FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Three Watched Literals

Efficient Propagation for Lazy-Grounding Answer Set Programming Systems

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Lorenz Leutgeb
Matrikelnummer 1127842

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Thomas Eiter
Mitwirkung: Dr. Antonius Weinzierl

Wien, 20. September 2017

Lorenz Leutgeb	Thomas Eiter

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Three Watched Literals

Efficient Propagation for Lazy-Grounding Answer Set Programming Systems

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Lorenz Leutgeb
Registration Number 1127842

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Thomas Eiter
Assistance: Dr. Antonius Weinzierl

Vienna, 20th September, 2017

Lorenz Leutgeb

Thomas Eiter

Erklärung zur Verfassung der Arbeit

Lorenz Leutgeb
Engilgasse 3a, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. September 2017

Lorenz Leutgeb

Acknowledgements

I would like to thank Antonius Weinzierl for his consistent and continuous efforts in supporting me writing this thesis through numerous meetings, openness in discussion and acceptance of my contributions to the Alpha system.

Kurzfassung

Answer Set Programming (ASP) ist ein deklaratives Programmierparadigma in Anlehnung an Prädikatenlogik erster Stufe. ASP-Programme werden üblicherweise durch ASP-Systeme evaluiert, deren Design stark durch Löser (engl. „solver“) für das Erfüllbarkeitsproblem der Aussagenlogik (SAT, von engl. „satisfiability“) inspiriert ist. Ein zentrales Verfahren in diesen Systemen ist die sogenannte *Einheitsresolution* (engl. „unit propagation“), wodurch aus bekannten Wahrheitswerten für Konstanten und der Struktur der Formel neue Wahrheitswerte ableitet werden können.

Im Unterschied zu SAT-Formeln enthalten ASP-Programme jedoch Variablen, wodurch sich das sogenannte *grounding bottleneck* ergibt, welches die praktische Verwendbarkeit von ASP-Systemen einschränkt. An der Entwicklung sogenannter *lazy-grounding* ASP-Systeme wird aktiv geforscht. Durch die Einführung eines dritten Wahrheitswerts *must-be-true* (neben den zwei Boolschen Werten *true* und *false*) kann die Suche nach Lösungen beschleunigt werden. Diese Optimierung hat sich für lazy-grounding ASP-Systeme als besonders wirkungsvoll herausgestellt, sie ist allerdings mit den bewährten Varianten der Einheitsresolution nicht kompatibel, da diese nur die zwei Booleschen Wahrheitswerte berücksichtigen.

Der Hauptteil der Arbeit beschreibt einen möglichen Lösungsansätz der effizienten Einheitsresolution mit drei Wahrheitswerten, insbesondere für lazy-grounding ASP-Systeme, aufbauend auf der *Two Watched Literals* (2WL) Methode. Die neue Methode nennt sich *Three Watched Literals* (3WL).

Die Korrektheit des Algorithmus wird gezeigt. Die erste bekannte Implementierung des Verfahrens – Teil des lazy-grounding ASP-Systems „Alpha“ – wurde zu einer Evaluation anhand von verfügbaren Eingabeprogrammen herangezogen, deren Ergebnisse diskutiert werden.

Abstract

Answer Set Programming (ASP) is a declarative programming paradigm based on first order logic. ASP programs are usually evaluated by ASP systems, whose design is strongly inspired by solvers for the boolean satisfiability problem (SAT). A central method in these systems is *unit propagation*, enabling to derive new truth assignments for constants from known truth assignments, using the structure of the input formula.

Contrary to SAT formulas, ASP programs contain variables, leading to the so-called *grounding bottleneck*, which limits the practical applicability of ASP systems. The development of *lazy-grounding* ASP systems to overcome the grounding bottleneck is an active area of research. By introducing a third truth value *must-be-true* (in addition to the two boolean truth values *true* and *false*) search for solutions can be improved. This optimization was shown to be especially effective in lazy-grounding ASP systems, but it is not compatible with established variants of unit propagation, because they only consider the boolean truth values.

The main part of this work describes an approach for efficient unit propagation with three truth values, especially considering lazy-grounding ASP systems, based on the *Two Watched Literals* (2WL) method. The new approach is called *Three Watched Literals* (3WL).

Soundness of the new algorithm is shown. The first known implementation – part of the lazy-grounding ASP system "Alpha" – was used to evaluate performance with freely available input programs. Results are discussed.

Contents

CHAPTER 1

Introduction

Since the inception of computer programming, different ways of encoding algorithms, structuring data and modeling the real world in computer programs have led to the development of various conceptually diverse programming languages. Families of programming languages that share some properties are usually grouped into so called "programming paradigms". Such paradigms describe the common, most important concepts and lay a framework for a family of languages.

The languages most widely used in industry follow the *imperative* programming paradigm (some more strict, some less), which revolves around instructing the computer step by step. In this work, in contrast to mainstream software engineering, the main focus is Answer Set Programming (ASP, [15]), a declarative programming paradigm that roots in first order logic.

A key difference between declarative and imperative programming is that in the former the focus is explaining the problem to be solved and *what* a solution should look like, while for the latter, programmers write code that specifies *how* the problem is to be solved.

Without going into further details about how ASP works internally, consider the following example for an intuition.

Example 1.1. *Suppose one needs to decide what to wear. The following program narrows down the selection of garments in the wardrobe by checking whether specific pieces are designed to be worn in the current season. For the sake of the example we assume that it*

is summer.

$$
\begin{aligned}
garment(winterjacket) &\leftarrow . && (f_1)\\
garment(jeans) &\leftarrow . && (f_2)\\
garment(tshirt) &\leftarrow . && (f_3)\\
garment(shorts) &\leftarrow . && (f_4)\\
warm(winterjacket) &\leftarrow . && (f_5)\\
light(shorts) &\leftarrow . && (f_6)\\
summer &\leftarrow . && (f_7)\\
inseason(G) &\leftarrow winter, garment(G),\ \text{not } light(G). && (r_1)\\
inseason(G) &\leftarrow summer, garment(G),\ \text{not } warm(G). && (r_2)\\
wear(G) &\leftarrow garment(G), inseason(G). && (r_3)\\
&\leftarrow winter, summer. && (c_1)
\end{aligned}
$$

The program is made up of eleven rules: $f_1 \ldots f_4$ list the four pieces we have in the wardrobe (a winter jacket, a pair of jeans, a t-shirt and shorts). As these rules have no premises, they are also called facts*. Rules f_5 and f_6 qualify garments that are light or warm. The current season is encoded in f_7. Through r_1 and r_2 we define when a garment is "in season", i.e. whether a piece is not light (resp. not warm) which means it should be worn in winter (resp. summer). Finally, we express that a garment can be worn in case it is in season in rule r_3. The last rule c_1 models a* constraint*: It is unreasonable that it is both winter and summer for one evaluation of the program.*

The evaluation of an ASP program is done by an ASP system, which may be comprised of further components. For user and programmer, no detailed knowledge about the system is required. The abstract input/output behaviour of these systems is enough to use them in most cases: Given an input program (and sometimes a few options on which algorithms or data structures to employ) it computes the answer sets. Loosely speaking, answer sets are possible combinations of values that satisfy all requirements that are stated in the program.

Many ASP system implementations use solving algorithms closely related to those found in boolean satisfiability (SAT, [5]) solvers to search for answer sets. Finding an answer set for a logic program, which means exploring truth assignments for atomic expressions under the rules defined in the program, is similar to finding a satisfying assignment for a formula, which means exploring possible truth values for constants under the structure of the formula. Adapting SAT solvers and their methods turned out to be highly effective for solving ASP. To keep up this close relation however, one key difference between formulas in SAT and programs in ASP must be accounted for: formulas in SAT do not have variables, but logic programs do. Variables such as G in the above example must be removed. The process of substituting variables with constants, e.g. substituting G

with one of the four concrete garments, is referred to as *grounding* and the resulting variable-free program is said to be *ground*.

Example 1.2. *Reconsider Example 1.1, naïvely substituting all occurrences of* G *yields eight instances of the rule* r_1 *(two per garment), and four of rule* r_2 *(one per garment).*

$$
\begin{aligned}
garment(winterjacket) &\leftarrow . && (f_1)\\
garment(jeans) &\leftarrow . && (f_2)\\
garment(tshirt) &\leftarrow . && (f_3)\\
garment(shorts) &\leftarrow . && (f_4)\\
warm(winterjacket) &\leftarrow . && (f_5)\\
light(shorts) &\leftarrow . && (f_6)\\
summer &\leftarrow . && (f_7)\\
inseason(winterjacket) &\leftarrow winter, garment(winterjacket),\\
&\qquad \text{not } light(winterjacket). && (r_{1,1})\\
inseason(jeans) &\leftarrow winter, garment(jeans), \text{not } light(jeans). && (r_{1,2})\\
inseason(tshirt) &\leftarrow winter, garment(tshirt), \text{not } light(tshirt). && (r_{1,3})\\
inseason(shorts) &\leftarrow winter, garment(shorts), \text{not } light(shorts). && (r_{1,4})\\
inseason(winterjacket) &\leftarrow summer, garment(winterjacket),\\
&\qquad \text{not } warm(winterjacket). && (r_{1,5})\\
inseason(jeans) &\leftarrow summer, garment(jeans), \text{not } warm(jeans). && (r_{1,6})\\
inseason(tshirt) &\leftarrow summer, garment(tshirt), \text{not } warm(tshirt). && (r_{1,7})\\
inseason(shorts) &\leftarrow summer, garment(shorts), \text{not } warm(shorts). && (r_{1,8})\\
wear(winterjacket) &\leftarrow garment(winterjacket), inseason(winterjacket). && (r_{2,1})\\
wear(jeans) &\leftarrow garment(jeans), inseason(jeans). && (r_{2,2})\\
wear(tshirt) &\leftarrow garment(tshirt), inseason(tshirt). && (r_{2,3})\\
wear(shorts) &\leftarrow garment(shorts), inseason(shorts). && (r_{2,4})\\
&\leftarrow winter, summer. && (c_1)
\end{aligned}
$$

Generally, the ground program may be exponentially bigger than the input program.

Example 1.3 ([21, Example 1])**.** *Consider the following program P which selects at most one element from a domain:*

$$\begin{aligned} dom(1) &\leftarrow . && (f_1)\\ &\vdots\\ dom(n) &\leftarrow . && (f_n)\\ selected(X) &\leftarrow dom(X), \text{not } notSelected(X). && (r_1)\\ notSelected(X) &\leftarrow dom(X), \text{not } selected(X). && (r_2)\\ &\leftarrow selected(X), selected(Y), X \neq Y. && (c_1) \end{aligned}$$

Facts f_1 to f_n span a domain of size n, while rules r_1 and r_2 together achieve that each element from the domain is either selected or not selected, and finally the constraint c_1 ensures that there are no two (or more) selected elements. The corresponding ground program will have $n^2 + 2n$ rules. In other words, the size of the ground program is polynomial in the size of the input program.

Adding another rule to the program will however cause the size of the ground program to grow out of polynomial relation:

$$p(X_1, \ldots, X_k) \leftarrow selected(X_1), \ldots, selected(X_k). \quad (r_3)$$

Now, the size of the ground program of $P \cup \{r_3\}$ will be exponential[1] *in k, precisely $n^k + n^2 + 2n$.*

State-of-the-art ASP systems feature two components: the *grounder* takes care of substituting all variables, i.e. generating the ground program, while the *solver* takes this ground program as input and computes answer sets. Traditionally, the grounder is invoked first, and only after it has output the ground program, the solver starts execution. This two-phased mode of operation is referred to as *ground-and-solve*. With the ground program being exponentially larger than the original input program with variables, these systems are prone to what we call the *grounding bottleneck*: When the ground program is so large that it does not fit into memory, the search for answer sets is impossible in practice. For more examples that exhibit the grounding bottleneck, we refer to [19, Section 1].

