aspen-tree: A system for non-ground meta-programming in ASP (Preliminary Report)

Amadé Nemes[1], Torsten Schaub[1,2]

[1]University of Potsdam, Germany
[2]Potassco Solutions GmbH, Germany

Abstract

Meta-programming is a powerful technique in Answer Set Programming (ASP) for prototyping language extensions and exploring alternative semantics. While traditional approaches predominantly operate on ground logic programs, many language design tasks require the manipulation of non-ground source code prior to instantiation. We introduce *aspen-tree*, a novel system for the reification, transformation, and analysis of non-ground ASP programs. Built upon the *tree-sitter* parsing library, *aspen-tree* provides a language-independent and iterative framework for mapping source code to a reified fact base. These facts are then processed by *clingo* using meta-encodings to generate high-level transformation instructions, which are applied via a templating engine to update the source code. We demonstrate the utility of *aspen-tree* through two case studies on program analysis and transformation in ASP-Core-2.

1. Introduction

A cornerstone of the success of Answer Set Programming (ASP; [1]) is its high-performance solving engines, ranging from pioneering systems like *smodels* [2] and *dlv* [3] to the more recent *clingo* system [4]. However, while their application programming interfaces have become increasingly sophisticated, the internal architectures of these systems remain highly optimized and inherently rigid, hindering the exploration of new ASP extensions. This limitation can be overcome through meta-programming [5, 6, 7, 8, 9, 10, 11, 12], which provides an accessible and flexible environment for prototyping alternative language designs natively within ASP. To achieve this, meta-programming relies on reification: the representation of a logic program's syntax as a set of facts. These reified facts are combined with a meta-encoding that implements the desired semantics by reinterpreting the original program's components.

Despite its benefits, existing meta-programming techniques in ASP predominantly operate on ground logic programs. However, in practice, expanding the language with novel constructs requires manipulating the non-ground source code prior to instantiation. To date, no general-purpose meta-programming framework has existed to support such pre-grounding transformations natively within ASP. To bridge this gap, we introduce *aspen-tree*,[1] a novel system specifically designed for the reification, transformation, and analysis of non-ground ASP programs. Demonstrating its practical viability, *aspen-tree* already serves as a core component of the *metasp* framework [13, 12], where it successfully drives a preprocessing pipeline to extend *clingo* with complex new language constructs.

Aspen-tree is built upon *tree-sitter*,[2] a widely adopted parser generator library with grammars for hundreds of programming languages. Consequently, *aspen-tree* is generally language-independent; while our primary focus is on non-ground meta-programming in ASP, the system's capabilities extend seamlessly to any language with an available *tree-sitter* grammar. In particular, *tree-sitter* generates incremental parsers, allowing syntax trees to be updated efficiently in response to localized source edits. Through its integration with the *tree-sitter* API, *aspen-tree* captures these incremental changes to ensure that the reified fact base accurately reflects the source code with minimal computational overhead.

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026) co-located with the Federated Logic Conference (FLoC 2026).

amade.nemes@uni-potsdam.de (A. Nemes); torsten.schaub@uni-potsdam.de (T. Schaub)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]https://github.com/potassco/aspen-tree
[2]https://tree-sitter.github.io/tree-sitter

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

CEUR-WS.org/Vol-4311/ASPOCP_regular_07.pdf

Given a specific grammar, *aspen-tree* operates through a multi-stage workflow. Initially, the source code is parsed into a syntax tree and reified into a set of ASP facts. These facts are then supplied to *clingo* alongside user-defined meta-encodings. This process yields a stable model containing high-level instructions, such as `aspen(edit(...))` for code transformations or `aspen(exception(...))` for error reporting. Finally, edits are applied to the source via a templating engine, triggering an incremental reparse to update the internal tree and fact base. This cycle can be repeated to chain multiple transformations using `aspen(next_program(...))`.

The remainder of the paper is structured as follows. Section 2 provides the necessary background on *tree-sitter* and details the reified fact representation used by *aspen-tree*. In Section 3, we describe the system's architecture, its workflow, and the templating mechanism used for source transformations. Section 4 illustrates the practical utility of *aspen-tree* through case studies on program analysis and transformation, specifically focusing on safety checks and syntactic shortcuts in ASP-Core-2. Finally, we discuss related work and future research directions in Section 5.

2. Background

While a mature *tree-sitter* grammar for *clingo* exists,[3] our examples use a minimalistic grammar[4] adhering to the ASP-Core-2 standard [14] to simplify the presentation. This underscores the generality of our approach: *aspen-tree* seamlessly supports any language with a *tree-sitter* grammar, including various ASP dialects and even languages beyond ASP. Swapping or extending grammars is effortless and requires no architectural changes.

