Justification Trees
Enhancing Automated Reasoning through Graphviz Representation

Roy Ratcliffe[1,*]

Abstract

This article presents a brief overview of Skolemised Constraint Answer Set Programming, or s(CASP), a powerful framework designed for resolving complex logical problems. By utilising Skolemisation to convert existential variables into Skolem functions, s(CASP) enables efficient automated reasoning through a clausal format. The important capability of s(CASP) lies in its ability to generate "justification" trees, which elucidate the rationale behind each answer, categorising them as true or false and likely or unlikely. This examination addresses the visualisation of justification trees through Graphviz, facilitating the comprehension of dependencies within the answers. We outline a systematic approach to produce an answer set in JSON format, followed by the parsing and recursive transformation of the data into Graphviz-compatible representations. This exploration not only clarifies the structure of justification trees but also enhances the interpretability of the results obtained from s(CASP) solutions, enriching the accessibility of automated reasoning methodologies in logic programming.

Keywords: Prolog, s(CASP)

What is s(CASP)? It stands for Skolemised Constraint Answer Set Programming. Essentially, it is a sophisticated method for discovering answers to logical problems, where each answer explains why it is true either by proof, assumption or abduction[2].

Skolemisation substitutes the existential variable $\exists y$ in the following with a Skolem function $f(x)$ which enables conversion to clausal form for efficient automated reasoning.

$$\forall x \, \exists y \, P(x, y) \implies \forall x \, P(x, f(x))$$

The s(CASP) tool optionally outputs a "justification" tree, describing *why* an answer is true, or false; likely, or unlikely. Every answer has its own tree comprising nodes and edges. Nodes represent facts and goals. Edges show their dependencies, either supporting or contradicting.

1. The Problem

Start with a super simple example; see below. The example asks, "Who is eligible?" The s(CASP) solver searches for citizens who are *not* criminals.

```
eligible(Who) :- citizen(Who), not criminal(Who).

citizen(alice).
```

*Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1]See more hackery at GitHub.
[2]inference to the best explanation, likely or unlikely

```
criminal(bob).

?- eligible(Who).
```

However, we only present two *known* facts: Alice is a citizen and Bob is a criminal. We cannot say that Alice is a criminal; no evidence exists. Nor can we say that Bob is a citizen. That makes Alice likely eligible, but Bob definitely not.

```
$ swipl -g "[library(scasp/main)]" -- -s0 --tree citizen.pl
```

The scasp tool does not automatically generate the justification tree. It outputs only the bindings by default. Use the `--tree` option to include the tree.

```
% Query
?- eligible(Who).
% □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
%                          Answer 1 (0.000 sec)
% □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
% Justification
   query ←
      eligible(alice) ←
         citizen(alice) □
         not criminal(alice).
% Model
{ citizen(alice),      not criminal(alice),  eligible(alice)
}
% Bindings
Who = alice
```

We want to visualise the justification tree. Justification tree in, Graphviz "dot" graph out [1]. That is the problem in a nutshell. We want a graph with nodes and edges.

2. A Solution

This is the plan in outline: Generate the answer set in JSON. Parse the JSON to extract the justification trees. Recursively convert the tree nodes into Graphviz nodes and edges.

The complete solution exists on GitHub. Go there to see the details in full. This section describes the highlights.

An s(CASP) justification tree is just a nested tree of nodes and their child nodes. Edges represent "implies" relationships. C implies P, or $C \implies P$, where C is the child node and P is the parent node. Read the *inverse* as "because;" that is, P because C, or $P \because C$ in math speak. The tree T is a graph of directed edges E.

$$(P, C) \in E$$

A solution only needs to walk the tree, generating the edges from child to parent in "dot" notation. It needs to convert the nodes to terms and term strings for Graphviz and generate edges from children to their parents. No need to worry about the layout. Graphviz will organise the nodes automatically according to layout constraints.

2.1. Reading a justification tree

Prolog makes this initial step easy. Given a source file *Src*, load the JSON as a dictionary.

```
read_json_dict(Src, Dict) :-
    setup_call_cleanup(open(Src, read, Stream),
                       json_read_dict(Stream, Dict),
                       close(Stream)).
```

The source will **not** be just a tree. It will be a set of justification trees: one tree for each query.

2.2. Walking the tree using a DCG

A tree is a recursive structure of nodes with sub-node children. What do justification trees look like when rendered in JSON? Dictionary objects with a list of sub-dictionary children—in a nutshell. See *Tree* in Figure 1.

A basic tree walk can become a definite-clause grammar (DCG) where the tree or sub-tree is an argument, and its difference list becomes a list of Graphviz directives.

```
json_dot(Dict, Options) -->
    answers_dot(Dict.answers, Options_).

answers_dot(Answers, Options) -->
    sequence(answer_dot(Options), Answers).

answer_dot(Options, Answer) -->
    answer_tree_dot(Answer.tree, Options).

answer_tree_dot(_{node:Node, children:Children}, Options) -->
    { value_term(Node.value, query)
    },
    sequence(answer_tree_query_dot(Options), Children).

answer_tree_query_dot(Options, Answer) -->
    sequence(answer_tree_query_dot(Options), Answer.children).
```

Each answer is a dictionary of pairs. Every dictionary answer has:

- an answer counter;
- a sub-dictionary of bindings;
- a sub-dictionary of constraints;
- a list of model terms;
- a justification tree; and
- a time.

