

ELPS manual

March 13, 2014

Contents

1	System installation	2
2	System usage	3
3	Syntax Description	4
3.1	Directives	4
3.2	Sort definitions	4
3.3	Predicate Declarations	7
3.4	Program Rules	8
4	Typechecking	8
4.1	Type errors	8
4.1.1	Sort definition errors	8
4.1.2	Predicate declarations errors	10
4.1.3	Program rules errors	11

1 System installation

For using the system, you need to have the following installed:

1. Java Runtime Environment (JRE):

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

The system was tested on Java versions 1.6.0_37 and 1.7.0_25.

2. The ELPS binary file:

<https://github.com/iensen/elps/blob/master/elps.jar?raw=true>.

3. Clingo (version 4.2.1 or later):

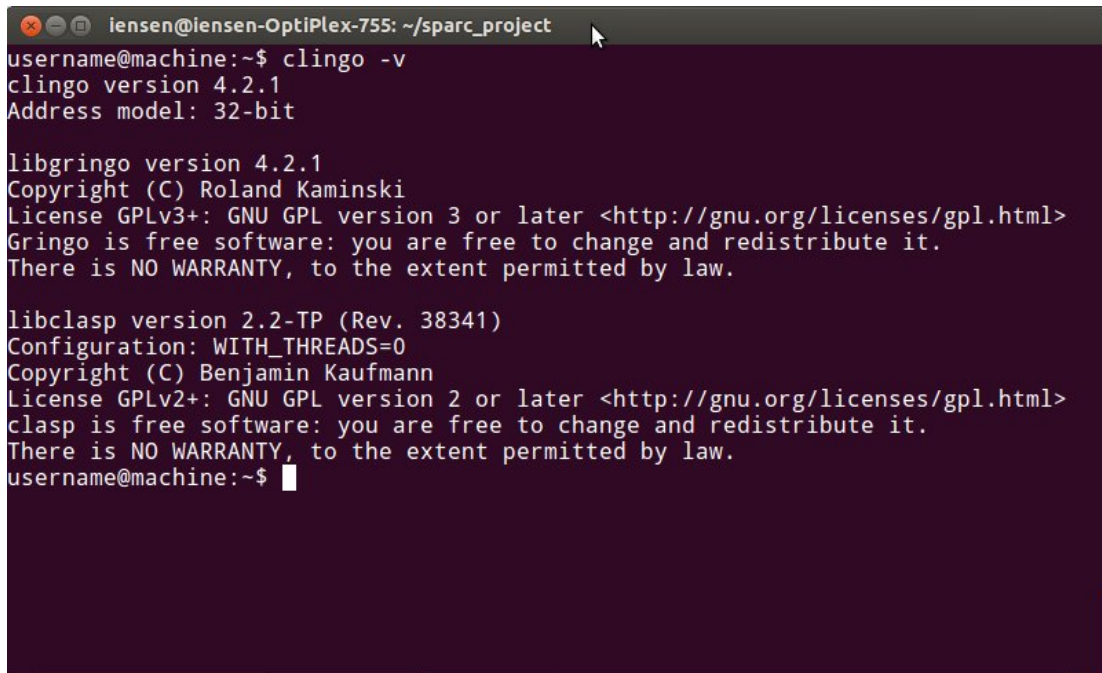
<http://sourceforge.net/projects/potassco/files/clingo/4.2.1>

Be sure the PATH system variable includes the directory where the clingo executable is located. For instructions on how to view/modify the PATH system variable, see either of the following links:

<http://www.java.com/en/download/help/path.xml>

<http://www.cyberciti.biz/faq/appleosx-bash-unix-change-set-path-environment-variable/>