1.1 Motivation

The goal of solver components in ASP systems is determining which propositions (such as $wear(winterjacket)$, for example) are *true* and *false* respectively. In the process, an *assignment* designates which propositions are *true* (resp. *false*), it allows to express

[1] In practice for most ASP systems. [7] shows that for programs with bounded predicate arities, polynomial space is sufficient.

statements such as "*wear*(*winterjacket*) is *true*". Once an assignment is found that associates a truth value with every proposition and does not conflict with any rule, an answer set is derived. Assignments conflicting with rules of the program are considered invalid. In Example 1.1, any assignment that allows stating both "*summer* is *true*" and "*winter* is *true*" at the same time, must be avoided as it cannot represent an answer set, because of the constraint in line c_1.

Through the combination of truth values and information about conflicting truth values of propositions, both encoded by the rules of the input program, solvers can infer truth values of other propositions. One method to extend an assignment this way is called *unit propagation*: rules are modeled as sets of literals (propositions and their negations), called *nogoods*, for which not all might satisfy the assignment. From their property that not all corresponding truth values in the assignment may conform with the nogood at the same time, it follows that when all but one elements of the nogood agree with the assignment (we say the nogood is *unit*), the truth value of the remaining proposition can be inferred and added to the assignment.

Example 1.4. *Consider Example 1.1 again, which contains the constraint* c_1*. It can be written as a clause winter* $\wedge$ *summer that must not evaluate to* true*, or simply as a nogood* $\delta = \{\mathbf{T}summer, \mathbf{T}winter\}$ *of the two propositions that cannot conform with the assignment at the same time. An assignment saying that "summer is* true*" in combination with* δ *implies a new truth value, namely that "winter is* false*".*

In order to perform unit propagation, ground-and-solve systems commonly translate the input program into a set of nogoods, and even more advanced algorithms have been devised to dynamically learn new nogoods in the process of searching for answer sets.

For solvers to be efficient, it is crucial to quickly identify which nogoods are unit, even when the assignment under which the nogoods might individually be unit changes frequently during search. Modern solvers implement the so called *Two Watched Literals* (2WL) strategy [25, 36], which describes both an algorithm and a data structure to track nogoods and propagate as soon as they become unit.

An optimization technique to speed up the solving process, pioneered by the ground-and-solve system `dlv` (cf. 2.2.1) is the introduction of a third truth value *must-be-true* in addition to the two usual values *true* and *false*. Using a third truth value also is an effective optimization in lazy-grounding systems (cf. Sections 2.2.2 and 2.2.2). However, 2WL does only account for two truth values, thus cannot be used in conjunction with *must-be-true*.

Goal. The goal of this work is to combine the two improvements (2WL for propagation, which was shown to be successfully in SAT solvers as well as ground-and-solve ASP systems, and *must-be-true* as a third truth value) within one algorithmic framework, such that it can employed in lazy-grounding ASP systems.

Motivation for this is twofold: firstly, enabling lazy-grounding systems to profit from the benefits of 2WL, i.e. improved propagation speeds, while still using *must-be-true*. Secondly, by integrating a procedure very similar to 2WL into a lazy-grounding framework, making it simpler to eventually re-use results that previously were exclusive to ground-and-solve systems in lazy-grounding systems as well.

Contributions. After an analysis of 2WL, algorithms and data structures that account for lazy-grounding are devised and their soundness is shown. We call the new approach Three Watched Literals (3WL). It comes with a description of *watch strctures*, abstract data structures that suggest a memory layout for implementations, as well as a procedural description of the algorithm. It is an extension of 2WL but uses three instead of two watched literals in order to account for the third truth value *must-be-true*. An implementation is contributed to the Alpha system, which in turn is compared against a naïve approach in form of a benchmark. Instances for the benchmark are taken from previous work in the field to maximize reproducibility.

1.2 Structure of the Work

In Chapter 2 we formally introduce logic programs such as the one in the above example (syntax and semantics) of ASP and stable models. Also, a selection of state-of-the-art ASP systems is discussed. The main part of the work is Chapter 3 introducing Three Watched Literals (3WL), an algorithm for efficient propagation for lazy-grounding Answer Set solving based on Two Watched Literals. Evaluation of 3WL as implemented in the Alpha system is evaluated in Chapter 4 and we conclude in Chapter 5.

CHAPTER 2

Preliminaries

This chapter revisits definitions of syntax and semantics of answer set programs. Interpretations and answer sets of such programs are defined.

Apart from these formal foundations, a brief overview of state-of-the-art ASP systems is given and systems that implement lazy-grounding are described.

2.1 Answer Set Programming

In Section 1 we presented an example program with an intuitive description. In this section we formally define syntactic structure and variants of logic programs, and their semantics under ASP. For a thorough introduction to ASP we refer to [8].

2.1.1 Syntax

Given a finite set of constants $\mathcal{C}$, a set of variables $\mathcal{V}$ and a finite set $\mathcal{P}$ of predicate symbols, with $\mathcal{C}$, $\mathcal{V}$, $\mathcal{P}$ pairwise disjoint, we define atoms as the "building blocks" of logic programs.

Definition 2.1. *An* atom *is an expression of the form* $p(t_1, \ldots, t_n)$ *where* $p \in \mathcal{P}$ *is a predicate symbol of arity* $n \geq 0$ *and* $\{t_1, \ldots, t_n\} \subseteq \mathcal{V} \cup \mathcal{C}$ *are terms.*

Note that for atoms of arity zero, parentheses usually are omitted. For simplicity, this definition of atoms does not account for function symbols or function terms. Towards a distinction of ground programs (cf. Example 1.2) from programs with variables, we formally define the class of ground atoms, for which the set of terms coincides with the set of constants.

Definition 2.2. *An atom* $p(t_1, \ldots, t_n)$ *is called* ground *if* $\{t_1, \ldots, t_n\} \subseteq \mathcal{C}$

We structure atoms in the form of rules, which are in turn divided into a head, consisting of at most one atom, and a body of arbitrary size.

Definition 2.3. *A* rule r *is an expression of the form*

$$a \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n.$$

where $a, b_1, \ldots, b_n$ *are atoms,* not *is* negation as failure *(or* default negation*),* $\{a\}$ *is the* head*, denoted* $H(a)$ *and* $\{b_1, \ldots, b_m\}$ *is the* positive body*, denoted* $B^+(r)$*, and* $\{b_{m+1}, \ldots, b_n\}$ *is the* negative body *of* r*, denoted* $B^-(r)$*. Positive and negative body together are simply the* body *of* r *i.e.,* $B(r) = B^+(r) \cup B^-(r)$*. A rule* r *is* ground *if all atoms in* $H(r) \cup B(r)$ *are.*

Note that for a rule r both body and head can be empty: in case $B(r)$ is empty, we say r is a *fact*, and when $H(r)$ is empty, we say that r is a *constraint*. Finally, we arrive at the notion of a logic program.

Definition 2.4. *A* (logic) program P *is a finite set of rules. It is* ground *if all* $r \in P$ *are ground.*

For the scope of this work, the above definition of rules is sufficient. Programs consisting only of rules of this form are called *normal* programs. More general, e.g. rules with disjunctive heads $a_1 \vee \cdots \vee a_k \leftarrow \ldots$ also have interesting properties, but are outside the scope of this thesis.

In practice, many more syntactic constructs are used to ease modelling programs: arithmetic expressions, aggregates allowing statements about sets (counting how many elements match some criteria, summation of numeric values, etc.) and choice rules that encode sets of atoms for which only a given number should be in an answer set, all of which can be translated into simple rules. The core standard, widely accepted by ASP many systems, is available from [2].

2.1.2 Semantics

An atom in the sense of ASP is a proposition with no deeper structure. It might represent some external state of affairs, e.g. *rainy* to indicate whether the weather is not nice or $handsome(yue)$ a statement about Yue being handsome. Their granularity or level of abstraction directly affects how detailed the world is modeled by a program, because atoms are treated as internally consistent statements and they cannot be split further.

In order to state the truth values of atoms in a program, we define an interpretation.

Definition 2.5. *An* interpretation *is a set of ground atoms* I*.*

Considering Example 1.1, note that the set $I_1 = \{wear(G)\}$ is not an interpretation, because the atom it contains is not ground, however $I_2 = \{wear(winterjacket)\}$ is an interpretation, because $winterjacket$ is.

The truth value of an atom is defined by an interpretation I, which in turn is simply the set of atoms that are considered *true*, i.e. we say that "a is *true*" if $a \in I$.

Interpretations and literals are related through satisfaction.

Definition 2.6. *An interpretation I* satisfies *a positive literal $l = p$, if $p \in I$ and a negative literal $l = \text{not } p$ if $p \notin I$. We denote this as $I \models l$.*

We commonly encode the truth value of atoms through satisfaction, e.g. we say that an atom p is *true* under some interpretation I if the the literal $l = p$ is satisfied by I and vice versa. Satisfaction of literals expands to satisfaction of rules.

Definition 2.7. *An interpretation I* satisfies *a ground rule r, denoted $I \models r$ if $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$ implies that $H(r) \subseteq I$, i.e. when the body of r is satisfied, then the head of r is satisfied as well.*

In case an interpretation satisfies all rules of a program, we call it a model thereof.

Definition 2.8. *A* (classical) model *of a ground program P is an interpretation I that is a model of all rules in the program, i.e. $I \models r$ for all $r \in P$.*

We use the Gelfond-Lifschitz reduct towards a definition of answer sets.

Definition 2.9 (see [15, Section 2])**.** *Given a ground program P and an interpretation I, the* Gelfond-Lifschitz reduct *(or just* reduct *in short) of P with respect to I, denoted P^I is defined as follows:*

$$P^I = \Big\{ a \leftarrow b_1, \ldots, b_m \mid a \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \in P \text{ and } \{b_{m+1}, \ldots, b_n\} \cap I = \emptyset \Big\}$$

Finally, an answer set is a $\subset$-minimal model of the corresponding reduct.

Definition 2.10. *An interpretation I is an* answer set *of a ground program P if it is a $\subset$-minimal model of P^I, i.e. there is no $J \subset I$ which is also a model of P^I.*

Note that the answer sets of some logic program P equal the answer sets of the corresponding ground program.

Example 2.1 (cf. [8, Example 16])**.** *Consider the following program P:*

$$\begin{aligned} \mathit{citizen}(\mathit{alice}) &\leftarrow . \\ \mathit{democrat}(C) &\leftarrow \mathit{citizen}(C),\ \text{not } \mathit{democrat}(C). \\ \mathit{republican}(C) &\leftarrow \mathit{citizen}(C),\ \text{not } \mathit{republican}(C). \end{aligned}$$

The corresponding ground program is as follows:

$$citizen(alice) \leftarrow . \tag{f_1}$$
$$democrat(alice) \leftarrow citizen(alice),\ \text{not}\ republican(alice). \tag{r_1}$$
$$republican(alice) \leftarrow citizen(alice),\ \text{not}\ democrat(alice). \tag{r_2}$$

Note that interpretations without $citizen(alice)$ *immediately conflict with* f_1*, i.e.* f_1 *is not satisfied. Conceivable interpretations are:*

$$I_1 = \{citizen(alice), democrat(alice)\}$$
$$I_2 = \{citizen(alice), republican(alice)\}$$
$$I_3 = \{citizen(alice), democrat(alice), republican(alice)\}$$
$$I_4 = \{citizen(alice)\}$$

Considering I_1 *we obtain the reduct* P^{I_1}

$$citizen(alice) \leftarrow .$$
$$democrat(alice) \leftarrow citizen(alice).$$

The negative body of r_1 *is removed because* $democrat(alice) \in I_1$ *and* r_2 *is not in the reduct because* $republican(alice) \notin I_1$*. The least model of* P^{I_1} *coincides with* I_1*, so* I_1 *is an answer set. For the same reason (symmetric argumentation),* I_2 *is an answer set as well.*

Considering I_3 *we see that* P^{I_3} *is just the fact* f_1*, but there are two more elements in* I_3*, so it is not an answer set.*

P^{I_4} *contains the positive part of both rules, i.e.* P^{I_4} *is:*

$$citizen(alice) \leftarrow .$$
$$democrat(alice) \leftarrow citizen(alice).$$
$$republican(alice) \leftarrow citizen(alice).$$

Two atoms that are in the smallest model of P^{I_4}*, inferred via the two rules, are missing from* I_4*:* $\{democrat(alice), republican(alice)\}$

In practice, interpretations cannot be guessed in their entirety but are constructed step-by-step in a search process called *solving*. A distinction between an unassigned atom and an atom that has been found to be *false* is not possible when directly working with interpretations like above; partial interpretations cannot be expressed, as everything that is not explicitly included in the interpretation is effectively assumed to be *false*. This is why interpretations are commonly represented by assignments, which explicitly encode the truth value of atoms.

