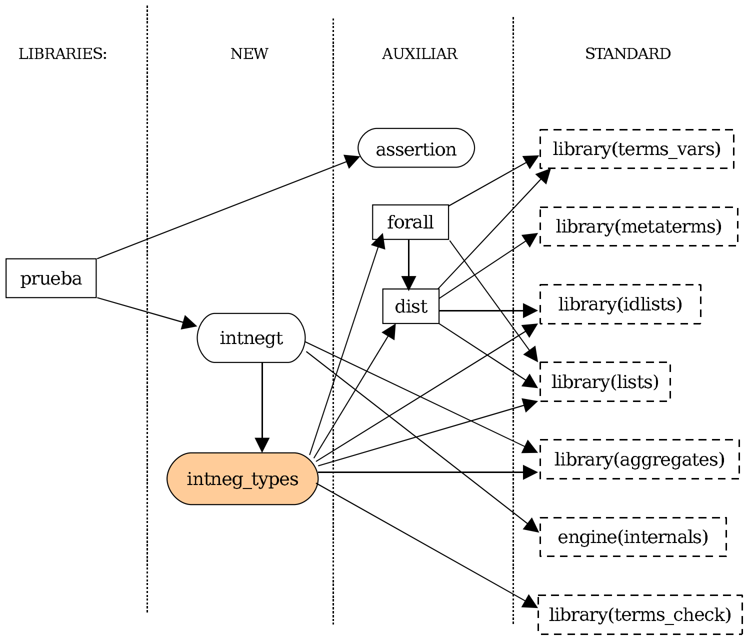

Fig. 4. Packages and modules that are use to implement the intensional constructive negation with types.

than classical and finite constructive negation.

Previous work [14] already presented the implementation of this technique with the predicate *intneg*/1. This predicate uses two important auxiliary predicates *dist*/2 or =/ =/2 that implement the disequality constraints between terms, and predicate *forall*/2 that implements universal quantification. The universal quantification is necessary when the predicate we are negating has free variables in its definition.

We have implemented the predicate *intnegt*/1 in the package *intnegt* which uses another auxiliary package that we have also developed (alike to *gen_types*) called *intneg_types* (this package takes the predicate *intneg*/1 as basis)

The predicate *intnegt*/1 negates goals but only goals of predicates whose code has no free variables. The reason is that we have not adapted yet the implementation of the universal quantification for working with types. It will be the following step to improve the implementation and we will include it in future works (section 6).

The package *intneg_types* expands the source code generating for each predicate *pred*/*n* (of the source program) the complementary predicate called *pred_check_types*/*n* where the types of the arguments of *pred*/*n* (that have been declared with the "success" assertions) are checked.

If a Prolog program, e.g.*prog.pl*, wants to use intensional constructive negation with types, then it should load the package *intnegt.pl* and the standard package *assertions.pl*. The first one is needed to obtain the complementary (negated) predicates and second one to be able to declare the types of the arguments of the predicates (using "success").

```
:- module(prog,_,[assertions,.(intnegt)]).
```

In Figure 4 we can see the tree of dependencies of the packages and modules that are needed. Again we differentiate the source of libraries.

## 5 Examples

*Even.* We define a predicate *type_nat*/1 for representing the infinite (it is a recursive type) Peano's natural numbers and the predicate *even*/1 to represent the even natural numbers. We declare that the type of the argument of *even* is of type *type_nat*:

```
% type_nat -> 0 ++ s (type_nat)
type_nat(0).
type_nat(s(X)):- type_nat(X).

% even: type_nat
even(0).
even(s(s(X))):- even(X).

:- success even(X) => (type_nat(X)).
```

Let us see a call to a positive goal:

```
?- even(X).
X = 0 ? ;
X = s(s(0)) ? ;
X = s(s(s(s(0)))) ? ;
X = s(s(s(s(s(s(0)))))) ? ;
X = s(s(s(s(s(s(s(s(0)))))))) ?
yes
```

Now, we can see a call to a negative goal, executed by our previous version of classical constructive negation. Constraints are returned:

```
?- cneg(even(X)).
X/s(s(fA(_A))), X/0 ? ;
X = s(s(_A)), _A/s(s(fA(_B))), _A/0 ? ;
X = s(s(s(s(_A)))), _A/s(s(fA(_B))), _A/0 ? ;
X = s(s(s(s(s(s(_A)))))), _A/s(s(fA(_B))), _A/0 ?
yes
```

