

An Encoding for CLP Problems in SMT-LIB

Daneshvar Amrollahi

University of Tehran

d.amrollahi@ut.ac.ir

Hossein Hojjat

TelAS, Khatam University

University of Tehran

hojjat@ut.ac.ir

Philipp Rümmer

University of Regensburg

Uppsala University

philipp.ruemmer@ur.de

The input language for today’s CHC solvers are commonly the standard SMT-LIB format, borrowed from SMT solvers, and the Prolog format that stems from Constraint-Logic Programming (CLP). This paper presents a new front-end of the Eldarica CHC solver that allows inputs in the Prolog language. We give a formal translation of a subset of Prolog into the SMT-LIB commands. Our initial experiments show the effectiveness of the approach and the potential benefits to both the CHC solving and CLP communities.

1 Introduction

Over the last years, a growing number of solvers for Constrained Horn Clauses (CHC) have been developed; for instance, Spacer [8], Eldarica [5], Golem [1], and RInGEN [2]. There are two main languages used to interface such solvers: the SMT-LIB language [3], in which Horn clauses can either be expressed using quantified assertions, or using the rule-based notation that was introduced by Z3; and dialects of Prolog [6] as a language in Constraint-Logic Programming (CLP). The former language is designed primarily for machine-generated input to solvers, as it is simple to parse, strongly typed, and has unambiguous semantics. The latter language is more concise and convenient for handwritten programs. There is, unfortunately, no exact match between the theories considered in CLP and SMT-LIB, and some Prolog language features have semantics that are not straightforward to model; for instance, the default absence of the “occurs-check” in equations.

This paper presents an ongoing effort to add a comprehensive Prolog/CLP front-end to the CHC solver Eldarica [5]. Eldarica has been able to process clauses in Prolog format since the beginning of its development; however, the existing Prolog front-end of Eldarica is restricted to the parsing of clauses over integers, and it is planned to replace it with a front-end with more extensive support for the different Prolog features through this project. To document the applied interpretation of Prolog, we define a formal translation of a fragment of Prolog to the SMT-LIB language. In the scope of this paper, we focus on three key features of Prolog: functions, defining a Herbrand universe of values; lists; and integer numbers as introduced by CLP(\mathbb{Z}). We model the semantics of those language features using a mapping to SMT-LIB data-types (i.e., algebraic data-types with free constructors) and SMT-LIB integers.

Example To illustrate the complementarity of CHC and CLP, we start with a simple routing example including integer arithmetic, lists, and functions in Prolog. Figure 1 shows a CLP(\mathbb{Z}) program to compute the distance between cities. The program consists of 9 facts and 2 rules. A fact of the form `distance(X, Y, Z)` states that the direct distance between cities X and Y is Z . The rule in line 12 implies that distance is symmetric. The meaning of `path(X, Y, Z, L)` is that there exists a path of length Z from X to Y where L represents the path as a list of points in the format `waypoint(C, D)`. This shows that city C lies on the path from X to Y with a distance of D from the starting point, which is X . The fact `path(A, A, 0, [waypoint(A, 0)])` states that for any city A , there is a path of length 0 to itself and the list

```

1  :- use_module(library(clpz)). % or clpfd
2
3  distance(tehran,    vienna, 31).
4  distance(vienna,   paris, 10).
5  distance(vienna,   munich, 3).
6  distance(paris,    munich, 10).
7  distance(paris,    rome,   11).
8  distance(lausanne, rome,   6).
9  distance(lausanne, munich, 4).
10 distance(tehran,   paris, 42).
11
12 distance(A, B, D) :- distance(B, A, D).
13
14 path(A, A, 0, [waypoint(A, 0)]).
15 path(A, C, D, [waypoint(C, D) | N]) :- path(A, B, P, N), distance(B, C, Q),
16                                     D #= P + Q.
17
18 ?- path(tehran, munich, D, X), D #< 40.

```

Figure 1: Prolog Program for distance between cities in CLP(\mathbb{Z})

presenting the path is the city A itself. Finally, the rule on line 15 implies that if there is path N of length P from A to B, and the direct distance from B to C is Q and D is equal to $P + Q$, then we have found a path from A to C, which is the previously found path (N) extended with the city C. The query `path(tehran, munich, D, X), D #< 40` searches for a path from munich to tehran of length less than 40. A possible answer to this query is $D = 34$ and $X = [\text{waypoint}(\text{munich}, 34), \text{waypoint}(\text{vienna}, 31), \text{waypoint}(\text{tehran}, 0)]$. This means that there is a path from tehran to munich of distance 34, with vienna being on the way with distance 31 from tehran.