Definition 2.11. *An* assignment *A is a set of literals. Assignments are consistent, i.e. for any literal $\mathbf{T}a \in A$, its negation is not contained, i.e. $\mathbf{F}a \notin A$.*

Literals can be related with assignments similar to interpretations: $A \models l$ if $l \in A$. We say that an atom p is unassigned if neither $A \models \mathbf{T}p$ nor $A \models \mathbf{F}p$. For the case where there are no unassigned atoms, the interpretation that corresponds to the assignment is the subset of all positive literals in the assignment itself.

2.2 State-of-the-art ASP Systems

In this section we survey state-of-the-art ASP systems in order to highlight similarities and differences in their approaches. As this work is concerned with the adoption of 2WL, a technique common in SAT solvers and ground-and-solve ASP systems, for lazy-grounding systems, we focus on the comparison of ground-and-solve and lazy-grounding systems.

The first ASP systems developed are based on the ground-and-solve approach, as it is less involved and can be implemented in a more straight-forward manner than lazy-grounding. We consider `clasp` as it uses two-watched literals and employs nogood learning and `dlv` because it introduced *must-be-true* as a truth value, which turns out to be very effective in lazy-grounding as well.

Only gradually, running into grounding issues with benchmarks and in industry settings, the grounding bottleneck was recognized, which lead to the development of lazy-grounding systems. With lazy-grounding systems being newer, and less time spent reasoning about their efficiency and engineering them, they often fail to achieve the performance of ground-and-solve systems. The main culprit of early lazy-grounding systems is that they cannot leverage state-of-the-art techniques from ground-and-solve systems: Among others, propagation using two-watched literals and heuristics are where the performance of ground-and-solve systems stems from.

2.2.1 Ground-and-Solve

We describe two very successful systems that follow the ground-and-solve approach.

clingo

`clingo`[1] is a ground-and-solve ASP system, written in C++[2], which combines the grounder `gringo` and the solver `clasp` [12]. `clasp` uses the 2WL strategy for propagation [13, Sec. 5.3] and implements conflict-driven nogood learning (CDNL) [13, Sec. 4.1] in analogy to conflict-driven clause learning (CDCL) in SAT solvers [32, 37]. It is among the fastest ASP systems according to [14][3].

[1] `https://potassco.org/clingo/`

[2] `http://www.open-std.org/jtc1/sc22/wg21/`

[3] See `http://aspcomp2015.dibris.unige.it/aspcomp2015-iclp-slides.pdf` for detailed results

DLV

`dlv` [3] is a versatile ground-and-solve ASP system divided into front-end, intelligent grounding, model generation and model checking components. It comes with front-ends for various applications: (Extended) Disjunctive Logic Programming [24] (similar to DISLOG [31] and DisLoP [1]), Diagnostic Reasoning, and the Structured Query Language[4]). With [10] the authors add two important features: Firstly, they introduce a third truth value, must-be-true, to enhance propagation. It marks atoms that must be assigned true in order to yield an answer set, but where there is no proof found in the search process (see Section 2.2.2 and Example 2.2 as well). Secondly, heuristics that increase the odds of choices to be correct greatly improve overall performance.

A detailed account on the development, optimization and industrial application of `dlv` is available [20].

2.2.2 Lazy-Grounding

In this section we briefly describe three different ASP systems that were designed to circumvent the grounding bottleneck. They all build on top of computations as described in [22].

ASPeRiX

`ASPeRiX` is one of the first ASP systems that were designed to avoid the grounding bottleneck. It was first prototyped in 2008[5] and published in [17, 18], with a detailed explanation in [19] and a C++ implementation. Its core algorithm evolved from the concept of computation as in [22].

In the process of finding a solution, it extends a partial interpretation [19, Def. 4] in form of a pair $\langle IN, OUT\rangle$ of disjoint atom sets, where the atoms in IN belong to the answer set that is currently searched and the atoms in OUT do not. Furthermore, the system tracks the set of ground rules, R, which is lazily extended by grounding the input program. Rules of the program r relate to a partial interpretation and are called *supported* ($B^+(r) \subseteq IN$), *blocked* ($B^-(r) \cap IN \neq \emptyset$), *unblocked* ($B^-(r) \subseteq OUT$), and *applicable* (supported and not blocked) [19, Def. 5].

The main concept in this ASP system is an `ASPeRiX` *computation* [19, Def. 7]: It is a sequence of pairs $\langle R_i, I_i\rangle$ that captures ground rules and a partial interpretation $I_i = \langle IN, OUT\rangle$. The computation starts with $\langle\emptyset, \langle\emptyset, \{\bot\}\rangle\rangle$ and is inductively defined through the rules *propagation* (monotonic; a new rule $r_i \notin R_{i-1}$ can be ground from the program such that it "fires", i.e. its head is added to IN), *choice* (non-monotonic; there are no rules that propagate, but the solver guesses whether an applicable rule fires,

[4] [3] references the 1999 version available from `https://www.iso.org/standard/26196.html` and `https://www.iso.org/standard/26197.html`

[5] According to the project's website `http://www.info.univ-angers.fr/pub/claire/asperix/#download`

i.e. forces or prohibits its instantiation) and *stability* (no rules for propagation or choice left). Through inductive definition of computations, it is guaranteed that the sequence converges to an answer set iff there exists one [19, Thm. 2].

The most important take-away from ASPeRiX for this work, however is the introduction of a third truth value which allows for more efficient convergence of above computations: The truth value must-be-true indicates that an atom that is not in already in IN must be in IN (cannot be in OUT) in order to find an answer set. It allows to mark certain atoms for which a "proof", i.e. a firing rule with the atom as its head, has not yet fired, but must, at some point in the computation.

Example 2.2 ([19, Example 7])**.** *Let* $\leftarrow$ not b. *be a constraint whose body contains only one literal* not b *with* $b \notin IN \cup OUT$. *In order to have an answer set,* b *must be in* IN *so that the constraint is not applicable but* b *is not yet proved (it is not in the head of a fired rule). Thus, one can only conclude that* b *must be true.*

This leads to a reduction of the search space in propagation [19, Ex. 8] and also decreases the size set of non-monotonic candidate rules for choice [19, Ex. 9]. Partial interpretation and computation are modified to consider must-be-true in [19, Def. 8] and [19, Def. 11] accordingly. Again, there is a correspondence to answer sets [19, Thm. 3].

ASPeRiX shows how lazy grounding ASP systems can leverage must-be-true as a third truth value. How unit propagation is affected by must-be-true is a central topic in Chapter 3.

OMiGA

OMiGA[6] [6], written in Java[7], uses computation to explore the search space similar to ASPeRiX. In contrast to that it uses a so called *Rete network* [11] as dominating data structure: Rete is an approach to search large collections of objects for those matching a pattern, and it was initially designed to be used in expert systems and rule based production systems, where there are many rules to consider.

In OMiGA the network is used to: (a) store ground atoms, (b) retrieve ground atoms for the purpose of grounding new rules, (c) find applicable non-ground rules (*propagation* in ASPeRiX), and (d) find rules to guess on (corresponding to *choice* in ASPeRiX). While ASPeRiX re-evaluates applicability of non-ground rules at each step in the computation, OMiGA keeps grounded atoms and partial ground rule interpretations in the network. In choosing a Rete network to store these data, OMiGA trades space for time, as information retrieval from the network is faster than re-computation of applicability.

OMiGA was also extended for learning of non-ground rules in [34], analogous to conflict driven nogood learning as in clasp and clause learning in SAT solvers.

[6] http://www.kr.tuwien.ac.at/research/systems/omiga/
[7] https://oracle.com/java/

GASP

GASP [26, 27][8] is implemented in SICStus Prolog 4[9]. It represents interpretations as finite domains and uses Constraint Logic Programming.

2.3 Two Watched Literals

As briefly mentioned in Section 1.1, Two Watched Literals (also "Two Literal Watching", 2WL, [25, 36]) is an algorithm for unit propagation commonly used in SAT solvers (cf. [16, Section 2.2.2, p. 94]). This section is devoted to a description of the method, first in the context of SAT, then about its application in ASP solving.

In SAT solving, formulas are typically input to solvers in conjunctive normal form (CNF), i.e. conjunctions of disjunctive clauses of classical literals[10]: $\psi = (l_{1,1} \vee \cdots \vee l_{1,k_1}) \wedge \cdots \wedge (l_{n,1} \vee \cdots \vee l_{n,k_n})$. As all disjunctive clauses are connected through conjunction, all of them must evaluate to *true* for the formula to be satisfied. Each disjunctive clause evaluates to *true* if at least one literal does. This property is exploited by SAT solvers to infer assignments. We call a disjunctive clause *unit* if all but one literal in the clause evaluate to *false*, and the last remaining literal has no truth value, i.e. the variable in the literal is unassigned. In this case, we say that the clause *propagates* and the unassigned varaible is assigned s.t. the disjunctive clause evaluates to *true*. Consider [30] for efficient algorithms in SAT solvers.

Example 2.3. *Consider the formula* $\psi = (a_1 \vee \neg a_2 \vee a_3 \vee \neg a_4) \wedge (\neg a_1 \vee \neg a_2 \vee a_3 \vee a_4)$ *It yields two disjunctive clauses,* $\psi = \phi_1 \wedge \phi_2$ *with* $\phi_1 = a_1 \vee \neg a_2 \vee a_3 \vee \neg a_4$ *and* $\phi_2 = \neg a_1 \vee \neg a_2 \vee a_3 \vee a_4$*, both of which have to be satisfied in order for* ψ *to be satisfied. As the solver is evaluating* ψ *under different (partial) assignments for atoms* $a_1, \ldots, a_4$ *it "watches" two literals per clause. This way it can detect whether a clause is unit. Whenever an atom* a *is assigned it considers the clauses where* a *is a watched literal for propagation. Conversely, all other clauses, where* a *is not watched, are guaranteed to not be unit.*

The concept of a nogood originates in constraint programming (CP) (cf. [13, 28, 29]), where the goal is to find an assignment for variables such that given constraints are satisfied. SAT and ASP can be viewed as specializations of CP. For example, in a problem containing the variable x and the constraint $x > 1$ one can forbid the value $x = 1$. The assignment $x = 1$ is a nogood. In this work, nogoods are only used as representations of boolean constraints. Unit propagation is sometimes also referred to as boolean constraint propagation.

[8] `https://users.dimi.uniud.it/~agostino.dovier/GASP/` and `https://users.dimi.uniud.it/~agostino.dovier/CLPASP/`

[9] `https://sicstus.sics.se/`

[10] The DIMACS format is commonly used to encode concrete formulas.

Definition 2.12 (cf. [28]). *A* nogood *is a set* $\{\sigma_1, \ldots, \sigma_n\}$ *of literals that cannot be extended to a solution.*

First proposed by Gebser et al. [12], in the solver components of ASP systems, nogoods take the role of clauses in SAT solvers. Nogoods correspond to partial assignments that cannot be extended to an answer set, and are derived from rules. `clasp` uses a translation procedure based on Clark's completion [4], described in [13, Section 3], which requires knowledge of the full ground program. The Alpha system uses a different scheme defined in [35, Definition 5] and other methods are conceivable.

Example 2.4. *For example, given a rule* r *of the form* $a_1 \leftarrow a_2, a_3, \text{not } a_4$*, the nogood* $\delta = \{\mathbf{F}a_1, \mathbf{T}a_2, \mathbf{T}a_3, \mathbf{F}a_4\}$ *excludes assignments where the body of the rule is satisfied, but its head is not: An interpretation* I *that satisfies all elements of* $B(r) = \{a_2, a_3, \text{not } a_4\}$ *and the negation of the only element of* $H(r)$*, cannot be a model of* r *(Definition 2.7).*

Given a nogood δ and an assignment A with $\delta \subseteq A$, then no interpretation derived from A can be a solution. This property allows for propagation: If $A \cap \delta = \{\sigma\}$, we say that δ is *unit* or unit on σ. In order to obtain a solution we may extend A to A' so that $\overline{\sigma} \in A'$.