Our grammar for ASP-Core-2 follows the EBNF specification in [14] verbatim, except that supertype nodes such as `term` and `head` are hidden to reduce tree height. Listing 1 illustrates the resulting parse tree for the single-rule program "`1 <= {a;b}.`" as generated by *tree-sitter*. The tree nodes are listed

```
0:0  - 1:0    source_file
0:0  - 0:11     statement
0:0  - 0:11       rule
0:0  - 0:10         choice
0:0  - 0:4            lower
0:0  - 0:1              number '1'
0:2  - 0:4              relation '<='
0:5  - 0:6            "{"
0:6  - 0:9            choice_elements
0:6  - 0:7              choice_element
0:6  - 0:7                classical_atom
0:6  - 0:7                  identifier 'a'
0:7  - 0:8              ";"
0:8  - 0:9              choice_element
0:8  - 0:9                classical_atom
0:8  - 0:9                  identifier 'b'
0:9  - 0:10           "}"
0:10 - 0:11         "."
```

Listing 1: Parse tree of the program "`1 <= a;b`" generated by *tree-sitter*.

in depth-first order; each line represents a node, prefixed by its textual range (line and column) and indented according to its depth in the tree. Leaf nodes additionally include their source text and may be unnamed. Since grammars typically define punctuation and delimiters as unnamed nodes, their removal effectively yields an abstract syntax tree.

Listing 2 shows the corresponding reified ASP facts produced by *aspen-tree* for this parse tree. The reification begins with a global description: *aspen-tree* assigns each source a unique identifier (e.g., `s(0)`)

[3] https://github.com/potassco/tree-sitter-clingo
[4] https://github.com/potassco/tree-sitter-aspcore2

```
language(s(0),"aspcore2"). source_root(s(0),0).
node(0). type(0,"source_file"). child(0,1).
  node(1). type(1,"statement"). child(1,2).
    node(2). type(2,"rule"). child(2,(3;4)).
      node(3). type(3,"choice"). child(3,(5;6;7;8)). next_sibling(3,4).
        node(5). type(5,"lower"). child(5,(16;17)). next_sibling(5,6).
          node(16). type(16,"number"). leaf_text(16,"1"). next_sibling(16,17).
          node(17). type(17,"relation"). leaf_text(17,"<=")
        node(6). leaf_text(6,"{"). next_sibling(6,7).
        node(7). type(7,"choice_elements"). child(7,(9;10;11)).
          node(9). type(9,"choice_element"). child(9,14). next_sibling(9,10).
            node(14). type(14,"classical_atom"). child(14,15).
              node(15). type(15,"identifier"). leaf_text(15,"a").
          node(10). leaf_text(10,";"). next_sibling(10,11).
          node(11). type(11,"choice_element"). child(11,12).
            node(12). type(12,"classical_atom"). child(12,13).
              node(13). type(13,"identifier"). leaf_text(13,"b").
        node(8). leaf_text(8,"}"). next_sibling(7,8).
      node(4). leaf_text(4,".").
```

Listing 2: *Aspen-tree* fact representation for parse tree of program "`1 <= a;b.`".

and links it to its language and parse tree root node via predicates `language/2` and `source_root/2`. Node identities are established by `node/1`, while the tree structure and sibling order are captured by `child/2` and `next_sibling/2`, respectively. Finally, `type/2` and `leaf_text/2` define node categories and leaf content.

By lifting the program's syntax into this reified format, *aspen-tree* leverages the full expressive power of ASP for meta-programming. These meta-encodings allow us to reason over the reified program structure to derive predicates that drive source transformations, diagnostic reporting, and further transformation cycles.

3. System Architecture and Workflow

Figure 1 illustrates the architecture and workflow of our *aspen-tree* system, which operates by reifying an input source and applying meta-encodings over its reified representation. To demonstrate the system's functionality, we walk through a transformation of our running example: "`1 <= {a;b}.`". In ASP-Core-2, the base language supports choice rules only with upper bounds, treating lower bounds as syntactic sugar. Our example transformation normalizes these rules by rewriting lower-bound constraints into their equivalent upper-bound representations.

The workflow begins in the parsing layer, where a *tree-sitter* parser for the target language generates a parse tree from the input source. Listing 1 shows the resulting tree for our example rule. This tree is then reified into the set of ASP facts shown in Listing 2 and added to the system's fact store.

Next, these facts are fed into the transformation layer alongside user-defined meta-encodings. To facilitate development, *aspen-tree* includes a library of auxiliary encodings defining common predicates derived from the core fact representation. Users can selectively include these utilities to streamline the implementation of their transformations. Finally, the reified facts and meta-encodings are passed to *clingo* to compute a stable model, which is required to be unique to provide unambiguous transformation instructions.

This stable model is scanned for instances of predicate `aspen/1`, which define operations to be executed by *aspen-tree*.