The top level of each answer has a node atom value of `query`. It acts as a root node. The children of the root node constitute its justifications.

2.3. Node values to Prolog terms

Every node has a value. See Figure 1, *Node* class.

Node values take the form of a dictionary after loading the JSON. The JSON becomes a Prolog dictionary of dictionaries and lists of dictionaries.

See the implementation below. It recursively constructs a *Term* given a node's *Value* dictionary. Mapping the arguments using itself leads to recursion. This implies that each argument could itself map to a sub-compound. The logic allows for this.

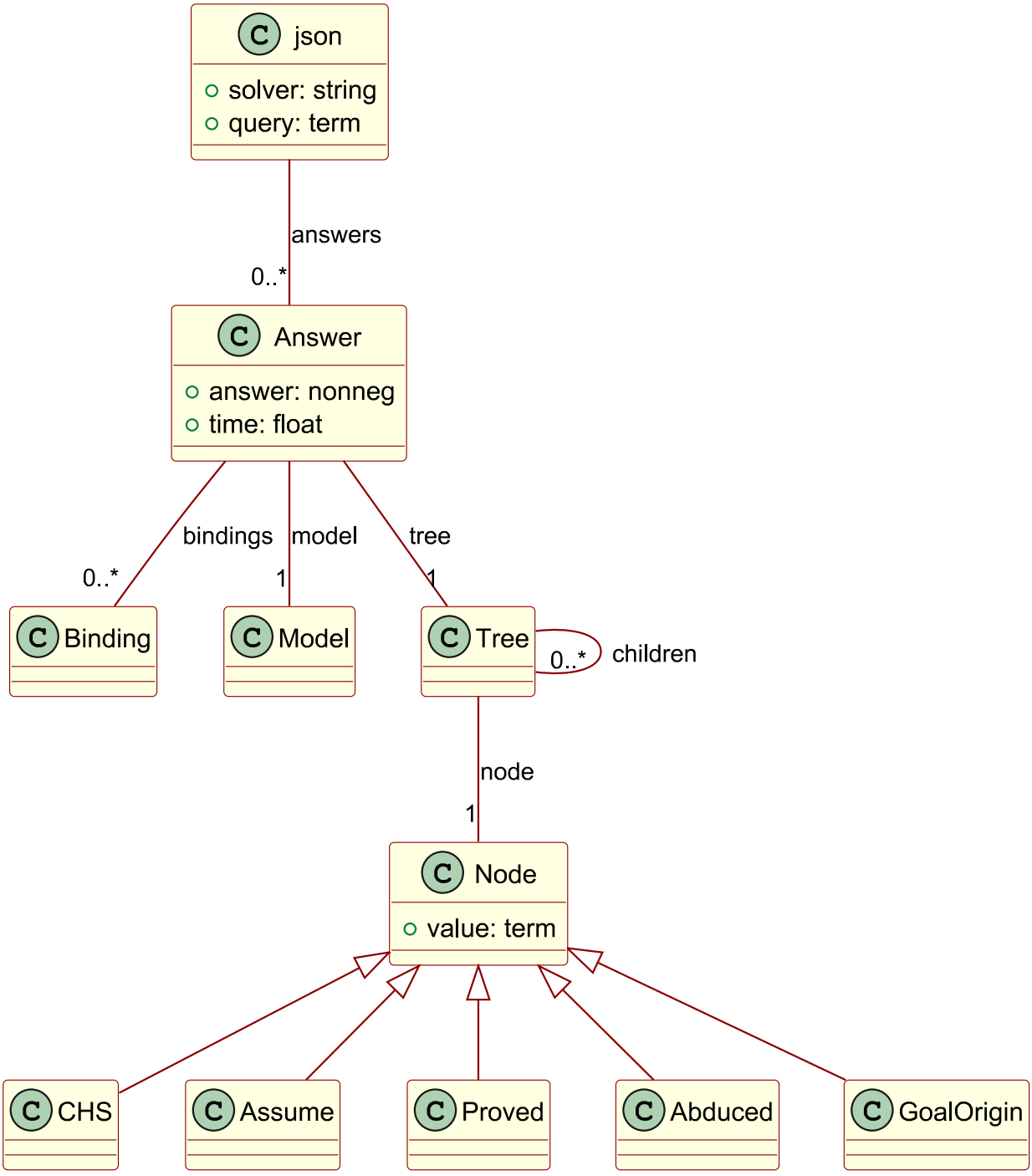

Figure 1: Results of an s(CASP) solver rendered in JSON

```
%!  value_term(+Value:dict, ?Term) is semidet.

value_term(Value, Term), is_dict(Value) => dict_term(Value, Term).
value_term(Value, List), is_list(Value) => maplist(value_term, Value, List).

dict_term(Value, Var), _{type:"var",
                         name:Var} :< Value => true.
dict_term(Value, Atom), _{type:"atom",
                          value:String} :< Value => atom_string(Atom, String).
dict_term(Value, Value), _{type:"number",
                           value:Value} :< Value => true.
dict_term(Value, Rational), _{type:"rational",
                              numerator:Numerator,
                              denominator:Denominator} :< Value =>
    rational(Rational, Numerator, Denominator).
dict_term(Value, Term), _{type:"compound",
                          functor:Functor,
                          args:Args} :< Value =>
    atom_string(Functor_, Functor),
    maplist(value_term, Args, Args_),
    Term =.. [Functor_|Args_].
```