Note that this Prolog program, while intuitive, is also rather inefficient, and interpreting it using a CLP engine might lead to non-termination due the recursion on lines 12,15. The program can be rewritten to a more operationally oriented (and harder to read) set of clauses to be terminating and efficient, in particular preventing cyclic paths from being explored, and inlining the length bound to allow branches without solutions to be pruned. Alternatively, tabling [4] could be used for the predicate `path`¹.

CHC solvers are generally less efficient than Prolog engines for solving constraint satisfaction problems. However, the abstraction techniques in CHC solvers are naturally able to cope with clauses written in declarative and otherwise inefficient style. They can easily find a path from tehran to munich of length 40 in Figure 1. CHC solvers are also able to determine that no such path exists for lengths less than 34, a task more challenging for at least some CLP solvers. More generally, CHC solvers are agnostic of the order of clauses, and the order of literals in clauses, and process clauses focusing more on their logical content than syntactic features. In this sense, CHC solvers can be a useful debugging tool, and have a complementary performance profile to CLP systems. The goal of our work is to simplify the integration of CLP and CHC methods, by providing a translation of a subset of Prolog into SMT-LIB, the common input language of CHC solvers.

Organization of the paper: In Section 2 we overview the CLP and the SMT-LIB input languages. Section 3 discusses translation rules for converting Prolog to SMT-LIB. Section 4 translates the introductory example to SMT-LIB. We conclude the paper and present directions of future research in Section 5.

¹Which, however, led to an error in experiments with SWI-Prolog [11].

$\langle \text{Database} \rangle ::= \langle \text{Clause} \rangle^*$ $\langle \text{Clause} \rangle ::= \langle \text{Predicate} \rangle \text{ '.' } \mid \langle \text{Predicate} \rangle \text{ ':-'} \langle \text{BodyItem} \rangle^* \text{ '.' } \mid \text{ '?-' } \langle \text{BodyItem} \rangle^* \text{ '.' }$ $\langle \text{BodyItem} \rangle ::= \langle \text{Predicate} \rangle \mid \langle \text{Constraint} \rangle$ $\langle \text{Predicate} \rangle ::= \langle \text{Atom} \rangle \mid \langle \text{Atom} \rangle \text{ '(' } \langle \text{Term} \rangle^* \text{ ')' }$	$\langle \text{Term} \rangle ::= \langle \text{Variable} \rangle \mid \langle \text{Atom} \rangle \mid \langle \text{Atom} \rangle \text{ '(' } \langle \text{Term} \rangle^* \text{ ')' } \mid \langle \text{List} \rangle \mid \langle \text{Integer} \rangle$ $\langle \text{List} \rangle ::= \text{ '[' } \langle \text{Term} \rangle^* \text{ ']' } \mid \text{ '[' } \langle \text{Term} \rangle^* \text{ ',' } \langle \text{Term} \rangle \text{ ']' }$
---	--

Figure 2: A simplified grammar for describing the syntax of Prolog programs.

2 Preliminaries

2.1 Constraint Logic Programming (CLP)

Constraint Logic Programming (CLP) [10], first introduced by Jaffar and Lassez in 1987 [7], is a programming paradigm that combines the benefits of constraint programming and logic programming. It enables the modeling of complicated real-world problems with variables and constraints, and uses logical and constraint reasoning to find a solution. CLP expresses the connections between variables as constraints and looks for a derivation of queries from given clauses.

Prolog is the main language of CLP, where the constraints are equations over the algebra of terms. CLP problems may in addition contain symbols with pre-defined meanings, defined by a *theory*. CLP(\mathbb{Z}) [9] refers to CLP over the theory of integer arithmetic. Figure 2 is a simplified grammar that is able to parse Prolog programs with functions, lists, and integer arithmetic. An extended version of it is used in our actual implementation.

As shown in Figure 2, every Prolog program consists of a set of *Clauses*. A clause has a *head* and a *body*. The head can have zero or one predicates, and the body is a list of predicates and constraints. A clause is either a *Fact*, a *Rule*, or a *Query*. For example, `man(tom)` is a fact, and `friends(X, Y) :- likes(X, Y), likes(Y, X)` is a rule with `friends(X, Y)` as the head and `likes(X, Y), likes(Y, X)` as the body. Queries always start with a `?-`. For instance, `?- likes(X, tom)` would search for an assignment for `X` such that `likes(X, tom)` can be derived from the set of given clauses. *Terms* can be variables, atoms, compound terms, lists, and integers.