- The atom `aspen(replace(node(Node),Template))` describes an edit to be made to the source code. The `Template` argument may be any *clingo* symbol. This template symbol is

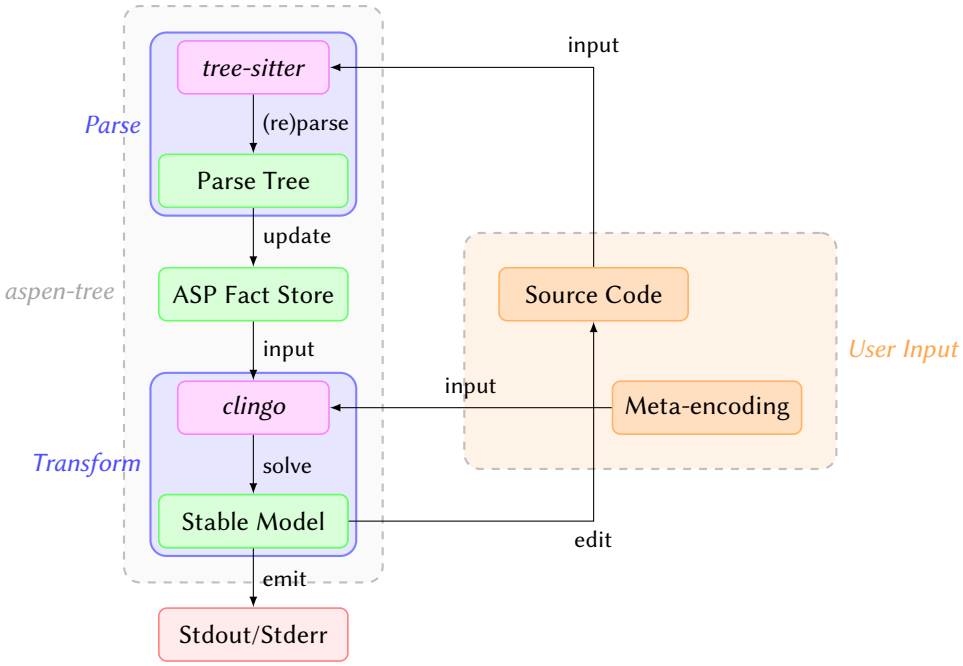


Figure 1: System diagram of *aspen-tree*.

evaluated according to a set of rules described below, and its resulting value replaces the textual span of `Node` in the source code.

- Atoms `aspen(append(node(Node),Template))` and `aspen(prepend(node(Node),Template))` behave similarly, but instead of replacing the textual span of `Node`, they append or prepend the evaluated value of the template symbol to it.
- Atoms `aspen(print(Template))`, `aspen(log(node(Node),Lvl,Template))`, and `aspen(exception(node(Node),Template)` print, log, and raise exception messages, respectively.
  The optional first argument `node(Node)` of `log/3` and `exception/2` may be supplied to prefix the log or exception message with the textual range (line, column) of the specified node.

The `Template` arguments of the operations above are recursively evaluated as follows:

Symbols If a template is of the form `node(Node)`, it evaluates to the source text spanned by that node. All other basic symbols are converted directly to their string representation.

Operators *Aspen-tree* uses nested lists of templates, which are either the empty tuple `()` or a pair `(Head, Tail)` where `Head` is a template and `Tail` is a list.

A *format symbol*, `format(FormatStr, List)`, evaluates by applying the Python `format()` method to the string `FormatStr` using the evaluated elements of `List`.

A *join symbol*, `join(Sep, List)`, evaluates by applying the Python `join()` method to the evaluated elements of `List` with the separator string `Sep`.

Once all edit symbols are collected and their template arguments evaluated, *aspen-tree* applies the resulting modifications to the source code.

```
1  #program choice_lower_sugar.
2  mirror("<",">";"<=",">=";"=","=";"!=","!=").
3  mirror(B,A) :- mirror(A,B).

5  aspen(replace(node(L),"");
6        append(node(C),format(" {0} {1}",(S',(node(T),())))))) :-
```

```
7    type(C,"choice"), child(C,L), type(L,"lower"),
8    child(L,T), child(L,O), next_sibling(T,O), leaf_text(O,S), mirror(S,S').
```

Listing 3: Meta-encoding transforming choice rules with lower bound to choice rules with upper bound.

To illustrate this process with our running example, Listing 3 presents the meta-encoding that implements the lower-bound rewriting transformation. Supplying the program `choice_lower_sugar` to the system triggers the grounding of this encoding over the reified fact store. Lines 2-3 define the mirror images of the relations bounding choice atoms. The rule spanning Lines 5–8 executes the actual program transformation. Its body performs pattern matching over the reified syntax tree through a sequence of joins on the `child/2` predicate. This structural query identifies choice rules restricted by lower bounds and determines the appropriate flipped bounding relation via the `mirror/2` predicate.