To check if the solver is installed correctly, run the command `clingo -v`. See figure 1 for the expected output.



```
iensen@iensen-OptiPlex-755: ~/sparc_project
username@machine:~$ clingo -v
clingo version 4.2.1
Address model: 32-bit

libgringo version 4.2.1
Copyright (C) Roland Kaminski
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
Gringo is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

libclasp version 2.2-TP (Rev. 38341)
Configuration: WITH_THREADS=0
Copyright (C) Benjamin Kaufmann
License GPLv2+: GNU GPL version 2 or later <http://gnu.org/licenses/gpl.html>
clasp is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
username@machine:~$
```

Figure 1: Checking the version of Clingo solver

2 System usage

To demonstrate the usage of the system we will use the program Π below described in [2].

```
sorts
#student = {mike,mary,ann}.

predicates

eligible(#student).
highGPA(#student).
fairGPA(#student).
minority(#student).
interview(#student).

rules
eligible(X):- highGPA(X).
eligible(X):- minority(X), fairGPA(X).
-eligible(X):- -fairGPA(X), -highGPA(X).
interview(X):- not K$ eligible(X), not K$ -eligible(X).

% data
fairGPA(mike) | highGPA(mike).
highGPA(mary).
```

To run \mathcal{ELPS} solver on the program above, we change current directory to a directory having the file `program.sp` with the program written in it, and the downloaded file `elps.jar`. Then, we run the command:

```
iensen@iensen-OptiPlex-755:~/elps$ java -jar elps.jar elib.lp
ELPS V1.02
program translated
World View 1 out of 1
{{eligible(mary), student(mary), student(mike), interview(mike),
  highGPA(mary), fairGPA(mike)}, {eligible(mary), student(mary),
  student(mike), interview(mike), highGPA(mike), highGPA(mary),
  eligible(mike)}}
```

The program has a single world view, which contains `interview(mike)` in both believe sets.

3 Syntax Description

3.1 Directives

Directives should be written before sort definitions, at the very beginning of a program. *ELPS* allows two types of directives:

#maxint

Directive `#maxint` specifies maximal nonnegative number which could be used in arithmetic calculations. For example,

```
#maxint=15.
```

limits integers to $[0,15]$.

#const

Directive `#const` allows one to define constant values. The syntax is:

```
#const constantName = constantValue.
```

where *constantName* must begin with a lowercase letter and may be composed of letters, underscores and digits, and *constantValue* is either a nonnegative number or the name of another constant defined above.

3.2 Sort definitions

This section starts with a keyword *sorts* followed by a collection of sort definitions of the form:

```
sort_name=sort_expression.
```

sort_name is an identifier preceeded by a pound sign `#`. *sort_expression* on the right hand side denotes collection of strings called *sorts*. We divide all the sorts into *basic* and *non-basic*.

Basic sorts are defined as named collections of identifiers, i.e, strings consisting of

- latin letters: $\{a, b, c, d, \dots, z, A, B, C, D, \dots, Z\}$
- digits: $\{0, 1, 2, \dots, 9\}$
- underscore: `_`

and either starting from a letter or containing only digits.

Non-basic sorts also contain *records* of the form $id(\alpha_1, \dots, \alpha_n)$, where id is an identifier and $\alpha_1, \dots, \alpha_n$ are either identifiers or records.

We define sorts by means of expressions (in what follows sometimes referred as statements) of five types:

1. numeric range.

`numeric_range := number1..number2`

number1 should be smaller or equal than *number2*. The expression defines the set of subsequent numbers $\{number1, number1 + 1, \dots, number2\}$

Example:

`#sort1=1..3`

`#sort1` consists of numbers $\{1, 2, 3\}$.

2. identifier range

`id_range := id1..id2`

id1 should be lexicographically smaller or equal than *id2*. *id1* and *id2* should both consist of digits and letters and start from a lowercase letter. The expression defines the set of all strings

$S = \{s : id1 \leq s \leq id2 \wedge |id1| \leq |s| \leq |id2|\}$

Example:

`#sort1=a..f.`

`#sort1` consists of latin letters $\{a, b, c, d, e, f\}$.