The number of nogoods generated from a program is proportional to the size of the corresponding ground program (depending on the translation scheme used). Thus, one can expect many nogoods, possibly millions, to be generated in the process. However, after constructing nogoods from the input program (or parts thereof, as in the lazy-grounding case) and propagating truth values, it is important to efficiently evaluate whether there is a nogood that is unit under the extended assignment. This is where the benefits of 2WL in SAT (over clauses) and in ASP (over nogoods) align.

For each nogood, two distinct and unassigned literals are chosen to be watched. When one of these literals is assigned, a different literal (distinct from the second watched literal) in the nogood is chosen as the new watched literal. This process is continued until there are no two distinct literals available in the nogood. Then the nogood is unit at this point and a new assignment can be generated.

Example 2.5. *Consider Figure 2.1. It shows the nogood* $\delta = \{\mathbf{F}a_1, \mathbf{T}a_2, \mathbf{F}a_3, \mathbf{T}a_4\}$ *(which might be part of the representation of a rule in an ASP program, or the nogood corresponding to* ϕ_1 *from Example 2.3) in the context of propagation using 2WL. The watched literals are marked by arrows.*

Initially, the two watched literals are selected arbitrarily. Transitioning from State 1 (empty assignment) to State 2, the solver will not check whether δ *is unit, because* $\mathbf{T}a_3$ *is not watched. When* a_2 *is assigned to* true*, the solver checks* δ*. As* a_2 *is assigned,* $\mathbf{T}a_2$ *cannot be a watched literal anymore, but* δ *also is not unit, because* a_1 *is still unassigned. The pointer that previously pointed at* $\mathbf{T}a_2$ *is moved to* $\mathbf{F}a_1$*. Note that this is the only option, as the atoms of all other literals are already assigned and* $\mathbf{T}a_4$ *is already being pointed at.*

Processing the assignment of a_1, again, δ is considered. Now, there is no unassigned literal left that is not also watched. The nogood is unit. The truth value of a_4 is inferred to be false. *At this point (State 5), pointer placement is largely irrelevant, because δ is satisfied.*

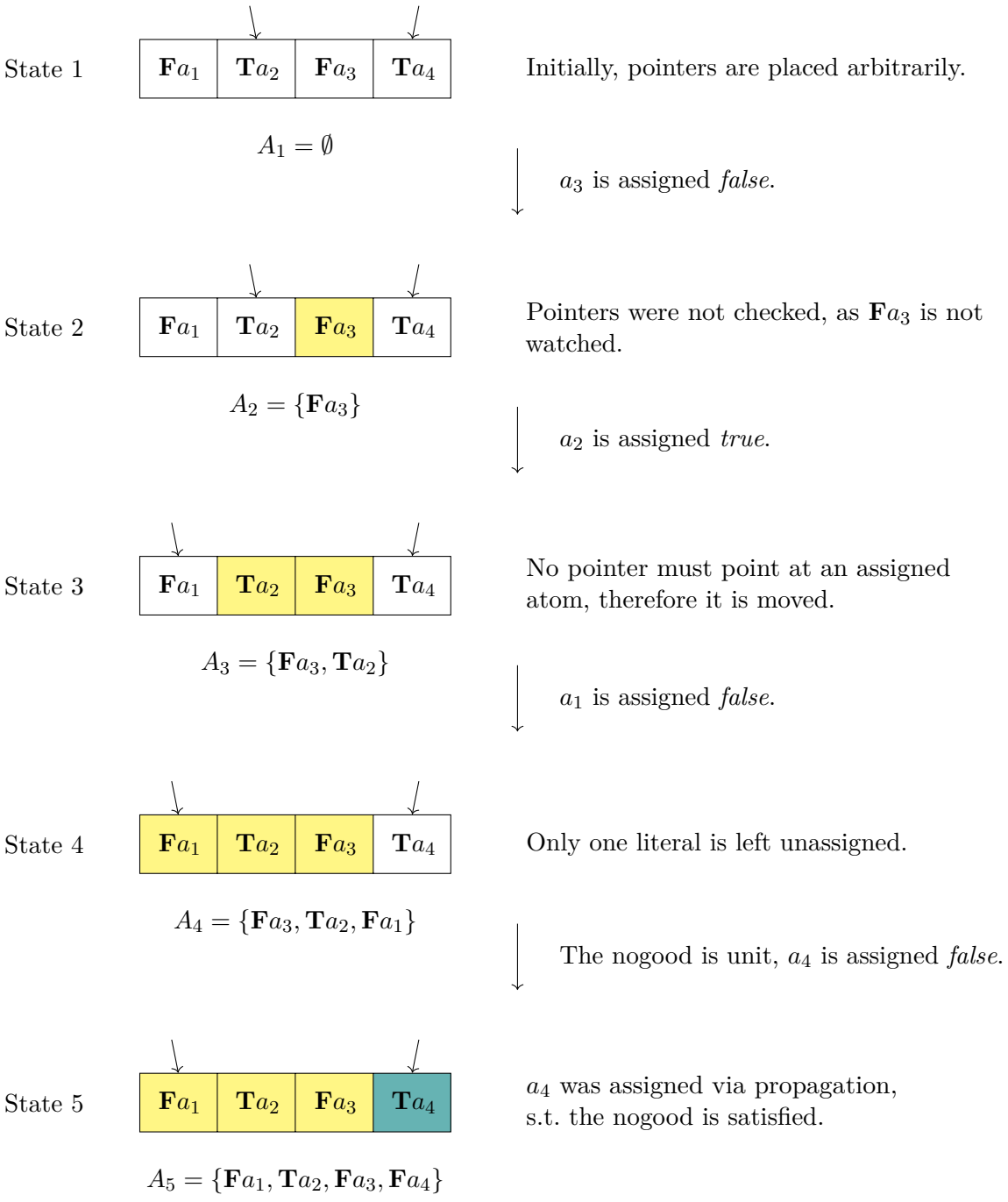

Figure 2.1: Step-by-step visualization of unit propagation with Two Watched Literals.

CHAPTER 3

Three Watched Literals

The central idea of propagation is to infer (*propagate*) new truth values from known truth values and nogoods, derived from the input program, as briefly explained in 1.1.

In this chapter we extend Two Watched Literals (2WL), and present an algorithm that accounts for *must-be-true* as third truth value. We call the new algorithm Three Watched Literals (3WL).

For the description of 3WL we will largely disregard the original structure of the input logic program (both forms: with variables and ground), i.e. its rules. We are only concerned with the solver component of the ASP system which deals with nogoods that in turn represent rules. The task of translating rules to nogoods is left to the grounder component and abstracted away for our purposes. However, we will still refer to a set of nogoods, meaning all nogoods that were generated from the input program by the grounder, and passed to the solver. Note that we do not require that these nogoods represent *all* rules in the ground program, which would defeat the purpose of lazy-grounding.

The propagation algorithm described here will usually be called in alternation with the grounder component (among other procedures), searching for an assignment that represents an answer set. Visualizing which part of the ASP system we are concerned with in this work, we show the architecture of the Alpha system in Figure 3.1. Nogood storage implements what we call *watch structures* in the following sections of the chapter, and the assignment component directly implements assignments as defined in the next section.

Concerning the control flow inside the solver system, we are not concerned with conflict resolution, but propagation only. As described below, the propagation algorithms is designed so that an implementation can stop computation as soon as a conflict is detected, but the resolution part is delegated to other components of the system.

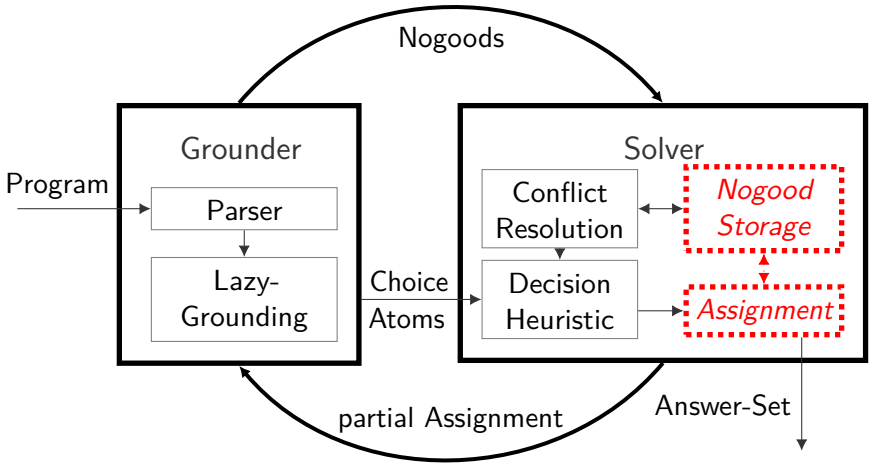

Figure 3.1: Architecture of the Alpha system [21, Figure 1]. Data flow is indicated by arrows. Grounder (left) and CDNL-based solver (right) interact cyclically for lazy-grounding. The components implementing Three Watched Literals (Nogood Storage and Assignment) are highlighted in red italics and have dotted borders.

3.1 Extended Notions for Lazy-Grounding

In this section we amend the definition of literals and nogoods for the purpose of propagation with *must-be-true*. Atoms remain unchanged in their definition, however we will allow literals to not only be *true* and *false* but also *must-be-true*. Also, we will use assignments as sequences (not sets) to exploit the ordering of literals in the assignment for our method.

For the remainder of the work we consider boolean signed literals, which mean atoms and their negations in the classical sense, and, more generally, signed literals, which also allow for a third sign that allows modeling *must-be-true*.

Definition 3.1. *A* boolean signed literal σ *is of the form* $\mathbf{T}v$ *or* $\mathbf{F}v$ *where* v *is an atom and* $\mathbf{T}v$ *expresses that* v *is* true*, and* $\mathbf{F}v$ *that it is* false*.*

Definition 3.2. *A* signed literal μ *is of the form* $\mathbf{T}v$*,* $\mathbf{F}v$*, or* $\mathbf{M}v$ *where* v *is an atom and* $\mathbf{T}v$ *expresses that* v *is* true*,* $\mathbf{F}v$ *that it is* false *and* $\mathbf{M}v$ *that it* must-be-true*.*

With the introduction of literals like $\mathbf{M}v$, we also explicitly define two forms of taking the complement of such literals.

Definition 3.3. *Strong complement, denoted by* $\overline{\mu}^s$*, and weak complement,* $\overline{\mu}^w$*, mapping* $\mathbf{F}v$ *to* $\mathbf{T}v$ *and* $\mathbf{F}v$ *to* $\mathbf{M}v$ *respectively, of a signed literal are defined by the following truth table:*

μ	$\overline{\mu}^s$	$\overline{\mu}^w$
$\mathbf{T}v$	$\mathbf{F}v$	$\mathbf{F}v$
$\mathbf{M}v$	$\mathbf{F}v$	$\mathbf{F}v$
$\mathbf{F}v$	$\mathbf{T}v$	$\mathbf{M}v$

A literal does not only conflict with its strong complement but also with its weak complement.

Definition 3.4. *Two literals μ_1, μ_2 are said to* conflict *in case $\mu_1 = \overline{\mu_2}^w$ or $\overline{\mu_1}^w = \mu_2$.*

Occasionally the notation $\mu = \mathbf{X}v$ will be used to express that μ is a literal of v where $\mathbf{X} \in \{\mathbf{T}, \mathbf{M}, \mathbf{F}\}$, i.e. μ is over the atom v, but the sign is not important.

Also, we will need to find the boolean signed literal for a given signed literal.

Definition 3.5. *The* boolean projection *of a signed literal μ, denoted $\mathcal{B}(\mu)$ translates a signed literal into a boolean signed literal and is defined as follows:*

μ	$\mathcal{B}(\mu)$
$\mathbf{T}v$	$\mathbf{T}v$
$\mathbf{M}v$	$\mathbf{T}v$
$\mathbf{F}v$	$\mathbf{F}v$

We define assignments similar to [13, Section 2, p. 3]. However, here assignments contain not boolean signed literals, but signed literals, and the notation for expressing a partial assignment differs.