When grounded over the reified facts of our running example, the rule yields the following instances:

```
aspen(replace(node(5),""))
aspen(append(node(3),format(" {0} {1}",(">=",(node(16),()))))))
```

The first atom instructs *aspen-tree* to delete the original lower bound by replacing the textual span of node 5 (`"1 <="`) with an empty string, temporarily reducing the program to `"{a;b}."`. The second atom appends a new string to the choice elements at node 3 (`"{a;b}"`). This string is generated by evaluating the template `format(" {0} {1}",(">=",(node(16),())))`. Specifically, the nested list provided as the second argument evaluates to the sequence of `">="` and `"1"`. These values are positionally inserted into the format string to yield `" >= 1"`. Appending this evaluated string to the choice atom completes the transformation, resulting in the normalized program `"{a;b} >= 1."`

Although our running example uses independent operations, the outcome of multiple edits generally depends on their execution order. Since a template may reference the text of a node that is itself subject to change, *aspen-tree* must explicitly manage these inter-dependencies. To ensure a consistent outcome, the system analyzes these relations to establish a topological ordering of all pending edits. If cyclical dependencies are detected, making the intended result ambiguous, the system raises an error.

After all modifications are applied, the updated source code is returned to the parsing layer, where it is efficiently reparsed by *tree-sitter*, keeping as much of the unchanged parse tree as possible. *Aspen-tree* leverages the *tree-sitter* API to identify structural differences between the updated parse tree and its prior state. Newly formed subtrees are reified and integrated into the system's fact store, linked to the preserved portions of the tree via `child/2` and `next_sibling/2` predicates. Conversely, when subtrees are deleted, *aspen-tree* isolates the corresponding facts and purges them from the fact store.

In our running example, *tree-sitter* recognizes that the subtrees representing the choice elements `"{a;b}"` and the terminal period `"."` remain unchanged, allowing their reified facts to be preserved. Conversely, the subtree rooted at node 5 is marked for deletion, while a new subtree spanning the text `">= 1"` is identified. Consequently, *aspen-tree* updates the fact store by removing the following atoms:

```
child(3,5). next_sibling(5,6).
node(5). type(5,"lower"). child(5,(16;17)). next_sibling(5,6).
node(16). type(16,"number"). leaf_text(16,"1"). next_sibling(16,17).
node(17). type(17,"relation"). leaf_text(17,"<=")
```

At the same time, the following facts are added to represent the new structure:

```
child(3,19). next_sibling(8,19).
node(19). type(19,"upper"). child(19,(20;21)).
node(20). type(20,"relation"). leaf_text(20,">="). next_sibling(20,21).
node(21). type(21,"number"). leaf_text(21,"1").
```

The integration of this new subtree into the existing parse tree is established through the new facts `child(3,19)` and `next_sibling(8,19)`.

Following these updates, the control flow typically terminates. However, if the most recent stable model includes an atom of the form `aspen(next_program(Name,Args))`, *aspen-tree* initiates a new transformation cycle. Control loops back to the transformation layer, passing the updated fact store and the activated encodings to *clingo*, which then grounds the specified program `Name` with `Args`, computes

its stable model, and proceeds as before. This iterative mechanism is crucial for composing modular transformation pipelines. It allows developers to chain multiple meta-encodings and specify complex termination conditions directly within ASP. For instance, returning to our lower-bound normalization example, we can append the following fact to the encoding:

```
aspen(next_program("choice_sugar",())).
```

This instructs *aspen-tree* to follow the lower-bound elimination with the transformation detailed in Listing 4 below. By relying on the guarantee that choice rules now only possess upper bounds, the subsequent encoding can safely rewrite them without handling lower bounds as a special case.

4. Case Study: ASP-Core-2

To demonstrate the versatility of *aspen-tree*, we present two advanced applications of its meta-programming capabilities. First, we detail a transformation meta-encoding that compiles away choice rules. Second, we provide a diagnostic meta-encoding that verifies the safety of rules.

4.1. Rewriting Choice Rules to Disjunctive Rules

4.1.1. Definition of Choice Rules

The semantics of choice rules in ASP-Core-2 [14] is not defined directly, rather, it is specified as a syntactic shorthand for a set of rules. Furthermore, this translation applies exclusively to choice rules bounded solely by an (optional) upper bound. Lower bounds are treated as an independent syntactic shorthand, which is eliminated during preprocessing by the meta-encoding shown in Listing 3.