The notation fA(X) means for all X, i.e. $\forall X$. So, the answers are equivalent to the formula $((\forall A.X \neq s(s(A))) \wedge X \neq 0) \vee (X = s(s(A)) \wedge (\forall B.A \neq s(s(B))) \wedge A \neq 0) \vee (X = s(s(s(A)))) \wedge (\forall B.A \neq s(s(B))) \wedge A \neq 0) \vee ....$ An equivalent result is obtained by using our new proposal of classical constructive negation with types over $even(X)$ but obtaining so many values (instead of constraints) as wanted because there are infinitely many:

```
?- cnegt(even(X)).
X = s(0) ? ;
```

```
X = s(s(s(0))) ? ;
X = s(s(s(s(s(0))))) ? ;
X = s(s(s(s(s(s(s(0))))))) ?
yes
```

*Intersection.* We define a predicate *list_digit*/1 to define the type of list of digits, and the predicate *intersection*/3 that returns in the third argument the intersection of the two first arguments. We declare that the type of the three arguments is *list_digit* but also other constraints can be declared, for example that the length of the result should be a list of only two elements:

```
% digit -> 0 ++ 1 ++ 2 ++ 3 ++ 4 ++ 5 ++ 6 ++ 7 ++ 8 ++ 9
digit(0).     digit(1).     digit(2).     digit(3).     digit(4).
digit(5).     digit(6).     digit(7).     digit(8).     digit(9).
% list_digit -> [] ++ [digit|list_digit]
list_digit([]).
list_digit([H|T]):-
        digit(H),
        list_digit(T).
% intersection: list_digit * list_digit * list_digit
intersection([],_,[]):-  !.
intersection([H|T],L,[H|R]):-
        member(H,L),
        intersection(T,L,R), !.
intersection([_|T],L,R):-
        intersection(T,L,R).

:- success intersection(L1,L2,Res) =>
        (list_digit(L1),list_digit(L2),length(Res,2),list_digit(Res)).
```

Let us see a couple of calls to positive goals:

```
?- intersection([1,3,4,5],[4,5,6],Res).
Res = [4,5] ? ;
no
?- intersection([j,3,h,5],[6,h,h],Res).
Res = [h] ? ;
no
```

Executing this program in Ciao Prolog, we notice that the type constraint is not taken into account, returning a solution (in this case $[h]$) that is not of type *list_digit*. However, in our typed framework the call to the finite constructive negation of that goal returns:

```
?- cnegf(intersection([j,3,h,5],[6,h,h],Res)).
Res/[h] ? ;
no
```

The answer is equivalent to the formula $Res \neq [h]$. Nevertheless, the result using finite constructive negation with types is completely different because $cnegft/1$ takes into account type information. So, it fails:

```
?- cnegft(intersection([j,3,h,5],[6,h,h],Res)).
no
```

*Days.* We define a predicate $monthDays/2$ where the first argument is of type $month/1$ and the second is a natural number (28, 29, 30 or 31) and also a simple predicate $irregularMonth/1$ to identify February:

```
month(january).                 ...     month(december).
irregular_month(february).

month_days(january,31).                 month_days(february,28).
month_days(february,29).                month_days(march,31).
...
month_days(november,30).                month_days(december,31).

:- success irregular_month(X) => (month(X)).
:- success month_days(X,_Y) => (month(X)).
```

An example can be querying for the months that have not 31 days and that are not irregular months. If we use intensional constructive negation we will obtain a constraint:

```
?- intneg(month_days(X,31)), intneg(irregular_month(X)).
X/january, X/february, X/march, X/may, X/july, X/august,
X/october, X/december ? ;
no
```

If we use intensional constructive negation with types, then we obtain the particular values, i.e. the four real answers:

```
?- intnegt(month_days(X,31)), intnegt(irregular_month(X)).
X = april ? ;
X = june ? ;
X = september ? ;
X = november ? ;
no
```

*Length.* We define the predicate $length/2$ where the first argument is a list of even numbers (type $even\_list/2$) and the second argument is the number of elements of the list. We also use the predicate $even/1$ defined in the first example of this section.

```
length([],0).
length([X|Rest],s(N)):-
length(Rest,N).
```

| goals | Goal | cneg(Goal) | cnegt(Goal) |
|---|---|---|---|
| goes_throught(X,[a,b,d,e]) | 999.173 | 1036.509 | 1029.397 |
| goes_throught(X,[a,b,d,e]) x 1000 rep. | 1044.065 | 4116.26 | 1108.069 |
| ancestor(X,maria) | 1036.953 | 1014.730 | 1038.287 |
| ancestor(X,maria) x 1000 rep. | 988.062 | 10728.67 | 4100.259 |
| even(X) | 1044.065 | 992.062 | 1000.062 |