Structured data is represented using compound terms. A compound term consists of a function and a sequence of one or more sub-terms. A function is characterized by its name, which is an atom, and its arity. For instance, in the fact `man(father(claire))`, the term `father(claire)` is a compound term.

In Prolog, there are different kinds of built-in equality operators (`#=`, `=`, `==`, `is`) with different semantics. The `=/2` predicate, which accepts two arguments, is used to ensure that its two arguments are syntactically equal through unification. For instance, the response to the query `?- father(X) = father(john)` will be positive as the sub-terms can be unified by setting `X = john`. By default, Prolog does not apply any “occurs-check”, however, which can sometimes cause non-termination. An example of this is `?- X = father(X)`. The unification algorithm decides to unify `X` with the right-hand-side, which is `father(X)`. But still there is an `X` left in this term; interpretation can get stuck in this loop of replacing `X` by `father(X)` and never terminate.

Lists are terms representing sequences of elements; for instance, `[a, [b, c], 7]` is a list of terms. A non-empty list can also be thought of as having two parts: *head* as the first element of the list, and

tail as the remainder of the list. Both representations can be parsed using the grammar rules in Figure 2.

The body of the clauses in a Prolog program may also include *constraints*, in addition to predicates. An example of a constraint in $\text{CLP}(\mathbb{Z})$, Constraint Logic Programming over the domain of integers, is $X + Y \#>= 3 * Z - 17$. Some other operators for constraints in $\text{CLP}(\mathbb{Z})$ include $\# =$, $\#>$, $\#>=$, $\#<$, $\#<=$, $+$, $-$, $*$, $/$, mod . Throughout the rest of this paper, whenever we refer to constraints, we refer to constraints in $\text{CLP}(\mathbb{Z})$.

2.2 SMT-LIB

SMT-LIB, or the Satisfiability Modulo Theories Library [3], is a standardized format for specifying logical formulas modulo various background theories. It is widely used in applications like software verification, hardware design, and automated reasoning, and is also used as a standard input format of CHC solvers. The SMT-LIB language supports a range of theories, including arithmetic, bit-vectors, and arrays, allowing users to express a wide variety of constraints.

An SMT-LIB script consists of a list of commands to be processed by an SMT solver. Important SMT-LIB commands are:

- `(set-logic L)`: Set the logic, i.e., the combination of theories that will be used. For CHC solvers, typically $L = \text{HORN}$.
- `(declare-fun f ($T_1 \dots T_n$) T)`: Declaration of a function or predicate f with the given argument types $T_1 \dots T_n$ and result type T . Types in SMT-LIB include the types provided by background theories (e.g., `Int` for integers, `Bool` for Booleans), as well as user-defined types. When interfacing CHC solvers, typically only predicates (i.e., Boolean-valued functions) are declared.
- `(declare-datatype T ($C_1 \dots C_n$))`: Declaration of an algebraic data-type T with constructors $C_1 \dots C_n$. Data-types can be recursive, and are the main modelling technique in SMT-LIB to represent structured data. For every constructor C , the data-type declaration will also introduce a tester (`_ is C`) to identify terms constructed using C , as well as selectors for the arguments. An extended version of the command, `declare-datatypes`, exists to define several mutually recursive data-types. We will see various examples of algebraic data-types in this article.
- `(assert ϕ)`: Assert a constraint ϕ , which can be formulated using the standard operators of first-order logic, the functions and predicates provided by background theories, and declared symbols. CHC solvers require that asserted constraints ϕ are *constrained Horn clauses*, which means that they fall into one of the following classes of formulas:

Facts:	$(p \ t_1 \ \dots \ t_k)$
Quantified facts:	$(\text{forall } (X_1 \ \dots \ X_n) \ (p \ t_1 \ \dots \ t_k))$
Rules:	$(\text{forall } (X_1 \ \dots \ X_n) \ (=> (\text{and } \psi_1 \ \dots \ \psi_m) \ (p \ t_1 \ \dots \ t_k)))$
Queries:	$(\text{forall } (X_1 \ \dots \ X_n) \ (=> (\text{and } \psi_1 \ \dots \ \psi_m) \ \text{false}))$

- `(check-sat)`: Instruct the SMT solver to check whether the constraints asserted up to this point are satisfiable.