Definition 3.6. *An* assignment *A is a sequence $(\mu_1, \ldots, \mu_n)$ of signed literals $\mu_i = \mathbf{X}v_i$ with $1 \leq i \leq n$, or the special sequence* conflict*, indicating a conflict.*

Below, assignments are sometimes also used as sets, in which case the set represented by some assignment is simply the set of all signed literals contained in the sequence.

For an assignment $A = (\mu_1, \ldots, \mu_i, \ldots, \mu_j, \ldots, \mu_n)$, we denote its size as $|A|$ with $|A| = n$ and $|conflict| = 0$, and a partial assignment, which is a sub-sequence constructed from an assignment, as $A[i, j] = (\mu_i, \ldots, \mu_j)$ where $1 \leq i < j \leq n$. Furthermore, we reference a single literal using $A[i] = \mu_i$ for $1 \leq i \leq n$.

The assignment obtained by appending the literal μ to $A = (\mu_1, \ldots, \mu_n)$ is denoted by

$$A \circ \mu = \begin{cases} \mathit{conflict} & \text{if } \{\overline{\mu}^s, \overline{\mu}^w\} \cap A \neq \emptyset \text{ or } A = \mathit{conflict} \\ (\mu_1, \ldots, \mu_n, \mu) & \text{otherwise} \end{cases}$$

For every assignment, we can also construct a corresponding boolean assignment, which is obtained by applying the boolean projection to all contained literals. We overload the function $\mathcal{B}$ for assignments. Boolean assignments are just sets, and not sequences like assignments are, because we do not require any ordering in this work.

Definition 3.7. *A* boolean assignment *is a set of boolean signed literals derived from an assignment, denoted $\mathcal{B}(A)$ and defined as $\mathcal{B}(A) = \{\mathcal{B}(\mu) \in A\}$, where $\mathcal{B}$ applied to literals is the boolean projection.*

Relating assignments with literals and atoms, we say that an atom v is *assigned* under some assignment A if any literal containing it is in A, i.e. if $A \cap \{\mathbf{T}v, \mathbf{M}v, \mathbf{F}v\} \neq \emptyset$. We say that an atom is unassigned under A if it is not assigned under A.

Definition 3.8. *A literal μ_1 is said to conflict with A in case it conflicts with any $\mu_2 \in A$.*

We extend Definition 2.12. For propagation with *must-be-true*, it is important to qualify literals in nogoods that correspond to heads of rules, as the head might only propagate to *true* in case all positive literals in the body are *true* as well (but none of them is *must-be-true*).

Definition 3.9 (cf. [21])**.** *The* head *of a nogood δ, is a single literal $H(\delta)$, for which $H(\delta) \in \delta$. Not every nogood must have a head.*

We need two variants of a nogood being unit. One that accounts for *must-be-true* and one that does not, i.e. it treats $\mathbf{M}v$ the same as $\mathbf{T}v$.

Definition 3.10. *A nogood δ is* strongly unit *under an assignment A if $\delta \setminus A = H(\delta)$. Only nogoods with a head can be strongly unit.*

Note that any nogood that is strongly unit under some assignment is also weakly unit, but not the other way round. Below, nogoods are sometimes referred to be *unit* which amounts to stating that they are weakly unit. Also, when clear from context, reference to a particular assignment is omitted.

Violation of a nogood (signaling that the assignment under question violates a rule) does not distinguish between *true* and *must-be-true*.

Definition 3.11. *A nogood δ is* violated *under an assignment A if $\delta \subseteq \mathcal{B}(A)$.*

Definition 3.12. *A nogood δ is* satisfied *under an assignment A if there is no $A' \supseteq A$ s.t. δ is violated under A'.*

Definition 3.13. *A nogood δ is* weakly unit *under an assignment A if $\delta \setminus \mathcal{B}(A) = \{\sigma\}$*

Example 3.1. *Consider the nogoods $\delta_1 = \{\mathbf{T}a_1, \mathbf{F}a_2\}, \delta_2 = \{\mathbf{T}a_3, \mathbf{F}a_4\}$ with the head of δ_2 being $H(\delta_2) = \mathbf{F}a_4$ and how they relate to the following assignments:*

Assignment	*State of* δ_1	*State of* δ_2
$A_1 = \{\mathbf{T}a_1, \mathbf{T}a_3\}$	*weakly unit*	*strongly unit*
$A_2 = \{\mathbf{M}a_1, \mathbf{M}a_3\}$	*weakly unit*	*weakly unit*
$A_3 = \{\mathbf{F}a_1, \mathbf{F}a_3\}$	*satisfied*	*satisfied*
$A_4 = \{\mathbf{F}a_2, \mathbf{F}a_4\}$	*weakly unit*	*weakly unit*
$A_5 = \{\mathbf{T}a_1, \mathbf{F}a_3\}$	*violated*	
$A_6 = \{\mathbf{M}a_1, \mathbf{F}a_3\}$	*violated*	
$A_7 = \{\mathbf{T}a_3, \mathbf{F}a_4\}$		*violated*
$A_8 = \{\mathbf{M}a_2, \mathbf{T}a_4\}$	*satisfied*	*satisfied*

Note that for an assignment A, any nogood that is satisfied under A will also never be unit, and never propagate, for any $A' \supseteq A$.

Definition 3.14. *A nogood* δ *is* silent *under some assignment* A *if* δ *is not violated by* A *and* δ *is not unit wrt.* A*.*

Note the difference between some δ being silent vs. satisfied under an assignment: With δ being silent under A_1, there might well be some $A'_1 \supset A_1$ under which δ is unit or violated. However δ satisfied under A_2 is stronger and implies that there is no larger assignment $A'_2 \supset A_2$ such that δ is not satisfied under A'_2. It follows that δ can neither be unit nor violated under A'_2.

3.2 Watch Structures

In Section 2.3 we did not detail how a data structure that stores which two literals are watched for each nogood might look like. References to the watched literals are illustrated in Figure 2.1 as pointers, and indeed we are suggesting some form of lookup table: As a new literal μ is added to the assignment, the propagation algorithm will resolve the set of nogoods containing μ as a watched literal. We define an abstract data structure for this lookup.

Definition 3.15. *A* watch structure *is a function that maps an atom* a *to a triple containing sets of nogoods, the so called* watch sets *of* a*:*

$$\Delta(a) = \langle W^+, W^-, W^\alpha \rangle.$$

For some signed literal $\mu = \mathbf{X}a$ *let*

$$\Delta^{\pm}(\mu) = \left\{ \begin{array}{ll} W^+ & \text{if } \mu = \mathbf{T}a \text{ or } \mu = \mathbf{M}a \\ W^- & \text{if } \mu = \mathbf{F}a \end{array} \right\} \text{with } \Delta(a) = \langle W^+, W^-, W^\alpha \rangle.$$

For some signed literal $\mu = \mathbf{X}a$ *or an atom* a *let*

$$\Delta^\alpha(\mu) = \Delta^\alpha(a) = W^\alpha \text{ with } \Delta(a) = \langle W^+, W^-, W^\alpha \rangle.$$

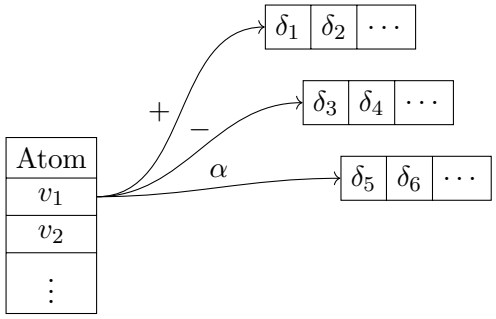

Figure 3.2: Example layout of a watch structure.

When we refer to the "nogoods in Δ", we mean all nogoods in all watch sets of all atoms that Δ is defined for. For the remainder of the work we further assume that all known nogoods, i.e. all those generated by the grounder, are in Δ and only their membership in watch sets, i.e. their watched literals, change.

The functions $\Delta^{\pm}(\mu)$ and $\Delta^{\alpha}(\mu)$ can for example be implemented through table lookups (visualized in Figure 3.2), which take constant time. When a literal $\mathbf{X}a$ is appended to the assignment, all nogoods that might have become unit can be found in the sets $\Delta^{\pm}(\mathbf{X}a)$ and $\Delta^{\alpha}(\mathbf{X}a)$.

A crucial part of this work is the relation between watch structures and assignments: In order to use lookups in watch structures for efficient propagation, we must define which atoms should be considered as watched literals. We first consider single nogoods, and then extend the definition to watch structures as sets of nogoods. Given a nogood δ and an assignment A we call the subsets of literals in δ that might be watched the *candidate sets*. For the case of 2WL, there is just one candidate set, which is the set of unassigned literals. For $\Delta^{\pm}$ we use the same definition. However, with *must-be-true*, we must also consider candidates for Δ^{α}.

Definition 3.16. *The* candidate sets *of a nogood δ under an assignment A are subsets of the nogood and defined as follows:*

$$C_{\pm}(\delta, A) = \{\sigma \in \delta \mid \sigma \notin \mathcal{B}(A)\}$$

$$C_{\alpha}(\delta, A) = \{\mathbf{T}v \in B(\delta) \mid \mathbf{T}v \notin A\}$$

Intuitively, $C_{\alpha}(\delta, A)$ is the set of positive body literals in δ that are not assigned to be *true*. Based on the definition of candidates, we say that a watch structure *watches* an assignment if the watch sets of all nogoods contain candidates so that unit nogoods can be detected by checking the watch sets for newly assigned atoms.

Definition 3.17. *A watch structure* Δ watches *an assignment A if for every nogood $\delta \in \Delta$ holds:*

1. $|C_{\pm}(\delta, A)| \geq 2$ *implies there exist* $\sigma_1, \sigma_2 \in C_{\pm}(\delta, A)$ *such that* $\sigma_1 \neq \sigma_2$ *and* $\delta \in \Delta^{\pm}(\sigma_1)$ *and* $\delta \in \Delta^{\pm}(\sigma_2)$*, and*

2. $C_{\alpha}(\delta, A) \neq \emptyset$ *implies that there is exactly one* $\sigma \in C_{\alpha}(\delta, A)$ *with* $\delta \in \Delta^{\alpha}(\sigma)$.

Given Δ watching A and a new assignment for some atom μ, the nogoods that are candidates for propagation are those in $\Delta^{\pm}(\mu)$ (in case $\mu \notin A$, thus μ changes from *unassigned* to *true*, *must-be-true* or *false*) and $\Delta^{\alpha}(\mu)$ (in case $\mu \in A$, thus μ changes from *must-be-true* to *true*).

Note that with Δ watching A and extending the assignment $A' = A \circ \mu$, now Δ does not trivially watch A'.

Considering a set of nogoods and an assignment, one way to ensure that no nogood will propagate under the assignment is to require all nogoods being silent under the assignment. We combine this condition with a watch structure watching an assignment and use it as the goal state for 3WL.

Definition 3.18. *A watch structure* Δ silently watches *assignment* A *in case all nogoods* $\delta \in \Delta$ *are silent under* A *and* Δ *watches* A*.*

3.3 Unit Propagation

In this section we detail Three Watched Literals (3WL). The process is split into four algorithms, varying in level of abstraction:

Algorithm 3.1 is the most high-level algorithm. Given a new literal to append to an assignment, it first appends the literal and then applies Algorithm 3.2 repeatedly, until no new assignments can be inferred. The input watch structure must be silently watching the assignment.

Algorithm 3.2 is concerned with processing a single assignment. Given an index in the assignment it calls 3.3, and in case the literal is of the form $\mathbf{T}a$ also calls 3.4.

Algorithm 3.3 handles propagation or chooses a new watched literal for nogoods that are weakly unit, based on $\Delta^{\pm}$.

Algorithm 3.4 handles propagation or chooses a new watched literal for nogoods that are strongly unit, based on Δ^{α}.