Table 1
Runtime comparison (in miliseconds) for classical constructive negation.

```
even_list([]).
even_list([Elem|Rest]):-
even(Elem),even_list(Rest).

:- success length(List,_Len) => (even_list(List)).
```

If we ask for the intensional constructive negation of the goal $length(X, s(0))$ (i.e. the lists that have not one element), then we obtain two answers (not exclusive) using constraints:

```
?- intneg(length(X,s(0))).
X/[fA(_B)|fA(_A)] ? ;
X = [_|_A], A/[] ? ;
no
```

The intensional constructive negation of the same goal give us (infinite) particular lists as answers of the query:

```
?- intnegt(length(X,s(0))).
X = [] ? ;
X = [0,0] ? ;
X = [0,0,0] ?
yes
```

## 6 Conclusion

Using some of the examples that are discussed in Section 5, and some additional examples not reported here by lack of space (see also http://babel.ls.fi.upm.es/~susana/code/negation/types/) we have obtained some measurements of run-times for evaluating efficiency results. The measurements for the classical constructive negation with types are displayed in Table 1, the results for finite constructive negation with types are shown in Table 2 and the results for intensional constructive negation with types are shown in Table 3.

All time measurements of classical/finite/intensional constructive negation with types are of the same complexity order than the measurements of the classical/finite/intensional constructive negation without types. So, the efficiency results of introducing types in the negation are as good as expected. The advantage of obtaining concrete values instead of constraints is not delaying the execution of negative queries. Furthermore, there are important speedups sometimes due to the

| **goals** | **Goal** | **cnegf(Goal)** | **cnegft(Goal)** |
|---|---:|---:|---:|
| member(X,[1,2,3]) | 1112.514 | 1101.024 | 1131.182 |
| member(X,[1,2,3]) x 100000 rep. | 1048.066 | 6792.424 | 3404.212 |
| basic(X,Y,Z) | 1133.404 | 1090,512 | 1114.736 |
| basic(X,Y,Z) x 100000 rep. | 1148.072 | 4708.294 | 2832.177 |
| student(pedro) | 1084.068 | 0.0 | 0.0 |
| student(X) | 1130.737 | 1142.673 | 1129.848 |
| student(X) x 100000 rep. | 1268.079 | 8152.509 | 2416.151 |
| intersection([1,3,4,5],[4,5,6],Res) | 1135.182 | 1095.624 | 1116.514 |
| length(X,s(s(0))) | 1113.403 | 1117.403 | 1105.402 |
| length(X,$s^{2000}$(0)) | 1088.068 | 4184.262 | 1008.063 |

Table 2
Runtime comparison (in miliseconds) for finite constructive negation.

| **goals** | **Goal** | **intneg(Goal)** | **intnegt(Goal)** |
|---|---:|---:|---:|
| monthDays(X,31) | 1095.624 | 1097.846 | 1116.069 |
| monthDays(X,31) x 1000 rep. | 1072.067 | 7776.487 | 1124.07 |
| length(X,0) x 500 rep. | 1060.060 | 1184.074 | 1080.07 |
| length(X,s(s(0))) x 500 rep. | 1073.845 | 28741.080 | 1088.068 |
| connect(X,Y) | 962.282 | 963.171 | 990.728 |
| connect(X,Y) x 1000 rep. | 936.058 | 37457.337 | 984.061 |

Table 3
Runtime comparison (in miliseconds) for intensional constructive negation.

reduction of the search tree by using types instead of the whole Herbrand Universe.

One of the problems that we found at implementation level was a bug in Ciao Prolog, that causes the only problem we could not solve. While modules and packages allows for more than one module of package inclusion this is not the case for programs when more than one non-standards package needs to be used. This is our case if we want to include all (*cneg.pl* and *cnegt.pl* on one side, and *intneg.pl* and *intnegt.pl* on the other side). We have solved it by including the contents of one package inside the new package. We have reported the bug to the Ciao developers for future releases. In this case we can provide a more clean and re-usable structure.