Throughout the paper, only a small subset of the SMT-LIB commands are used. For a full definition of SMT-LIB, we refer to the standard documentation [3].

```

(declare-datatype U (
  (claire)
  (father (father_1 U))
)
)
(declare-fun man
  (U)
  Bool
)
(assert
  (man (father claire))
)

```

Figure 3: SMT-LIB equivalent of `man(father(claire))`

3 Translating CLP to SMT-LIB

In this section, we explain our translation from CLP problems in Prolog to SMT-LIB. We begin by a motivating example as a warm-up. After that, we present the translation rules for Prolog facts, rules, lists, and CLP(\mathbb{Z}) constraints. We remark that our implementation does not explicitly translate input problems to SMT-LIB, but instead directly constructs clauses using Eldarica’s internal data structures; the translation to SMT-LIB is presented for documentation purposes, and reflects the semantics that is applied.

3.1 A Motivating Example

Consider the Prolog fact `man(father(claire))`. This fact consists of the function `father` applied to an atomic term `claire`. To model this term, one could consider declaring corresponding uninterpreted functions in SMT-LIB. However, with uninterpreted functions, `claire` and `father(claire)` might be assigned the same value, despite being syntactically different terms: the equation `(= claire (father claire))` is satisfiable in SMT-LIB. This is inconsistent with Prolog semantics, in which `claire` and `father(claire)` are different and non-unifiable terms, and it is guaranteed that they stand for distinct elements of the Herbrand universe. In addition, most CHC solvers do not support uninterpreted functions.

We therefore propose to treat `claire` and `father` as constructors of an algebraic data-type. To translate the mentioned fact to SMT-LIB, the first step is to introduce a new algebraic data-type for all of the terms occurring in a program. The data-type will be called `U`, standing for *U*niversal. The constructors of `U` will be all atoms and functions appearing in the clauses. Figure 3 shows the full translation of the example. The constructor `father` is unary, and its definition also adds a selector `father_1` to retrieve the sub-term. We also define the predicate `man`, as a Boolean function with a single argument of type `U`. Finally, we add the fact `man(father(claire))` to the set of our clauses using the `assert` command. Section 3.2 provides further details.

It has to be noted that our encoding of terms does not exactly model Prolog semantics. As explained in Section 2, the unification algorithm of Prolog will not detect the non-unifiability of a query `?- X = father(X)` due to the missing occurs-check. In contrast, elements of SMT-LIB data-types are finite constructor terms, which implies that the equation `(= X (father X))` is not satisfiable in SMT-LIB. However, we believe that the finite-term semantics is usually the intended semantics of a Prolog program. It remains to be investigated in future work whether there is a way to represent Prolog semantics more closely in SMT-LIB.

3.2 Algebraic Data Types (ADTs)

We define a single algebraic data-type `U` for representing the type of all terms in a program. The constructors of `U` will be all functions and atoms appearing in the clauses. As this global type `U` is defined

$$\text{collectFunctions}(t) = \begin{cases} \{(atom, 0)\}, & \text{if } t = atom \\ \{(c, n)\} \cup \bigcup_{i=1}^n \text{collectFunctions}(a_i), & \text{if } t = c(a_1, \dots, a_n) \\ \emptyset, & \text{otherwise} \end{cases}$$

Figure 4: Function *collectFunctions* collecting functions in the terms

```
(declare-datatype U
(
  (f1 (f11 U) ... (f1a1 U))
  ...
  (fn (fn1 U) ... (fnan U))
)
```

Figure 5: Definition of the universal type U for functions $\{(f_1, a_1), \dots, (f_n, a_n)\}$

only once, we need to iterate through all clauses and collect the functions and atoms appearing in them. We define a recursive function responsible for this matter in Figure 4. This function is applied to a term t , and recursively collects the atoms and functions occurring in t , as well as their arity. For instance, the value of $\text{collectFunctions}(a(X, b, c(d, Y, e)))$ is $\{(e, 0), (d, 0), (c, 3), (b, 0), (a, 3)\}$.

Given that the set collected using *collectFunctions* is $\{(f_1, a_1), \dots, (f_n, a_n)\}$, the universal algebraic data type U should be defined as in Figure 5.