The solver component initially and after each decision expands its assignment by inferring new literals through unit propagation. We define Algorithm 3.1 “exhaustively”, i.e. in such a way that it will only terminate when no new assignments can be inferred through unit propagation or a conflict was reached. It takes the current assignment A and a watch structure Δ that silently watches A, and the next literal to be appended to the

Algorithm 3.1: UnitPropagateAll(A, Δ, μ)

Input: An assignment A, a watch structure Δ, a signed literal μ.
Output: The assignment A, extended by means of unit propagation with literal $A[j]$ and Δ with updated watches.

```
1 ⟨A_1, Δ_1⟩ ← ⟨A ∘ μ, Δ⟩
2 i ← 1
3 while |A_i| ≥ |A| + i do
4 |   ⟨A_{i+1}, Δ_{i+1}⟩ ← UnitPropagate(A_i, Δ_i, |A| + i)
5 |   i ← i + 1
6 end
7 return ⟨A_i, Δ_i⟩
```

Algorithm 3.2: UnitPropagate(A, Δ, j)

Input: An assignment A, a watch structure Δ, an index j.
Output: The assignment A, extended by means of unit propagation and Δ with updated watches.

```
1 μ ← A[j]
2 ⟨A', Δ'⟩ ← UnitPropagateWeakly(A, Δ, μ)
3 if μ is of the form Ta then
4 |   ⟨A'', Δ''⟩ ← UnitPropagateStrongly(A', Δ', μ)
5 |   return ⟨A'', Δ''⟩
6 end
7 return ⟨A', Δ'⟩
```

assignment μ. In the first line of the algorithm, A_1 is generated by appending μ to A. However, Δ, the input watch structure, equals Δ_1. So while Δ silently watches A, this might not be the case for Δ_1 and A_1, because A_1 contains one more literal that might lead to some nogood in Δ_1 being unit. This triggers a domino effect of unit propagation.

The loop in lines 3 to 6 will then restore Δ_{i+1} to silently watch $A_{i+1}[1, |A| + 1]$. At the point where all all nogoods are silently watched again, the loop terminates because $|A_i| < |A| + i$. In case a conflict is reached in any iteration, the loop will break, because we carefully defined $|conflict| = 0$ and i is at least one.

Before new assignments are resolved to the nogoods that must be checked, Algorithm 3.2 is invoked. It serves the purpose to decide which propagation algorithm(s) should be called. Algorithm 3.3 is always called, but it is sufficient to Algorithm 3.4 only if an atom was assigned to be *true*, i.e. μ is of the form $\mathbf{T}a$. This is because the relevant candidate set $C_{\pm}(\delta, A)$ changes only as atoms are assigned to *true*.

Algorithm 3.3 shows how unit propagation is used to infer assignments from watched nogoods. Its input is the current assignment A, a watch structure Δ and the literal μ that has been newly assigned and should be processed. Tracing the invocation of the

Algorithm 3.3: UnitPropagateWeakly(A, Δ, μ)

Input: An assignment A, a set of watched nogoods Δ, and a literal $\mu \in A$.
Output: A extended by means of nogood propagation and Δ with updated watches.

```
1  foreach δ ∈ Δ±(μ) do
2      if δ is violated then
3          return ⟨conflict, Δ⟩
4      else if δ is strongly unit then
5          A ← A ∘ complement(H(δ))^s
6      else if δ is weakly unit with σ unassigned then
7          A ← A ∘ complement(σ)^w
8      else
9          foreach σ ∈ δ do
10             Δ±(σ) ← Δ±(σ) \ {δ}
11         end
12         Let σ1, σ2 ∈ C±(δ, A) be arbitrary
13         Δ±(σ1) ← Δ±(σ1) ∪ {δ}
14         Δ±(σ2) ← Δ±(σ2) ∪ {δ}
15     end
16 end
17 return ⟨A, Δ⟩
```

algorithm we notice that μ in Algorithm 3.3 corresponds with μ in Algorithm 3.2 and $A_i[|A| + i]$ in Algorithm 3.1.

The set of nogoods that are touched by the algorithm is $\Delta^{\pm}(\mu)$, i.e. all nogoods δ where μ is one of the watched literals. In lines 2 to 7 all cases that require no change of watched literals are handled:

- δ is violated (lines 2-3) and all propagation can be stopped because a conflict was reached.
- δ is strongly unit (lines 4-5) and a new assignment for the head literal of δ is generated by taking its strong complement.
- δ is weakly unit (lines 6-7) and a new assignment for some literal is generated based on the weak complement.

Note that these three cases are not mutually exclusive, e.g. a nogood might be strongly unit and weakly unit under the same assignment. In this case, the algorithm will run into the branch in lines (4-5), so that propagation to *true* overrides propagation to *must-be-true*. Without checking for δ being strongly unit, cases where assignments to

false lead to δ being strongly unit would be overlooked, as Algorithm 3.4 is only called for assignments to *true*.

With the above cases handled, there must be at least two literals in the candidate set $C_{\pm}(\delta, A)$, but μ is assigned, so a new watched literal has to be found for δ to be maintained. In lines 9 to 11 all watches are removed, or rather δ is removed from the watch sets of all its watched literals. Then, in lines 12 to 14, δ is again inserted into the watch sets for two arbitrary candidates.

Algorithm 3.4: UnitPropagateStrongly(A, Δ, μ)

Input: An assignment A, a watch structure Δ, and a literal $\mu \in A$.
Output: A extended by means of nogood propagation.

```
1  foreach δ ∈ Δ^α(μ) do
2      if δ is strongly unit then
3          A ← A ∘ complement(H(δ))^s
4      else if C_α(δ, A) ≠ ∅ then
5          Δ^α(μ) ← Δ^α(μ) \ {δ}
6          Let σ ∈ C_α(δ, A) be arbitrary
7          Δ^α(σ) ← Δ^α(σ) ∪ {δ}
8      end
9  end
10 return ⟨A, Δ⟩
```

Algorithm 3.4 works in a similar way but without the need to check for violations/conflicts and nogoods being weakly unit, as these cases are handled by Algorithm 3.3. Further, by the definition of a nogood δ being strongly unit, we know that all positive literals (except the head literal) of δ must be assigned to *true*, and not only to *must-be-true* for propagation.

Theorem 3.1. *Algorithm 3.1 is sound, i.e. given an assignment A, a watch structure Δ, a literal μ, and the following precondition* P*, it results in a new assignment A' and a new watch structure Δ' such that the postconditions* P'_1 *through* P'_3 *all hold:*

P Δ *silently watches* A.

P'_1 *If μ does not conflict with A and there are assignments that can be inferred by means of unit propagation without a conflict, then Δ' silently watches A'.*

P'_2 *If μ does not conflict with A but a literal inferred from Δ and $A \circ \mu$ conflicts with A, then $A' =$* conflict.

P'_3 *If μ conflicts with A, then $A' =$* conflict *and $\Delta' = \Delta$.*

Sketch of Proof 3.1. *Before establishing* P'_1 *and* P'_2 *observe the loop in lines 3-6 of Algorithm 3.1 closely:*

Initialization *Construct* A_1 *by appending* μ *to* A *(line 1). Initialize some loop counter* $i = 1$ *(line 2). It points at the next literal on which propagation should be performed, i.e. initially* $\mu = A_1[|A| + 1]$.

Iteration *Let the pair* $S_{i+1} = \langle A_{i+1}, \Delta_{i+1}\rangle = \textsc{UnitPropagate}(A_i, \Delta_i, |A| + i)$ *(line 4) denote the output of the* i*-th iteration which performs propagation on assignment* A_i*, watch structure* Δ_i *and literal* $A_i[|A| + i]$ *(line 1). Conversely* S_i *can be interpreted as the input for the* $(i + 1)$*-th iteration. Analysis of the difference between* Δ_i *and* Δ_{i+1} *is what resembles soundness below.*

Termination *Because the number of literals in all nogoods* $\delta \in \Delta$ *is finite, for some* $i = k$ *the loop invariant (line 3) is violated, i.e.* $|A_k| < |A|+k$*. Intuitively, this is the case if no new assignment can be inferred (*$S_{k-1} = \langle A_{k-1}, \Delta_{k-1}\rangle$ *and* $S_k = \langle A_k, \Delta_k\rangle$ *with* $A_{k-1} = A_k$*) or a conflict is reached (*$A_k = \mathrm{conflict}$*). When the loop terminates,* $\langle A', \Delta'\rangle = S_k$ *is returned (line 7).*

We consider two subsequences of the assignment A_n*:* A_n^L *is the "left" subsequence, which contains all assignments that already are processed and will not cause propagation. Conversely,* A_n^R *is the "right" subsequence, containing all literals that might still cause unit propagation. As expected, the two subsequences add up to the full sequence* $A_n = A_n^L \cup A_n^R$.

Towards showing P'_1 *and* P'_2 *from the preconditions, assume that* μ *does not conflict with* A *and let* $\mathcal{P}(n) =$ *(I)* $\vee$ *(II) where*

(I) Let $A_n^L = A_n[1, |A| + n - 1], A_n^R = A_n[|A| + n, |A_n|]$.

- *(a) The structure* Δ_n *watches* A_n^L *, and*
- *(b) for each nogood* δ *in* Δ_n*:*
  - *(i) If* δ *is weakly unit under* A_n*, then* $\exists \mu \in A_n^R : \delta \in \Delta_n^{\pm}(\mu)$.
  - *(ii) If* δ *is strongly unit under* A_n*, then* $\exists \mathbf{T}v_1 \in A_n^R : \delta \in \Delta_n^{\alpha}(v_1)$ *or* $\exists \mathbf{F}v_2 \in A_n^R : \delta \in \Delta_n^{\pm}(\mathbf{F}v_2)$.
  - *(iii) If* δ *is violated under* A_n*, then* $\exists \mu \in A_n^R : \delta \in \Delta_n^{\pm}(\mu)$.

(II) $A_n = \mathrm{conflict}$

Overview and Intuition. *Using* $\mathcal{P}(n)$ *above, we show that* P'_1 *and* P'_2 *follow from the preconditions by induction, closely following the structure of Algorithm 3.1 as outlined above. Initially (*$n = 0$, $A = A_0 = A_L^0$ *and* $A_R^0 = \emptyset$*), we require the input nogoods* Δ *to all be silent through* Pre *as a "stable" starting point. While new literals will be appended to the assignment on the "right" side, we will show that at termination for some* $n = k$ *as hinted above,* $A_k^R = \emptyset$ *again. From there we establish the postconditions. For the*

induction step deriving $\mathcal{P}(i+1)$ from $\mathcal{P}(i)$ we will "move" exactly one literal μ_i from right to left ($\mu_i \notin A_i^L$, $\mu_i \in A_i^R$ and $\mu_i \in A_{i+1}^L$, $\mu_i \notin A_{i+1}^R$ as well as $A_{i+1}^L = A_i^L \cup \{\mu_i\}$). Here, properties (I)(b) are crucial, as they establish the connection between A_i^R and A_i^L and cover the propagation scenarios for μ_i. Note that the "right" subsequence "grows" (with the exception of μ_i), i.e. $A_i^R \setminus \{\mu_i\} \subseteq A_{i+1}^R$ because of new literals inferred by propagation.

Base Case. *Concerning the input assignment $A_0 = A$ and $\Delta_0 = \Delta$, $\mathcal{P}(0)$ directly follows, because (I)(a) is given by* Pre*, and all nogoods being silent (also given by* Pre*) implies that the antecedents of (I)(b)(i), (I)(b)(ii), (I)(b)(iii) are false.*

Induction Hypothesis. *$\mathcal{P}(i)$ holds for some i.*

Induction Step. *For two consecutive iteration steps of the loop $S_i = \langle A_i, \Delta_i \rangle$ and $S_{i+1} = \langle A_{i+1}, \Delta_{i+1} \rangle$, assume $\mathcal{P}(i)$. Then show $\mathcal{P}(i) \rightarrow \mathcal{P}(i+1)$ as follows:*

1. *To show (I)(a) let $\delta \in \Delta$: If $\mathcal{B}(\mu_i) \notin \delta$, then δ stays silent and watched. For $\mathcal{B}(\mu_i) \in \delta$, consider the following two cases:*

   a) *Assume $\mu_i \neq \mathbf{T}v$. Then Algorithm 3.3 is invoked and processes all nogoods $\delta \in \Delta^{\pm}(\mu)$. For any δ, if $|C_{\pm}(\delta, A)| \geq 2$, then δ is inserted into the watch sets for exactly two candidates in lines 11-14.*

   b) *Assume $\mu_i = \mathbf{T}v$. Then Algorithm 3.3 is invoked and behaves like for the case where $\mu_i \neq \mathbf{T}v$, i.e. if $C_{\pm}(\delta, A) \geq 2$ then δ is watched. Additionally, Algorithm 3.4 is invoked and processes all nogoods $\delta \in \Delta^{\alpha}(\mu)$. For any δ, if $|C_{\alpha}(\delta, A)| \neq \emptyset$, then δ is inserted into the watch set of exactly one candidate in lines 4-7.*

   If $\mu_i \in \Delta$ then the premises of the "watch" property regarding candidate sets are false, or pointers were moved such that μ_i does not watch δ anymore.