We briefly recall this translation, incorporating minor adaptations. Consider a choice rule of the form

$$C \prec u \leftarrow b_1, \ldots, b_n \tag{1}$$

where u is a term, each b_i is a literal, and C is a set of choice elements of the form $a : l_1, \ldots, l_k$ with atom a and literals l_j. For any atom q, let $\bar{q}$ denote the term obtained by replacing q's predicate symbol with a function symbol of the same name and arity. Furthermore, let $\hat{q} = h(\bar{q})$ and $\widehat{\neg q} = h(-\bar{q})$, where h is a fresh predicate symbol. Then, the choice rule in (1) translates to the disjunctive rules

$$a \mid \hat{a} \leftarrow b_1, \ldots, b_n, l_1, \ldots, l_k \tag{2}$$

for each choice element $(a : l_1, \ldots, l_k) \in C$, along with the constraint

$$\leftarrow b_1, \ldots, b_n, \textbf{not}\ \#\text{count}\{\bar{\hat{a}} : a, l_1, \ldots, l_k \mid (a : l_1, \ldots, l_k) \in C\} \prec u. \tag{3}$$

For instance, the following rule from [14]

$$\{p(a) : q(2); \neg p(a) : q(3)\} \leq 1 \leftarrow q(1) \tag{4}$$

is translated into the following set of rules:

$$\begin{aligned} p(a) \mid h(\bar{p}(a)) &\leftarrow q(1), q(2) \\ \neg p(a) \mid h(-\bar{p}(a)) &\leftarrow q(1), q(3) \\ &\leftarrow q(1), \textbf{not}\ \#\text{count}\{\bar{h}(\bar{p}(a)) : p(a), q(2); \bar{h}(-\bar{p}(a)) : \neg p(a), q(3)\} \leq 1. \end{aligned}$$

4.1.2. Meta-Encoding

Listing 4 presents an *aspen-tree* meta-encoding that transforms choice rules of the form (1) into the disjunctive rules and integrity constraints defined in (2) and (3), respectively.

The initial lines (1–3) import utility encodings from *aspen-tree*, organized by language: the `generic` folder contains grammar-independent encodings, while `asp-core-2` provides those specific to the ASP-Core-2 grammar. The imported encodings define: (1) `named_child_index(Node,Idx)`, mapping a

```
1  #include "aspen/generic/derived_predicates/named_child_index.lp".
2  #include "aspen/generic/queries/make_list.lp".
3  #include "aspen/aspcore2/derived_predicates/symbol_signature.lp".

5  #program choice_sugar.

7  choice(R,E) :-
8    type(R,"rule"),
9    child(R,C), type(C,"choice"),
10   child(C,ES), type(ES,"choice_elements"),
11   child(ES,E), type(E,"choice_element").

13 e_head(E,A) :- choice(R,E), child(E,A), type(A,"classical_atom").

15 _templ(body(R),node(B)) :- choice(R,E), child(R,B), type(B,"body").
16 _templ(upper(R),node(U)) :-
17   choice(R,E), child(R,C), child(C,U), type(U,"upper").
18 _templ(cond(E),format(", {}",(node(LS),(())))) :-
19   choice(R,E), child(E,LS), type(LS,"naf_literals").

21 templ(I,T) :- _templ(I,T).
22 templ(I,"") :- default(I), not _templ(I,_).
23 default(body(R);upper(R);cond(E)) :- choice(R,E).

25 templ(hat(E),format("not_choose({0})",(node(A),(())))) :- e_head(E,A).

27 aspen(exception(node(A),"Predicate not_choose/1 is system reserved.")) :-
28   type(A,"classical_atom"), symbol_signature(A,("not_choose",1)).

30 templ(rule(E),format("{0} | {1} :- {2}{3}.", (node(A),(H,(B,(LS,(()))))))) :-
31   choice(R,E), e_head(E,A), templ(hat(E),H),
32   templ(body(R),B), templ(cond(E),LS).

34 templ(agg_element(E),format("{0} : {1}{2}",(H,(node(A),(LS,(())))))) :-
35   choice(R,E), e_head(E,A), templ(hat(E),H), templ(cond(E),LS).

37 aspen(query(make_list(agg_elements(R),T,I))) :-
38   choice(R,E), named_child_index(E,I), templ(agg_element(E),T).

40 templ(constraint(R), format(":- {0}, not #count{{ {1} }} {2}.",(B,(J,(U,(()))))) :-
41   choice(R,E), templ(body(R),B), templ(upper(R),U),
42   aspen(return(make_list(agg_elements(R)),C)), J=join("; ",C).

44 aspen(query(make_list(rules(R), T, I))) :-
45   choice(R,E), named_child_index(E,I), templ(rule(E),T).

47 aspen(replace(node(R),join("\n",(T, C)))) :-
48   choice(R,_), aspen(return(make_list(rules(R)), C)),
49   templ(constraint(R),T).
```

Listing 4: Meta-encoding rewriting choice rules of form (1) into rules as in (2) and (3).

node to its index among named siblings, (2) a utility encoding for constructing lists from predicates, and (3) the ASP-Core-2 predicate `symbol_signature(Node,(Name,Arity))`, identifying the name and arity of function or atom nodes.