3.3 Translation of Facts

Section 3.1 shows an example of translating a fact to SMT-LIB. We now introduce the general case of this translation, and add rules for lists and integer arithmetic in Sections 3.5 and 3.6. We refer to triplets of the form $s \triangleright s' : \Phi$ as *translation judgements*. The meaning of a judgement is that the Prolog term s can be translated to an SMT-LIB term s' under a set of side conditions Φ . Side conditions are mainly needed to capture typing requirements. For instance, Prolog (CLP(\mathbb{Z})) raises an error when encountering

$$\begin{array}{c} \frac{}{a \triangleright a : \emptyset} \quad \text{a is an atom} \qquad \frac{}{V \triangleright V : \emptyset} \quad \text{V is a variable} \\[10pt] \frac{\{t_i \triangleright t'_i : \Phi_i\}_{i=1}^n}{f(t_1, \dots, t_n) \triangleright (f \ t'_1 \ \dots \ t'_n) : \bigcup_{i=1}^n \Phi_i} \quad \text{f is a function or predicate} \\[10pt] \frac{s \triangleright s' : \Phi_1 \quad t \triangleright t' : \Phi_2}{s = t \triangleright (= \ s' \ t') : \Phi_1 \cup \Phi_2} \qquad \frac{s \triangleright s' : \Phi}{\backslash + s \triangleright (\text{not } s') : \Phi} \\[10pt] \frac{s \triangleright s' : \Phi_1 \quad t \triangleright t' : \Phi_2}{s = \backslash = t \triangleright (\text{not } (= \ s' \ t')) : \Phi_1 \cup \Phi_2} \end{array}$$

Figure 6: Basic rules for translating Prolog to SMT-LIB

```

(declare-fun p
  (U ... U)
  Bool
)
(assert
  (forall ( (X1 U) ... (Xm U) )
    (=> (and Φ1 ... Φn) (p t'1 ... t'n)))
)

```

Figure 7: Translation of a fact $p(t_1, \dots, t_n)$. We assume that the elements of a set Φ_i are implicitly conjoined.

```

(declare-fun likes
  (U U)
  Bool
)
(declare-fun friends
  (U U)
  Bool
)
(assert
  (forall ( (X U) (Y U) )
    (=>
      (and (likes X Y) (likes Y X))
      (friends X Y)
    )
  )
)

```

Figure 8: Translation of a rule in SMT-LIB

the expression $a + b$, where a or b are not integers. Thus, when translating $a + b$ to SMT-LIB, we will later add side conditions that ensure the correct type of a and b . More details concerning the use of side conditions are given in Sections 3.5 and 3.6.

Figure 6 shows the basic translation rules. Each rule derives a translation judgement, the conclusion under the bar, from premises shown above the bar. The rules in Figure 6 are mostly self-explanatory, and recursively translate a given term or constraint to SMT-LIB. The rules assume, for sake of presentation, that atoms, variables, and functions are translated to SMT-LIB symbols with the same name.

The SMT-LIB translation of a Prolog fact $p(t_1, \dots, t_n)$ is shown in Figure 7 in terms of expressions t'_1, \dots, t'_n and side conditions Φ_1, \dots, Φ_n , which are defined as follows:

- For all $i \in \{1, \dots, n\}$ it holds that $t_i \triangleright t'_i : \Phi_i$.
- X_1, \dots, X_m are all the variables appearing in the terms t_1, \dots, t_n .

The fact is captured using an SMT-LIB assertion in Figure 7, listing the side conditions required by the translation as assumptions.

3.4 Translation of Rules

Before presenting the translation of rules, we begin with a concrete example, using the Prolog rule $\text{friends}(X, Y) \text{ :- likes}(X, Y), \text{ likes}(Y, X)$. This rule will be turned into a universally quanti-

```

(assert
  (forall ( (X1 U) ... (Xk U) )
    (=> (and ψ'1 ... ψ'm Φ0 ... Φm) ψ'0)
  )
)

```

Figure 9: Translation of a rule $\psi_0 \text{ :- } \psi_1, \dots, \psi_m$

```

(declare-datatypes () (
  (U
    (aList (theList L))
    ...
  )
  (L
    nil
    (cons (head U) (tail L))
  )
))

```

Figure 10: Algebraic data-types including lists

```

list_concat([], L, L).
list_concat([X1|L1], L2, [X1|L3]) :- list_concat(L1, L2, L3).

```

Figure 11: A Prolog program concatenating lists

fied implication in SMT-LIB. Given that we have already defined our universal data-type U as explained in Section 3.2, all we need is to declare functions in SMT-LIB for `likes` and `friends`, and create the rule using the universal quantifier. The full translation is illustrated in Figure 8.

The general schema for expressing a rule $\psi_0 :- \psi_1, \dots, \psi_m$ in SMT-LIB is shown in Figure 9, assuming that:

- For all $i \in \{0, \dots, m\}$ it holds that $\psi_i \triangleright \psi'_i : \Phi_i$.
- X_1, \dots, X_k are all the variables appearing in the rule.