3. set of ground terms

`ground_terms_set := {t_1, ..., t_n}`

`ground_terms_set` denotes a set of ground terms $\{t_1, \dots, t_n\}$, defined as follows:

- numbers and constants are ground terms;
- If f is an identifier and $\alpha_1, \dots, \alpha_n$ are ground terms, then $f(\alpha_1, \dots, \alpha_n)$ is a ground term.

Example :

```
#sort1={f(a),a,b,2}.
```

4. record

```
functional_term := f(sort_name1(var_1),..., sort_namen(var_n)):
                  condition(var_1,...,var_n)
condition(var_1,...,var_n) := var_i REL var_j
condition(var_1,...,var_n) := condition and condition
                           | condition or condition
                           | not(condition)
                           | (condition)
```

Variables var_1, \dots, var_n are optional as well as the condition. Condition can only contain variables from the list var_1, \dots, var_n . If there is a subcondition $var_i \text{ REL } var_j$, where REL is either $\{>, \geq, <, \leq\}$ then $sortname_i$ and then $sortname_j$ must be defined by basic statements.

The expression defines a collection of ground terms

$\{f(t_1, \dots, t_n) : condition(t_1, \dots, t_n) \text{ is true} \wedge t_1 \in s_i \wedge \dots \wedge t_n \in s_n\}$

Example

```
#s=1..2.
#sf=f(s(X),s(Y),s(Z)): (X=Y or Y=Z).
```

The sort $\#sf$ consists of records $\{f(1, 1, 2), f(1, 1, 1), f(2, 1, 1)\}$

5. set-theoretic expression.

```
set_expression := sort_name | ground_term_set | functional_term
set_expression := (set_expression)
                  | (set_expression + set_expression)
                  | (set_expression * set_expression)
                  | (set_expression - set_expression)
```

$sort_name$ must be a name of a sort occurring in one of the preceeding sort definitions. The operations $+$ $*$ and $-$ stand for union, intersection and difference correspondingly.

Example :

```
#sort1={a,b,2}.
#sort2={1,2,3} + {a,b,f(c)} + f(#sort1).
```

$\#sort2$ consists of ground terms $\{1, 2, 3, a, b, f(c), f(a), f(b), f(2)\}$.

6. concatenation

`concatenation := [b_stmt_1] ... [b_stmt_n]`

`b_stmt_1, ..., b_stmt_n` must be *basic statements*, defined as follows:

- statements of the forms (1)-(3) are basic
- statement S of the form (5) is basic if:
 - it does not contain sort expressions of the form (4), denoting sets of records
 - all curly brackets occurring in S contain only constants consisting of latin letters and digits
 - all sorts occurring in S are defined by basic statements

Note that basic statement can only define a basic sort not containing records.

*Example*¹..:

```
#sort1=[b] [1..100].
```

`sort1` consists of identifiers $\{b_1, b_2, \dots, b_{100}\}$.

3.3 Predicate Declarations

The second part of a \mathcal{ELPS} program starts with the keyword *predicates*

and is followed by statements of the form

`pred_symbol(#sortName, ..., #sortName)`

Multiple declarations containing the same predicate symbol are not allowed. 0-arity predicates must be declared as `pred_symbol()`. For any sort name SN , the system includes declaration $SN(SN)$ automatically.

¹We allow a shorthand 'b' for singleton set $\{b\}$

3.4 Program Rules

The third part of a *SPARC* program starts with the keyword *rules* followed by rules of the form.

$$\ell_0 | \ell_1 \dots | \ell_n \leftarrow \ell_{n+1}, \dots, \ell_m, \text{not } \ell_{m+1} \dots \text{not } \ell_k \quad (1)$$

where each ℓ_i is either a literal or a subjective literal. Subjective literals can be in one of the forms $K\$ \ell$, $M\$ \ell$, $\text{not } K\$ \ell$, $\text{not } M\$ \ell$.