The core of the encoding begins in Lines 7–13 by defining predicates `choice(R,E)` and `e_head(E,A)` to link choice rules to their elements and, in turn, elements to their head atoms.

Subsequently, template symbols are defined to facilitate the creation of replacement rules. A predicate `templ(I,T)` maps each template symbol `T` to a unique identifier `I`. First, in Lines 15–19, we define template symbols for the body of a choice rule, its upper bound, and the condition of each choice element. Since these constructs are optional, Lines 21–23 set their default value to the empty string.

Next, on Line 25, for any choice element head a, we define the template symbol of $\hat{a}$, using the unary predicate symbol `not_choose` for the role of h. Note that the textual representation of q is identical to

$\bar{q}$ (and that of $\neg q$ to $-\bar{q}$). Thus, for $a = q$ or $a = \neg q$, we obtain $\hat{a}$ by evaluating the template symbol. To ensure correctness, Lines 27–28 verify that `not_choose/1` is not used in the input program, emitting a located exception if it is.

With the necessary template symbols in hand, we define the template for the rules of (2) in Lines 30–32. Then, in Lines 34–35, we instantiate the template for the aggregate elements of integrity constraint (3). These aggregate elements are collected into a list using the `make_list.lp` utility encoding. This encoding takes a set of `aspen(query(make_list(ID,E,IDX)))` atoms and derives a single `aspen(return(make_list(ID),C))` atom, where `C` is the constructed list (and `ID` is an identifier shared by the atoms, and `E`, `IDX` is a desired element and its index in the list). The list is then joined with the separator ";" and used to instantiate the template for the integrity constraint (3) in Lines 40–42.

Finally, in Lines 44–49, all rules from (2) and the integrity constraint from (3) are joined, separated by newlines. *Aspen-tree* is then instructed to replace the original choice rule with the resulting template symbol.

Giving our example choice rule in (4)

```
{p(a) : q(2); -p(a) : q(3)} <= 1 :- q(1).
```

as the input program to *aspen-tree*, along with our encoding `choice_sugar` yields the following transformed program:

```
 p(a) | not_choose( p(a)) :- q(1), q(2).
-p(a) | not_choose(-p(a)) :- q(1), q(3).
:- q(1), not #count{ not_choose( p(a)) :  p(a), q(2);
                     not_choose(-p(a)) : -p(a), q(3) } <= 1.
```

4.2. Safety check in ASP-Core-2

4.2.1. Safety Definition

We briefly recall the definition of safety from [14], noting that this definition applies strictly to the core language, requiring choice rules to be compiled away beforehand.

Given a set of variables V and a set of atoms $B = \{b_1, \ldots, b_n\}$, a variable $v \in V$ is inductively defined as *bound* by B if v occurs outside the scope of arithmetic operators in some $b_i \in B$, and b_i is either

1. a classical atom,
2. a built-in equality atom $t = u$ or $u = t$, where the binding occurrence of v is in u, and all variables from V occurring in t are bound by $B \setminus \{b_i\}$, or
3. an aggregate atom #aggr $E = u$, where the binding occurrence of v is in u, and all variables from V occurring in E are bound by $B \setminus \{b_i\}$.

A rule (or a weak constraint) r is considered *safe* if its set of global variables V is bound by the set of non-negated body atoms in r, and, for each aggregate element in r containing a set of variables W, the subset of local variables $W \setminus V$ is bound by the non-negated atoms in the element's condition.

Consider the following example rule from [14]:

```
p(X,Y) :- q(X), #sum{S,X: r(T,X), S = (2 * T) - X} = Y.
```

The global variable `X` is bound by the body atom `q(X)`. Consequently, the global variable `Y` is bound by the aggregate atom, since the only global variable within the aggregate element is `X`, which is already bound by `q(X)`. To establish the overall safety of the rule, we must also identify bindings for the local variables within the aggregate. The local variable `T` is bound by the atom `r(T,X)`. Consequently, the built-in equality atom establishes the safety of `S`, since `T` is the only local variable appearing on the right-hand side of the equation. Having identified valid bindings for all local and global variables, we conclude that the rule is safe.

In contrast, consider the following unsafe rule, also from [14]:

```
p(X,Y) :- q(X), #sum{S,X: r(T,X), S + X = 2 * T} = Y.
```

Here, the local variable S is not safely bound because its occurrence in the built-in equality atom is within the scope of the + arithmetic operator. Hence, both the variable and the rule are unsafe.