The assertion representing the rule in Figure 9 has a similar shape as the assertion for facts in Figure 7.

3.5 Translation of Lists

We now add lists to the picture. The absence of static typing in Prolog makes it a-priori impossible to tell whether a Prolog variable will represent a list or a term constructed using some function (or possibly both). We therefore include lists as one case in our universal data-type U , and ensure the correct typing of terms through side-conditions in our translation rules.

```

(declare-fun list_concat (U U U) Bool)

(assert (forall ((X U)) (list_concat (aList nil) X X)))

(assert (forall ((X1 U) (L1 U) (L2 U) (L3 U))
  (=> (and (list_concat L1 L2 L3) ((_ is aList) L1) ((_ is aList) L3))
    (list_concat
      (aList (cons X1 (theList L1))) L2 (aList (cons X1 (theList L3))))))
)

```

Figure 12: SMT-LIB representation of the program in Figure 11

$$\begin{array}{c}
\frac{}{[] \triangleright (\text{aList nil}) : \emptyset} \\
\\
\frac{\{t_i \triangleright t'_i : \Phi_i\}_{i=1}^n}{[t_1, \dots, t_n] \triangleright (\text{aList (cons } t'_1 (\text{cons } t'_2 (\dots \text{cons } t'_n \text{ nil})) \dots)) \triangleright \bigcup_{i=1}^n \Phi_i} \\
\\
\frac{h \triangleright h' \triangleright \Phi_1 \quad t \triangleright t' \triangleright \Phi_2}{[h|t] \triangleright (\text{aList (cons } h' (\text{theList } t')) : \Phi_1 \cup \Phi_2 \cup \{((_ \text{ is aList}) t')\}}
\end{array}$$

Figure 13: Translation rules for lists

```

(declare-datatypes () (
  (U
    (anInt (theInt Int))
    ...
  )
)
)

```

Figure 14: Modified declaration of the universal ADT for integer arithmetic

Lists can be viewed as an algebraic data-type (call it *L*) with two constructors *nil* and *cons*. The constructor *nil* has zero arguments and represents empty lists, and *cons* has two arguments: one referring to the head of the list (a term of type *U*), and one of type *L* referring to the tail of the list.

To implement lists in SMT-LIB, it is therefore natural to declare a new data-type *L* with the *nil* and *cons* constructors. This new data-type will have a mutual dependency on our universal data-type *U*, as defined before, and therefore has to be declared along with *U* as shown Figure 10. Note that in Figure 10, there may be other constructors for *U* coming from the collected functions in Section 3.2, denoted by three dots. The *aList* constructor is introduced to wrap terms of type *L* as a term of the universal type *U*. An example of a Prolog program involving lists is presented in Figure 11. The translation of the program in Figure 11 to SMT-LIB is presented in Figure 12.

A set of inference rules for translating lists in Prolog to SMT-LIB is presented in Figure 13. The first rule in Figure 13 translates the empty list. The second rule represents the translation of a list with explicitly enumerated elements. In this case, the list can be constructed in SMT-LIB as a simple iterated application of *cons*. The final rule directly matches the functional definition of a list. A list in Prolog with *h* as the head, and *t* as the tail, can be constructed using the *cons* constructor in SMT-LIB. This construction is type-correct only if *t* is again a list; thus, the rule adds the side condition $((_ \text{ is aList}) t')$, and unwraps the tail using the *theList* selector. In all the rules in Figure 13, the result is finally wrapped using the *aList* constructor and turned into a *U*-term.

3.6 Integer Arithmetic

We finally introduce inference rules for converting constraints in the background theory of integer arithmetic (e.g., $2 * X + 7 \#> Y$) to their corresponding expressions in SMT-LIB. To represent integers, sim-

$$\begin{array}{c}
\frac{s \triangleright s' : \Phi_1 \quad t \triangleright t' : \Phi_2}{s \star t \triangleright (\text{anInt } (op \text{ (theInt } s') \text{ (theInt } t')))) : \Phi_1 \cup \Phi_2 \cup \{(_ \text{ is anInt } s'), (_ \text{ is anInt } t')\}} \\
\text{where } (\star, op) \in \{(+, +), (-, -), (*, *), (/ , \text{div}), (\text{mod}, \text{mod})\} \\
\\
\frac{s \triangleright s' : \Phi_1 \quad t \triangleright t' : \Phi_2}{s \circ t \triangleright (op \text{ (theInt } s') \text{ (theInt } t')) : \Phi_1 \cup \Phi_2 \cup \{(_ \text{ is anInt } s'), (_ \text{ is anInt } t')\}} \\
\text{where } (\circ, op) \in \{(\# =, =), (\# >, >), (\# >=, >=), (\# <, <), (\# <=, <=)\} \\
\\
\frac{}{I \triangleright (\text{anInt } I) : \emptyset} \quad I \text{ is a decimal non-negative integer} \\
\\
\frac{}{-I \triangleright (\text{anInt } (- I)) : \emptyset} \quad I \text{ is a decimal non-negative integer}
\end{array}$$