Literals occurring in the heads of the rules must not be formed by predicate symbols occurring as sort names in sort definitions. In addition, rules must not contain *unrestricted variables*.

Definition 1 (*Unrestricted Variable*) A variable occurring in a rule of a *SPARC* program is called *unrestricted* if all its occurrences in the rule either belong to some relational atoms of the form *term1 rel term2* (where $\text{rel} \in \{>, >=, <, <=, =, !=\}$) and/or some terms appearing in a head of a choice or aggregate element.

Example 1 Consider the following *ELPS* program:

```

sorts
#s={f(a),b}.
predicates
p(#s).
rules
p(f(X)) :- Y<2, 2=Z, F>3, #count{Q:Q<W, p(W), T<2}, p(Y).

```

Variables F,T,Z,Q are unrestricted.

4 Typechecking

If no syntax errors are found, a static check program is performed all found type-related problems, classified into type errors and type errors.

4.1 Type errors

Type errors are considered as serious issues which make it impossible to compile and execute the program. Type errors can occur in all four sections of a *SPARC* program.

4.1.1 Sort definition errors

1. Set-theoretic expression (statement (2) in section 3.2) contains a name of undefined sort.

Example:


```

sorts
#s={a} .
#s2=#s1-s .

```

2. Sort with the same name is defined more than once. *Example:*

```

sorts
#s={a} .
#s={b} .

```

3. In an identifier range $id1..id2$ (statement (2) in section 3.2) the first identifier($id1$) is lexicographically greater than $id2$. *Example*

```

sorts
#s=zbc..cbz .

```

4. In a numeric range $n1..n2$ (statement (2) in section 3.2) $n1$ is greater than $n2$. *Example:*

```

sorts
#s=100500..1 .

```

5. Numeric range (statement (2) in section 3.2) $n1..n2$ contains an undefined constant.

```

#const n1=5.
sorts
#s=n1..n2 .

```

6. In an identifier range $id1..id2$ (statement (3) in section 3.2) the length of the first identifier($id1$) is greater than length of the second.

Example:

```

sorts
#s=abc..a .

```

7. Concatenation (statement (4) in section 3.2) contains a non-basic sort.

Example:

```

sorts
#s={f(a)} .
#sc=[a][#s] .

```

8. Record definition (statement (5) in section 3.2) contains an undefined sort.

Example:

```

sorts
#s=1..2.
#fs=f(s,s2) .

```

9. Definition of record (statement (5) in section 3.2) contains a condition with relation $>, <, \geq, \leq$ such that the corresponding sorts are not basic. *Example:*

```

#s={a,b} .
#s1=f(#s) .
#s2=g(s1(X),s2(Y)):X>Y.

```

10. Variable is used more than once in record definition(statement (5) in section 3.2).
Example:

```

sorts
#s1={a} .
#s=f(#s1(X),#s1(X)): (X!=X) .

```

11. Sort contains an empty collection of ground terms.
Example

```

sorts
#s1={a,b,c}
#s=#s1-{a,b,c} .

```

4.1.2 Predicate declarations errors

1. A predicate with the same name is defined more than once. *Example:*

```

sorts
#s={a} .
predicates
p(#s) .
p(#s,#s) .

```

2. A predicate declaration contains an undefined sort. *Example:*

```

sorts
#s={a} .
predicates
p(#ss) .

```

4.1.3 Program rules errors

In program rules we first check each atom of the form $p(t_1, \dots, t_n)$ and each term occurring in the program Π for satisfying the definitions of program atom and program term correspondingly[1]. Moreover, we check that no sort occurs in a head of a rule of Π .

References

- [1] Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. Towards answer set programming with sorts. In *Logic Programming and Nonmonotonic Reasoning*, pages 135–147. Springer, 2013.
- [2] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *The Journal of Logic Programming*, 19:73–148, 1994.