4.2.2. Meta-Encoding

```
1  #program safety_check.
2  stm(S) :- type(S,("rule";"weak_constraint")).

4  var(V;V,N) :- type(V,"variable"), leaf_text(V,N).

6  atoms(S,A) :- child(S,B), type(B,"body"), child(B,NL),
7    type(NL,"naf_literal"), child(NL,A), not type(A,"naf").

9  atoms(S,A) :- child(S,B), type(B,"body"), child(B,A), type(A,"aggregate").

11 atoms(S,A) :- type(S,"aggregate_element"), child(S,NLS),
12   type(NLS,"naf_literals"), child(NLS,NL),
13   type(NL,"naf_literal"), child(NL,A), not type(A,"naf").

15 pos_atoms(S,A) :- atoms(S,A), #false: next_sibling(N,A), type(N,"naf").

17 eq(A,T,U;A,U,T) :- pos_atoms(_,A), type(A,"builtin_atom"), child(A,U),
18    next_sibling(U,R), leaf_text(R,"="), next_sibling(R,T).

20 eq(A,E,U) :- pos_atoms(_,A), type(A,"aggregate"),
21   child(A,B), type(B,"upper"), child(B,R), leaf_text(R,"="),
22   child(B,U), next_sibling(R,U), child(A,E), type(E,"aggregate_elements").

24 desc(S,S) :- atoms(S,_).
25 desc(arith,B) :- type(B,("binary_operation";"unary_operation")).
26 desc(T,T;U,U) :- eq(A,T,U).
27 desc(A,A) :- type(A,"classical_atom").
28 desc(A,C) :- desc(A,P), child(P,C).

30 scope(V,"global") :- var(V), stm(S), desc(S,V),
31   not desc(S',V): atoms(S',_), desc(S,S'), S'!=S.

33 scope(V',"global") :- var(V,N), scope(V,"global"),
34   stm(S), desc(S,V), desc(S,V'), var(V',N).

36 scope(V,"local") :- var(V), not scope(V,"global").

38 binds(S,A,V') :- binds(S,A,V), var(V,N), var(V',N), desc(S,V').

40 binds(S,A,V) :- pos_atoms(S,A), type(A,"classical_atom"),
41   var(V), not desc(arith,V), desc(A,V).

43 other_binds(S,A,V) :- pos_atoms(S,A),  eq(A,T,_),
44   desc(T,V), var(V), binds(S,A',V), A'!=A.

46 binds(S,A,V) :- pos_atoms(S,A), eq(A,T,U), desc(U,V), var(V),
47   scope(V,C), not desc(arith,V),
48   other_binds(S,A,V'): var(V'), scope(V',C), V'!=V, desc(T,V').

50 aspen(exception(node(S),format("Unsafe variables in:\n{}",(node(S),()))));
51       exception(node(V),format("Variable '{}' is unsafe.",(node(V),())))) :-
52   stm(S), desc(S,V), var(V), not binds(_,_,V).
```

Listing 5: Safety check meta-encoding.

Listing 5 presents an *aspen-tree* meta-encoding that verifies the safety of input programs according to this definition. If an unsafe program is detected, it raises an exception with a message detailing the unsafe rules and their associated variables.

Lines 2–4 define auxiliary predicates that provide more compact access to rule, weak constraint, and variable nodes.

The rules on Lines 6–9 use the predicate `atoms/2` to map statements to the set of atoms $b_1, \ldots, b_n$ occurring in their bodies. Using the same predicate, the rule spanning Lines 11–13 maps aggregate elements $t_1, \ldots, t_k : l_1, \ldots, l_m$ to the atoms constituting their condition. From these atom sets, the rule on Line 15 filters out those that are negated.

The rules on Lines 17–22 collect built-in and aggregate equality atoms, using the predicate `eq/3` to associate them with the nodes forming the two sides of the equation. We later rely on this predicate to establish bindings for variables via the last two conditions of the safety definition.

The rules on Lines 24–28 define a descendant relation `desc/2` based on reachability via the `child/2` relation. To minimize its ground instantiation, we restrict `desc/2` to descendants of relevant nodes: atom sets, arithmetic operators, components of equality atoms, and regular atoms.

Next, Lines 30–36 establish whether a variable occurrence has global or local scope. According to Lines 30–33, a variable occurrence is global if it descends from exactly one atom set (i.e. occurs outside an aggregate element), or if another global variable occurrence in the same statement shares its name. Otherwise, the occurrence is considered local.