The `value_term/2` predicate converts a Prolog dictionary (parsed from JSON) that represents a term into an actual Prolog term. The predicate deals with cases where the dictionary represents different types of terms, such as variables, atoms, numbers, rational numbers, and compound terms. It recursively processes the arguments of compound terms to construct the final term. The predicate handles the following cases:

- If the input is a dictionary representing a variable, it extracts the variable name and returns it as a Prolog *string*.
- If the dictionary represents an atom, it converts the string value to a Prolog atom.
- If the dictionary represents a number, it returns the number as is.
- If the dictionary represents a rational number, it constructs a rational term from the numerator and denominator.
- If the input is a dictionary representing a compound term, it recursively processes the arguments and constructs the compound term using the functor and the arguments.

The predicate employs dictionary pattern matching (`:<`) to safely extract the relevant fields from the dictionary, and for clarity. It handles nested compounds by recursively mapping the arguments to value terms. The predicate is semi-deterministic, meaning it succeeds at most once; if the input dictionary does not match any of the expected patterns, it fails silently. This allows for safe handling of the JSON structure without raising exceptions for unexpected formats.

The implementation assumes that the JSON structure follows a specific format, where each term is represented as a dictionary with a `type` field indicating whether it is a variable, atom, number, rational, or compound. If the JSON structure changes or new term types are added, the predicate will require updating.

The `value_term/2` predicate is used to convert the JSON representation of terms into Prolog terms, which can then be used in further processing or output—particularly useful in the context of generating a DOT graph from a JSON source produced by s(CASP), where terms represent nodes in the justification tree. It is a crucial part of justification tree processing, allowing the conversion of JSON representations of terms into Prolog terms that can be used to construct the graph structure, and designed to be used in conjunction with the `dict_term/2` helper predicate, which handles the conversion of a Prolog dictionary (parsed from JSON) into a Prolog term. The `dict_term/2` predicate is responsible for converting the dictionary representation of terms into actual Prolog terms.

2.4. Printing message lines

The `scasp_just_dot` module at GitHub adds node and edge styling; true edges receive a solid dark green edge while "likely" edges render as dotted dark green; similarly, false edges render as dark red while unlikely as dotted dark red. The underlying DCG generates "message lines" for use by `print_message_lines/3`.

3. Use Case

So, going back to the original simple "citizen" example. The following commands (i) generate all s(CASP) solutions with justification, (ii) convert the JSON to Graphviz "dot" format, then finally (iii) render the graph as SVG.

```
swipl -g "[library(scasp/main)]" -- -s0 --tree --json=citizen.json citizen.pl
swipl -g "[library(scasp/just_dot_main)]" -- citizen.json
dot -Tsvg citizen.dot -o citizen.svg
```

See the result in Figure 2.

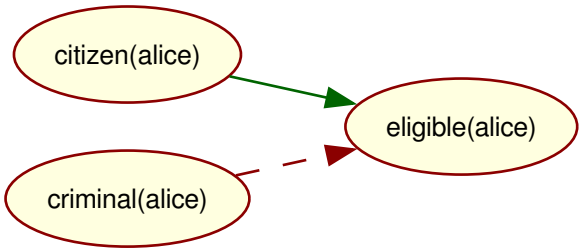

Figure 2: Citizen s(CASP) solution as a graph

Notice the edges. The fact that Alice is a citizen, i.e. `citizen(alice)`, implies that she is eligible. Meanwhile, since no facts exist to disprove that Alice is a criminal, the justification tree shows an "unlikely" negative implication for Alice's eligibility.

4. Conclusions

Justification trees are beneficial. They help a developer to understand why programs behave the way they do. This can prove crucial in applications that require transparency in reasoning. Diagnostics in safety-critical systems is one notable example, but not limited to: compliance checking, protocol analysis, fault diagnosis in medical diagnostics, and so on.

Having the ability to visualise the resulting justification tree helps understand the reasoning behind the logic.

References

[1] Graphviz Team, Graphviz - graph visualization software, accessed on June 13, 2025 (2025).
URL https://graphviz.org/

[2] K. Marple, G. Gupta, Goal-directed execution of answer set programs with constraints, in: Proceedings of the 1st Workshop on Goal-directed Execution of Answer Set Programs (GDE), Vol. 2970 of CEUR Workshop Proceedings, CEUR-WS.org, 2021.
URL https://ceur-ws.org/Vol-2970/gdepaper1.pdf