In order to show the usefulness of our proposal we have shown some simple but interesting examples. Additionally to the mentioned examples, we have also studied another set of examples. They include more everyday examples related to searching information in the web. Unfortunately they can't be shown here due to the lack of space. One of them is related to querying in a web with information of movies that can be seen in our city. Suppose we are looking for films with neither violence nor sex. One cannot accept an answer saying that the selected movie is not *$9\frac{1}{2}$ weeks* (constrained answer) but a response saying "try *Bambi*" is more useful (constructive answer). Another example is a web page of flats and apartments that are offered for being sold or rented. We may want to search for flats and apartments that are not in the center of the city and in both examples we want to obtain concrete results (a list of them, probably) but not an answer with constraints on concrete flats, streets or dimensions that does not resolve our needs. Notice that on this kind of examples related to web semantics, types of elements are well defined (including the general structure either html or XML).

Another interesting advantage related to the use of types in constructive negation is the detection of type errors (as we have seen in example *Intersection* of Section 5) that are not controlled directly by Prolog. Also notice that our approaches work identically for finite or infinite (recursive) defined types.

As future work we want to follow two directions. The first one is related to improving our implementations. The main point left (although not difficult) is extending intensional constructive negation for all cases, i.e. we have to adapt the implementation of the universal quantification to the use of types and then our implementation of intensional negation with types will be complete. The approach is quite interesting as can help in the generation of adequate coverings (i.e. schemes of elements that are checked for universal quantification). The second direction is more related to applications of the technique. An obvious area of application is querying and reasoning in the semantics web, that have been sketched above. Another area is related with coding rule conformance (see [10]). Coding rules are customarily used to constrain the use (or abuse) of certain programming language constructions. Usually they are written in natural language and include in many cases negative information. Our approach is to describe these rules in Prolog and incorporate them in a widely used compiler (GCC). Additional areas are databases querying, deduction of instances from ontologies, business rules, etc.

## References

[1] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19(3):281–313, 1992.

[2] R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *JLP*, 8(3):201–228, 1990.

[3] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and manual at `http://www.cliplab.org/Software/Ciao/`.

[4] D. Chan. Constructive negation based on the complete database. In *Proc. Int. Conference on LP'88*, pages 111–125. The MIT Press, 1988.

[5] D. Chan. An extension of constructive negation and its application in coroutining. In *Proc. NACLP'89*, pages 477–493. The MIT Press, 1989.

[6] N. Heintze and J. Jaffar. *Semantic types for logic programs*, volume [18], pages 141–155. 1992.

[7] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

[8] M. Hermenegildo, F. Bueno, D. Cabeza, M. García de la Banda, P. López, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming.* Nova Science, Commack, NY, USA, April 1999.

[9] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

[10] G. Marpons, J. Mariño, A. Herranz, L. Fredlund, M. Carro, and JJ. Moreno-Navarro. Automatic coding rule conformance checking using logic programming. In P. Hudak and D.S. Warren, editors, *Practical Applications of Declarative Languages, PADL 08*, pages 279–293, San Francisco, January 2008. Springer Verlag. ISBN //978-3-540-77441-9.

[11] G. Meyer. On the use of types in logic programming. Informatik beritche, FernUniversität Hangen, Germany, 1996.

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

[13] P. Mishra. Towards a theory of types in Prolog. In *International Symposium on Logic Programming*, pages 289–298, 1984.

[14] S. Muñoz, J. Mariño, and J. J. Moreno-Navarro. Constructive intensional negation. In *Proceedings of the 7th International Symposiun in Functional and Logic Programming, FLOPS'04*, number 2998 in LNCS, pages 39–54, Nara, Japan, 2004.

[15] S. Muñoz and J. J. Moreno-Navarro. Implementation results in classical constructive negation. In Mireille Ducassie, editor, *ICLP'04*, volume 3132 of *Lecture Notes in Computer Science (LNCS)*, pages 284–298, Saint-Maló, France, September 2004. Springer-Verlag.

[16] S. Muñoz, J. J. Moreno-Navarro, and M. Hermenegildo. Efficient negation using abstract interpretation. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning*, number 2250 in LNAI, pages 485–494, La Habana (Cuba), 2001. LPAR 2001.

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

[18] F. Pfening, editor. *Types in logic programming.* The MIT Press, 1992.

[19] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo. A practical type analysis for verification of modular prolog programs. In *ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation (PEPM'08)*, pages 61–70. ACM Press, January 2008.

[20] G. Puebla, F. Bueno, and M. Hermenegildo. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *LNCS*, pages 23–61. Springer-Verlag, September 2000.

[21] P. Stuckey. Negation and constraint logic programming. In *Information and Computation*, volume 118(1), pages 12–33, 1995.

[22] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.