2. *To show (I)(b)(i), let $\delta \in \Delta$ and assume δ is weakly unit under A_{i+1} (else, (I)(b)(i) trivially holds):*

   a) *Assume δ is weakly unit under A_i.*

   Then $\exists \mu \in A_i^R : \delta \in \Delta_i^{\pm}(\mu)$ by the induction hypothesis.

   Assume towards contradiction that $\mu = \mu_i$. As $\delta \in \Delta_i^{\pm}(\mu)$, the weak complement of μ is appended to the assignment (Algorithm 3.3, line 7), thus $\overline{\mu}^w \in A_{i+1}$. Then δ is not weakly unit under A_{i+1} contradicting our assumption.

   Therefore, $\mu \neq \mu_i$. From $A_i^R \setminus \{\mu_i\} \subseteq A_{i+1}^R$, it follows that $\mu \in A_{i+1}^R$. Further, we see that $\delta \in \Delta_i^{\pm}(\mu)$ implies $\delta \in \Delta_{i+1}^{\pm}(\mu)$ because of the behavior of Algorithm 3.3 in case $\mu \neq \mu_i$ and δ being weakly unit: The set of nogoods that are considered for processing is indicated in line 1 and consists of exactly those

nogoods that are element of $\Delta_i^{\pm}(\mu)$. From $\mu \neq \mu_i$ we know that the algorithm will not consider δ and specifically not add or remove it from any watch set $\Delta^{\pm}$. Thus we have $\exists \mu \in A_{i+1}^R : \delta \in \Delta_{i+1}^{\pm}(\mu)$, i.e. (I)(b)(i) holds.

b) Assume δ is not weakly unit under A_i.

Then $C_{\pm}(\delta, A_i) \geq 2$ and because Δ_i watches A_i (ind. hyp.), we have $\exists \sigma_1, \sigma_2 \in \delta : \sigma_1 \neq \sigma_2 \wedge \delta \in \Delta_i^{\pm}(\sigma_1) \wedge \delta \in \Delta_i^{\pm}(\sigma_2)$. We distinguish on whether μ_i coincides with one of the watched literals:

i. Case $\mu_i \neq \sigma_1 \wedge \mu_i \neq \sigma_2$.
$\Delta_i^{\pm}(\sigma_1) = \Delta_{i+1}^{\pm}(\sigma_1)$ and $\Delta_i^{\pm}(\sigma_2) = \Delta_{i+1}^{\pm}(\sigma_2)$, as watch sets are only modified in case $\sigma_1 = \mu_i$ and respectively $\sigma_2 = \mu_i$ (see Algorithm 3.3: Only nogoods that are being iterated over in line 1 are being removed from any watch sets in line 10). Both σ_1 and σ_2 are in A_{i+1}^R, because $A_i^R \setminus \{\mu_i\} \subseteq A_{i+1}^R$, thus we have (I)(b)(i).

ii. Case either $\mu_i = \sigma_1$ or $\mu_i = \sigma_2$.
This means, processing of μ_i makes δ become unit but while δ is processed by Algorithm 3.3 it is not unit yet, i.e., another nogood processed after δ is unit and the assignment done there leads to δ being unit. Without loss of generality (σ_1 and σ_2 arbitrary in $C_{\pm}(\delta, A_i)$), let $\mu_i = \sigma_1$. Therefore, $\sigma_1 \notin A_{i+1}^R$. While processing δ, in Algorithm 3.3 the branch in line 8 is taken (otherwise δ cannot be unit under A_{i+1}). All candidates $C_{\pm}(\delta, A_i)$ except σ_1, and therefore any literals σ_1', σ_2' (arbitrarily) chosen in the algorithm, are either unassigned under A_{i+1} or in A_{i+1}^R. As δ is unit under A_{i+1}, exactly one of these must be unassigned. Assume (w.l.o.g., symmetric) σ_1' unassigned under A_{i+1} and $\sigma_2' \in A_{i+1}^R$. Then (I)(b)(i) holds for σ_2'.

iii. Case $\sigma_1 = \mu_i \wedge \sigma_2 = \mu_i$ is impossible because $\sigma_1 \neq \sigma_2$.

3. To show (I)(b)(ii), let $\delta \in \Delta$ and assume δ is strongly unit (and therefore also weakly unit, i.e. $H(\delta) = \mathbf{T}v$ and $\mathbf{M}v \in A_{i+1}$) under A_{i+1} (else, (I)(b)(ii) trivially holds): we distinguish whether $\mathcal{B}(\mu_i) \in \delta$, i.e. whether the literal being processed is a watch for δ.

a) Case δ is strongly unit under A_i.

i. Case $\mathcal{B}(\mu_i) \in \delta$, then Algorithm 3.3 (line 5) assigns $H(\delta)$, i.e. $\overline{H(\delta)}^s \in A_{i+1}^R$, and therefore δ cannot be strongly unit under A_{i+1} which contradicts the assumption that δ is weakly unit under A_{i+1}.

ii. Case $\mathcal{B}(\mu_i) \notin \delta$. Because δ is strongly unit under A_i and by the induction hypothesis we have $\exists \mathbf{T}v_1 \in A_i^R : \delta \in \Delta_i^{\alpha}(v_1)$ or $\exists \mathbf{F}v_2 \in A_i^R : \delta \in \Delta_i^{\pm}(\mathbf{F}v_2)$. From $\mathcal{B}(\mu_i) \notin \delta$ it follows that $\mu_i \neq \mathbf{T}v_1 \wedge \mu_i \neq \mathbf{F}v_2$.
From $A_i^R \setminus \{\mu_i\} \subseteq A_{i+1}^R$, it follows that if $\mathbf{T}v_1, \mathbf{F}v_2$ exist, they are in A_{i+1}^R. Also, $\delta \in \Delta_i^{\alpha}(v_1)$ implies $\delta \in \Delta_{i+1}^{\alpha}(v_1)$ and $\delta \in \Delta_i^{\pm}(\mathbf{F}v_2)$ implies $\delta \in \Delta_{i+1}^{\pm}(\mathbf{F}v_2)$ because δ is strongly unit and $\mathcal{B}(\mu_i) \notin \delta$, i.e. no algorithm will

change the watches that contain δ. So we have $\exists \mathbf{T}v_1 \in A^R_{i+1} : \delta \in \Delta^\alpha_{i+1}(v_1)$ or $\exists \mathbf{F}v_2 \in A^R_{i+1} : \delta \in \Delta^\pm_{i+1}(\mathbf{F}v_2)$, i.e. (I)(b)(ii).

b) Case δ is not strongly unit under A_i. Then, $\mathcal{B}(A_i) \setminus \delta \neq \emptyset$.

i. Case $\mathcal{B}(\mu_i) \notin \mathcal{B}(A_i)$, i.e., none of the literals of δ is processed in this step. Since δ is strongly unit under A_{i+1} and not strongly unit under A_i it holds that $\delta \setminus A^R_{i+1} \neq \emptyset$.
Since δ is strongly unit under A_{i+1}, it is also weakly unit under A_{i+1}. Therefore there exists $\mu' \in A^R_{i+1} : \delta \in \Delta^\pm_{i+1}(\mu')$.

- *If $\mu' = \mathbf{F}v'$, then $\exists \mathbf{F}v' \in A^R_{i+1} : \delta \in \Delta^\pm_{i+1}(\mathbf{F}v')$ and (I)(b)(ii) holds.*
- *If $\mu' = \mathbf{T}v'$, then $\mathbf{T}v' \notin A^L_{i+1}$ and hence $\mathbf{T}v' \notin A_i$, which implies that $C_\alpha(\delta, A_i) \supset \{\mathbf{T}v'\} \neq \emptyset$. By induction hypothesis it therefore follows that there exists $\sigma \in C_\alpha(\delta, A_i) : \delta \in \Delta^\alpha(\sigma)$. By $C_\alpha(\delta, A_i)$ only containing literals of form $\mathbf{T}v \in \delta$ it follows that $\exists \mathbf{T}v \in A^R_i : \delta \in \Delta^\alpha_i(v)$. Since none of the literals of δ is processed, it follows that $\delta \in \Delta^\alpha_{i+1}(v)$. Consequently (I)(b)(ii) holds.*
- *If $\mu' = \mathbf{M}v'$, then it follows that $\mathbf{T}v' \notin A^L_{i+1}$, because $\mathbf{M}v'$ can only be added to an assignment in Algorithm 3.3 by line 7, which is only executed if v' is not assigned false, i.e. $\mathbf{F}v'$. From $\mathbf{T}v' \notin A^L_{i+1}$ the same reasoning as in the previous case applies and it follows that (I)(b)(ii) holds.*

ii. $\mathcal{B}(\mu_i) \in \delta \setminus \mathcal{B}(A_i)$, i.e. some literal in δ is processed. If δ is strongly unit while δ is processed by Algorithm 3.3, then $\overline{H(\delta)}^s \in A_{i+1}$ due to line 5 of Algorithm 3.3, which contradicts δ being strongly unit under A_{i+1}. Therefore δ is not yet strongly unit when Algorithm 3.3 processes δ.

A. Case δ is weakly unit with μ_i unassigned: since δ is strongly unit under A_i it must be the case that δ is weakly unit on $H(\delta)$, i.e., σ of line 7 in Algorithm 3.3 is such that $\sigma = H(\delta)$. From δ not being strongly unit under A_i then follows that some $\mathbf{T}v' \in \delta$ is assigned must-be-true*, i.e. $\mathbf{M}v' \in A^L_i$.*
Let $M = \{\mathbf{T}v \in \delta \setminus H(\delta) \mid \mathbf{M}v \in A_i\}$ be the set of all such must-be-true *assigned literals of δ except its head. Let $\mathbf{T}v'$ in the following be the literal of M that is assigned last under A_{i+1}. Intuitively, $\mathbf{T}v'$ triggers δ to be unit.*
Since $\mathbf{T}v'$ is the "last" of M that is assigned, it cannot be the case that $\mathbf{T}v' = \mu_i$, as otherwise δ would be strongly unit while processing μ_i.
We distinguish whether $\mu_i = \mathbf{T}w$ for one atom w.

- *If $\mu_i = \mathbf{T}w$, then Algorithm 3.2 is executed and calls Algorithm 3.4. Since δ is not strongly unit under A_i and $C_\alpha(\delta, A_i) \neq \emptyset$ because $\{\mathbf{T}v'\} \in C_\alpha(\delta, A_i)$ lines 4-7 are executed and it holds that $\exists \mathbf{T}v_1 \in A^R_{i+1} : \delta \in \Delta^\alpha_{i+1}(v_1)$, i.e. (I)(b)(ii) holds.*

- *If $\mu_i \neq \mathbf{T}w$, then Algorithm 3.4 is not executed and from $\mathbf{T}v' \notin A^L_{i+1}$ it follows that $\mathbf{T}v' \notin A_i$ and $C_\alpha(\delta, A_i) \neq \emptyset$, hence by induction hypothesis $\exists\sigma \in C_\alpha(\delta, A_i) : \delta \in \Delta^\alpha_i(\sigma)$ with $\sigma = \mathbf{T}v_1$ for some atom v_1. Since Algorithm 3.4 is not executed, $\delta \in \Delta^\alpha_{i+1}(\mathbf{T}v_1)$ and $\mathbf{T}v_1 \in A^R_{i+1}$, i.e. (I)(b)(ii) holds.*

B. Case δ is not weakly unit with μ_i unassigned, then in Algorithm 3.3 line 11-14 are executed and there exist σ_1, σ_2 with $\sigma_1, \sigma_2 \notin A_i : \delta \in \Delta^\pm(\sigma_1)$ and $\delta \in \Delta^\pm(\sigma_2)$ to establish that δ is watched. Similar to the reasoning in 2.b.ii, we see that the watches condition ensures that the assignment that leads to δ being strongly unit will be processed.