Figure 15: Translation rules for expressions and constraints in CLP(\mathbb{Z})

ilarly as with lists we add a new constructor to \mathcal{U} , so that we can wrap integers into a term of type \mathcal{U} . We call this new constructor `anInt`, and define it in Figure 14. We once again note that \mathcal{U} is a universal type and may include other constructors.

The translation rules for integer expressions and Boolean constraints over them are presented in Figure 15. The first rule translates integer-valued functions by recursively translating the sub-terms, unwrapping the result, and applying the corresponding SMT-LIB function. The result is finally wrapped again using `anInt`. The second rule performs the corresponding translation for integer predicates. The other two rules take care of the translation of integer literals.

The translation can be modified easily to model bounded integers, which are used by some Prolog implementations. In this case, instead of `Int` in Figure 14, a bit-vector type like `(_ BitVec 64)` has to be used, and in Figure 15 the corresponding bit-vector operations have to be applied. Bit-vector support in CHC solvers is much less mature than support for mathematical integers, however.

As an example, consider the expression `X #> 7`. According to the translation rule for variables in Figure 6, $X \triangleright X : \emptyset$, and according to the translation rule for integers in Figure 15, $7 \triangleright (\text{anInt } 7) : \emptyset$. Finally, according to the first rule in Figure 15 for translating expressions in integer arithmetic, the whole expression gets translated to `(anInt (+ (theInt X) (theInt (anInt 7))))`, with the side constraint that X is an integer.

4 An SMT-LIB Encoding of the Motivating Example

Combining the different Prolog features that were discussed, Figure 16 gives the SMT-LIB encoding of the Prolog program for computing paths between cities from Section 1. All the commands used in this encoding can be obtained using the translation rules explained throughout this article. The encoding begins by declaring the universal data-type \mathcal{U} . Its constructors include all the atoms and functions that appear in the clauses, in addition to `anInt` and `aList`. After that, the data-type \mathcal{L} is declared for lists, as explained in Section 3.5. The SMT-LIB script continues by declaring the functions `distance` and `path`

appearing in the Prolog program clauses. The rest of the encoding are assertions corresponding to the Prolog rules and facts, and finally the assertion as a clause with head `false`.

A CHC solver like Eldarica [5] can derive the status `unsat` for this SMT-LIB script, which implies that the clauses are contradictory. This can be interpreted as the negation of the last clause being derivable from the other clauses. A CHC solver could also discover the path discussed in Section 1.

When tightening the length bound to `(< (theInt D) 34)`, the problem becomes satisfiable, which can again be verified using a CHC solver.

5 Conclusion

We have presented work towards a new CHC solver front-end that allows input in Prolog format, bridging the gap between Prolog/CLP semantics and SMT-LIB. We are in the process of implementing the defined translation from Prolog to SMT-LIB in our Horn solver Eldarica [5], with the goal of achieving good coverage of the Prolog and CLP features.

The translation defined in this paper should be seen as a starting point, as there are several aspects that require further work, more research, or more discussion in the communities:

- We have only shown the translation rules for some of the most important $\text{CLP}(\mathbb{Z})$ operators. We believe that many other theories and constraints can be handled in a similar fashion.
- We keep typing constraints dynamic, and this way stay close to the actual Prolog semantics. In terms of the efficiency of CHC solvers on the translated program, of course, it could be beneficial to perform some amount of type inference upfront. This way, one could translate integer variables in Prolog to native SMT-LIB `Int` variables, etc. However, CHC solvers with support for algebraic data-types tend to perform such type inference themselves, so that the payoff from being more clever during the translation is unclear.
- Our translation includes all typing constraints as assumptions (Figures 7 and 9), i.e., the well-typedness of a Prolog program is assumed but not verified. It is not entirely obvious how correct typing should be asserted in the SMT-LIB representation, however. In the list example in Figure 11, for instance, note that the first clause implies that the second and third argument of `list_concat` can be terms of any kind, whereas the second clause relies on the third argument being a list. The clauses therefore entail ill-typed statements like `list_concat([X|[]], 42, [X|42])`. A CLP engine performing backward chaining will, however, not run into any typing errors.
- There are several aspects of Prolog semantics that are challenging in a translation to SMT-LIB. Those include, in particular, the *missing occurs-check* in equations, as well as *cuts*. It is unclear whether those features can or should be translated faithfully to SMT-LIB semantics.