The remainder of the encoding defines when a variable occurrence `V` is bound by an atom `A` in an atom set `S`, representing this relation with the predicate `binds(S,A,V)`. Line 38 specifies that an atom binding a variable occurrence within an atom set effectively binds all other occurrences of that variable within the same set. Line 40 formalizes variable binding for regular atoms, reflecting the first condition of the safety definition: a variable is bound if it occurs in an atom outside the scope of arithmetic operations. Finally, in Lines 43–48, we define the conditions under which built-in or aggregate equality atoms bind a variable. We first unify the application of the second and third safety conditions to establish variable safety when a variable appears on the right-hand side of an equation. That is, a non-negated (built-in or aggregate) equality atom binds a local (resp. global) variable if it occurs on the right-hand side outside the scope of arithmetic operators, and all local (resp. global) variables on the left-hand side are bound by a distinct atom in the same atom set. It suffices to formalize this condition through rules using the predicate `eq(A,T,U)`. Indeed, when `A` is an aggregate atom, the variable occurrence of interest is invariably on the right-hand side, corresponding to `U`. For built-in atoms, the predicate is defined symmetrically in its last two arguments in Line 17. To implement this, we define the predicate `other_binds(S,A,V)`, which holds if `A` is an equality atom in atom set `S`, variable `V` appears on the left-hand side of `A`, and `V` is bound by a distinct atom `A'` within the same set. Leveraging this helper predicate, the rule spanning Lines 46–48 formalizes the safety condition. That is, the conditional literal in Line 48 ensures that every variable within the same scope on the left-hand side of the equation is bound by a distinct atom in the set.

Finally, Lines 50–52 emit exception messages for all unsafe rules and variables. Each message identifies the specific rule or variable as unsafe and is prefixed with its corresponding textual range in the source code. For instance, processing the unsafe example rule with the safety meta-encoding in *aspen-tree* yields the following exception message:

```
-:1:0-53: Unsafe variables in:
p(X,Y) :- q(X), #sum{S,X: r(T,X), S + X = 2 * T} = Y.
-:1:21-22: Variable 'S' is unsafe.
-:1:34-35: Variable 'S' is unsafe.
```

This message indicates that the rule spanning the textual range in row 1, columns 0–53 is unsafe, specifically identifying the occurrences of variable `S` at positions 1:21–22 and 1:34–35 as the source of the violation.

5. Conclusion

We have introduced *aspen-tree*, a novel system designed to enable the reification and transformation of non-ground source code. By leveraging the *tree-sitter* parsing library, the framework provides a language-independent and iterative workflow where source code is mapped to reified facts, processed via *clingo* meta-encodings, and updated through a robust templating engine. Our case studies on choice

rule normalization and variable safety verification demonstrate that *aspen-tree* effectively supports complex language extensions and diagnostic analysis within the ASP-Core-2 standard. Future research will focus on optimizing the transformation loop through multi-shot solving and extending the system's utility toward real-time static analysis and IDE integration.

6. Declaration on Generative AI

During the preparation of this work, the author(s) used Claude Sonnet 5 to identify and correct grammatical errors, typos, and other writing mistakes. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] V. Lifschitz, Answer Set Programming, Springer-Verlag, 2019. doi:10.1007/978-3-030-24658-7.

[2] P. Simons, I. Niemelä, T. Soininen, Extending and implementing the stable model semantics, Artificial Intelligence 138 (2002) 181–234.

[3] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Transactions on Computational Logic 7 (2006) 499–562.

[4] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory and Practice of Logic Programming 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[5] M. Gelfond, T. Son, Reasoning with prioritized defaults., in: J. Dix, L. Pereira, T. Przymusinski (Eds.), Third International Workshop on Logic Programming and Knowledge Representation, volume 1471 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997, pp. 164–223.

[6] C. Baral, M. Gelfond, Reasoning agents in dynamic domains, in: J. Minker (Ed.), Logic-Based Artificial Intelligence, Kluwer Academic Publishers, Dordrecht, 2000, pp. 257–279.

[7] T. Son, C. Baral, T. Nam, S. McIlraith, Domain-dependent knowledge in answer set planning, ACM Transactions on Computational Logic 7 (2006) 613–657. doi:10.1145/1183278.1183279.

[8] T. Eiter, A. Polleres, Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications, Theory and Practice of Logic Programming 6 (2006) 23–60.

[9] M. Gebser, J. Pührer, T. Schaub, H. Tompits, A meta-programming technique for debugging answer-set programs, in: D. Fox, C. Gomes (Eds.), Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08), AAAI Press, 2008, pp. 448–453.

[10] M. Gebser, R. Kaminski, T. Schaub, Complex optimization in answer set programming, Theory and Practice of Logic Programming 11 (2011) 821–839.

[11] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to build your own ASP-based system?!, Theory and Practice of Logic Programming 23 (2023) 299–361. doi:10.1017/S1471068421000508.

[12] S. Hahn, A. Nemes, J. Romero, T. Schaub, Meta-programming for linear-time temporal answer set programming, Submitted, 2026.

[13] S. Hahn, A. Nemes, J. Romero, T. Schaub, *metasp*, https://github.com/potassco/metasp, 2026.

[14] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory and Practice of Logic Programming 20 (2020) 294–309.