Finally, it will be interesting to evaluate solver performance for the different design choices in the translation, for different CHC solvers, and to compare to the performance of state-of-the-art CLP engines. We have not done such a comparison yet due to the preliminary state of the implementation of the front-end. While we generally assume CLP to be significantly more efficient on classical constraint satisfaction problems than CHC, there might also be areas in which the abstraction-based algorithms used in CHC solvers have advantages.

```

1 (declare-datatypes () (
2   (U
3     (anInt (theInt Int))
4     (aList (theList L))
5     (waypoint (waypoint_1 U) (waypoint_2 U))
6     tehran vienna paris munich rome lausanne
7   )
8   (L
9     nil
10    (cons (head U) (tail L))
11  )
12 )
13 )
14 (declare-fun distance (U U U) Bool)
15 (declare-fun path (U U U U) Bool)
16
17 (assert (distance tehran vienna (anInt 31)))
18 (assert (distance vienna paris (anInt 10)))
19 (assert (distance vienna munich (anInt 3)))
20 (assert (distance paris munich (anInt 10)))
21 (assert (distance paris rome (anInt 11)))
22 (assert (distance lausanne rome (anInt 6)))
23 (assert (distance lausanne munich (anInt 4)))
24 (assert (distance tehran paris (anInt 42)))
25 (assert
26   (forall ( (A U) (B U) (D U) )
27     (=> (distance B A D) (distance A B D))
28   )
29 )
30 (assert
31   (forall ( (A U) )
32     (path A A (anInt 0) (aList (cons (waypoint A (anInt 0)) nil)))
33   )
34 )
35 (assert
36   (forall ( (A U) (B U) (C U) (D U) (N U) (P U) (Q U) )
37     (=>
38       (and
39         (path A B P N) (distance B C Q)
40         (= D (anInt (+ (theInt P) (theInt Q))))
41         ((_ is aList) N) ((_ is anInt) P) ((_ is anInt) Q)
42       )
43       (path A C D (aList (cons (waypoint C D) (theList N))))
44     )
45   )
46 )
47 (assert
48   (forall ( (D U) (X U) )
49     (=>
50       (and (path tehran munich D X) (< (theInt D) 40) ((_ is anInt) D))
51       false
52     )
53   )
54 )
55
56 (check-sat)

```

Figure 16: The SMT-LIB encoding of the motivating example introduced in the introduction

References

- [1] *The Golem Horn Solver*. Available at <https://github.com/usi-verification-and-security/golem>.
- [2] *RInGen: Regular Invariant Generator*. Available at <https://github.com/Columpio/RInGen>.
- [3] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2017): *The SMT-LIB Standard: Version 2.6*. Technical Report, Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.
- [4] W. Chen & D. S. Warren (1996): *Tabled Evaluation with Delaying for General Logic Programs*. *Journal of the ACM* 43(1), pp. 20–74, doi:10.1016/0304-3975(89)90088-1.
- [5] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn Solver*. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.
- [6] ISO (1995): *ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core*. Available at <http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+13211%2D1%3A1995>; <http://www.iso.ch/cate/d21413.html>.
- [7] J. Jaffar & J.-L. Lassez (1987): *Constraint Logic Programming*. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '87*, Association for Computing Machinery, New York, NY, USA, p. 111–119, doi:10.1145/41625.41635.
- [8] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2014): *SMT-Based Model Checking for Recursive Programs*. In Armin Biere & Roderick Bloem, editors: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, Lecture Notes in Computer Science 8559*, Springer, pp. 17–34, doi:10.1007/978-3-319-08867-9_2.
- [9] Markus Triska (2012): *The Finite Domain Constraint Solver of SWI-Prolog*. In Tom Schrijvers & Peter Thiemann, editors: *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings, Lecture Notes in Computer Science 7294*, Springer, pp. 307–316, doi:10.1007/978-3-642-29822-6_24.
- [10] Mark Wallace (2002): *Constraint Logic Programming*, pp. 512–532. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/3-540-45628-7_19.
- [11] Jan Wielemaker, Tom Schrijvers, Markus Triska & Torbjörn Lager (2010): *SWI-Prolog*. arXiv:1011.5332.