4. To show (I)(b)(iii), let $\delta \in \Delta$ and assume δ is violated under A_{i+1} (else (I)(b)(iii) trivially holds).

a) Assume δ is violated under A_i.
Then $\exists\mu \in A^R_i : \delta \in \Delta^\pm_i(\mu)$ via induction hypothesis.

i. Case $\mu = \mu_i$.
Then, according to Algorithm 3.3 (line 3) $A_{i+1} = \text{conflict}$, so (II) holds.

ii. Case $\mu \neq \mu_i$.
Then, from $A^R_i \setminus \{\mu_i\} \subseteq A^R_{i+1}$, it follows that $\mu \in A^R_{i+1}$, thus $\exists\mu \in A^R_{i+1} : \delta \in \Delta^\pm_{i+1}(\mu)$, because $\delta \in \Delta^\pm_i(\mu)$ implies $\delta \in \Delta^\pm_{i+1}(\mu)$, as watch sets are only modified for nogoods that are watched by μ_i. So we have $\exists\mu \in A^R_i : \delta \in \Delta^\pm_{i+1}(\mu)$, i.e. (I)(b)(iii) holds.

b) Assume δ is not violated under A_i. Then $|\mathcal{B}(A_i) \setminus \delta| \geq 1$ holds by Definition 3.11.

i. Assume $|\delta \setminus \mathcal{B}(A_i)| \geq 2$. This means that at least on literal in δ is unassigned. Then because Δ_i watches A_i it follows that $\exists\sigma_1, \sigma_2 : \delta \in \Delta^\pm_i(\sigma_1), \Delta^\pm_i(\sigma_2)$. Further $\sigma_1 \neq \mu_i$ or $\sigma_2 \neq \mu_i$, so w.l.o.g. $\sigma_1 \neq \mu_i$, so $\exists\ \mathbf{X}v \in X : \delta \in \Delta^\pm_{i+1}(v)$ with a reasoning similar to 2.b.ii.

ii. Assume $|\delta \setminus \mathcal{B}(A_i)| = 1$. Then δ is weakly unit under A_i, and with reasoning similar to 2.a one can show that $\exists\mu \in A^R_{i+1}$ with $\delta \in \Delta^\pm_{i+1}(\mu)$.

Above induction step serves as an explanation on how execution of the loop affects the assignment and especially the watch structure. By using induction, $\mathcal{P}(i)$ was shown for any $i \geq 0$, and therefore for any iteration of the loop in lines 3-6 as well. Specifically $\mathcal{P}(i)$ holds for the last iteration $i = k$ represented by $S_k = \langle A_k, \Delta_k \rangle$.

For cases without conflict and the loop terminating at S_k, the outcome that Δ' watches A' directly follows from (I)(a) as $A' = A_k$ and $\Delta' = \Delta_k$. From termination follows $A^R_k = \emptyset$, i.e. there are no nogoods $\delta \in \Delta'$ that are weakly unit, strongly unit or violated under A', thus all δ are silent under A'. Together, this amounts to $\mathrm{P'}_1$.

Cases in the induction step leading to (II) map to $\mathrm{P'}_2$.

To show $\mathrm{P'}_3$ *assume* μ *conflicts with* A*. Then, on line 1 of Algorithm 3.1, the assignment* A_1 *will be assigned* conflict*. Consequently, the loop in lines 3-6 is not executed as* $|\text{conflict}| = 0$ *and* i *must be at least 1, as it is assigned 1 on line 2.* $\langle \text{conflict}, \Delta \rangle$ *is returned.*

CHAPTER 4

Evaluation

To support our results of performance improvement with 3WL compared to naive propagation, and in order to showcase the Alpha system as the first ASP system to combine 3WL with lazy grounding, we evaluated the run time performance for the publicly available[1] set of problem instances used in [21].

Setup. The benchmarks were run with Alpha `v0.2.0`[2] on a machine with an Intel® Core™ i7-7500U CPU @ 2.7GHz, 16GB main memory, Linux kernel 4.12.5 and Oracle Java™ SE Runtime Environment (build `1.8.0_144-b01`).

Method. All measured timings are times until the first 10 (or all, if less than 10) answer sets were found, and the solver used a fixed randomization seed of 0 (via the `--deterministic` command line switch). The Java virtual machine was instructed to limit memory usage to approximately 8GB (flags `-XX:MaxRAM=8000M -Xmx3500M`) and the process running the JVM was terminated for time-out after 300 seconds (wall time).

In the following four paragraphs we discuss the properties of the input problems and briefly comment on the benchmark results.

Grounding Explosion is a benchmark modeled after the program shown in Example 1.3 and aims to exhibit the grounding bottleneck. Given a domain of size n, the problem is selecting at most one element from a domain of size n and deriving an atom $p(X_1, \ldots, X_6)$ where all $X_1, \ldots, X_6$ are the selected element. As reported

[1] Concrete instances are available via `http://www.kr.tuwien.ac.at/research/systems/alpha/instances.zip`, whereas Java code to generate new instances can be obtained from `https://github.com/alpha-asp/benchmarks`.

[2] Download freely available at `http://github.com/alpha-asp/alpha/releases/v0.2.0`.

in [21], significant differences in the results for this benchmark arise from the comparison of ground-and-solve vs. lazy-grounding systems. Because the difficulty in this benchmark lies in grounding, not in search and propagation, there is no relevant difference in run time expected, which aligns with our results as presented in Table 4.1.

Cutedge is taken from [6, Example 1]. This problem consists of computing reachability from a graph with exactly one edge removed. Our results (see Table 4.2) show that naïve propagation outperforms 3WL, which runs into memory issues: Because of the combinatorial nature of the problem (5.4 million nogoods for 500 vertices and adjacency-probability of 30%) and the more complex memory layout required by 3WL, it uses up all available memory. Naïve propagation performs well, because the search space is quite dense: For many calls to the propagation routine, 83% to 21% of nogoods are reported unit (some might be counted twice, as strongly and weakly unit counts were added up during analysis). With that many nogoods being unit, the naïve approach is faster because it does not have to do any bookkeeping operations to adjust references to watched literals.

Graph 5-Colorability is the problem of assigning one out of five possible colors to each vertex in a graph such that no connected vertices have the same color. Our results (shown in Table 4.3) clearly show an improvement when using 3WL compared to naïve propagation. We see many propagation cycles with just two to six nogoods (out of tens of thousands) being unit. While the naïve approach must check all nogoods, 3WL enables targeted propagation.

Reachability instances are positive programs that compute pairwise reachability of nodes in a graph. Again, sometimes as many as 10% to 20% of all nogoods are unit (with possible duplicates; nogoods that are strongly and weakly unit in one run are counted twice). This makes naïve propagation slightly faster than the 3WL approach. For positive programs, which contain no default negated literals, search is not necessary. Other systems such as `clingo` use *intelligent grounding* for this class of programs which could also be done for Alpha. We therefore include the results for this benchmark (Table 4.4) primarily for the sake of completeness, as it is not relevant for directly comparing naïve propagation with 3WL.

Summary. Evaluation of 3WL as implemented in the Alpha system has shown that the approach is especially effective when the number of nogoods that are not unit throughout search is low compared to the total set of nogoods. For these instances, speedups of several orders of magnitude were measured (Table 4.3). In cases where many nogoods propagate, bookkeeping operations (adjustment of watched literals) nullify gains in run time. Concerning memory usage, there is some overhead for 3WL, so the method generally runs into memory limits faster (i.e., for smaller instances) than the naïve approach.

Size	3WL	naïve
n	$\langle t_{1,\ldots,10} \rangle$ [s]	$\langle t_{1,\ldots,10} \rangle$ [s]
8	0.703	0.707
10	0.763	0.756
12	0.752	0.783
14	0.741	0.764
16	0.728	0.774
18	0.736	0.769
20	0.736	0.762
22	0.757	0.772
24	0.757	0.794
26	0.775	0.779
28	0.747	0.767
30	0.770	0.806
50	0.776	0.827
100	0.813	0.855
300	0.987	1.049
500	1.097	1.218
1000	1.452	1.626

Table 4.1: Results for grounding explosion benchmark: Time in seconds taken to compute 10 answer sets, averaged over 10 runs. Size corresponds to size of the domain.

Size		3WL			naïve		
$\|V\|$	$p(e)$	$\langle t_{1,\ldots,10} \rangle$ [s]	t/o	m/o	$\langle t_{1,\ldots,10} \rangle$ [s]	t/o	m/o
100	0.3	6.158	0	0	4.955	0	0
100	0.5	6.082	0	0	6.176	0	0
200	0.3	15.594	0	0	14.873	0	0
200	0.5	28.956	0	0	23.695	0	0
300	0.1	12.396	0	0	11.116	0	0
300	0.3	38.320	0	0	34.439	0	0
300	0.5	n/a	3	7	79.843	0	0
400	0.1	23.740	0	0	18.643	0	0
400	0.3	n/a	3	7	84.185	0	0
400	0.5	n/a	3	7	134.273	0	0
500	0.1	35.609	0	0	31.080	0	0
500	0.3	n/a	0	10	106.703	0	0
500	0.5	n/a	7	3	n/a	5	5

Table 4.2: Results for cutedge benchmark: Time in seconds taken to compute 10 answer sets, averaged over successful single runs on 10 randomly generated instances. Size is number of vertices combined with probability of any two vertices being connected. Number of timeouts and memory-outs is indicated in columns t/o and m/o, respectively.

Size		3WL			naïve		
$\|V\|$	$\|E\|/\|V\|$	$\langle t_{1,\ldots,10} \rangle$ [s]	t/o	m/o	$\langle t_{1,\ldots,10} \rangle$ [s]	t/o	m/o
10	4	1.374	0	0	0.916	0	0
20	4	0.845	0	0	1.373	0	0
30	4	1.025	0	0	2.077	0	0
40	4	1.161	0	0	2.938	0	0
50	1	0.910	0	0	2.205	0	0
50	2	1.032	0	0	2.626	0	0
50	4	1.290	0	0	4.461	0	0
50	6	8.494	2	0	23.105	4	0
50	8	61.043	7	0	n/a	10	0
50	10	41.317	3	0	n/a	10	0
75	4	1.682	0	0	8.983	0	0
100	4	2.305	0	0	16.840	0	0
200	4	4.940	0	0	115.977	0	0
300	4	7.017	0	0	n/a	10	0
400	4	10.375	0	0	n/a	10	0
500	4	12.979	0	0	n/a	10	0
750	4	25.820	0	0	n/a	10	0
1000	4	39.961	0	0	n/a	10	0

Table 4.3: Results for graph 5-colorability benchmark: Time taken in seconds to compute 10 answer sets, averaged over successful single runs on 10 randomly generated instances. Size is number of vertices combined with number of edges in relation to the number of vertices. Number of timeouts and memory-outs is indicated in columns t/o and m/o, respectively.

Size		3WL	naïve
$\|V\|$	$\|E\|/\|V\|$	$\langle t_{1,\ldots,10} \rangle$ [s]	$\langle t_{1,\ldots,10} \rangle$ [s]
1000	4	1.085	1.102
1000	8	1.737	1.607
10000	2	3.309	3.666
10000	4	6.779	5.416
10000	8	10.116	8.874

Table 4.4: Results for reachability benchmark: Time taken compute 10 answer sets, averaged over single runs on 10 randomly generated instances. Size is number of vertices combined with number of edges in relation to the number of vertices.

CHAPTER 5

Conclusion

We have presented a new method for unit propagation, tailored for lazy-grounding ASP solvers called Three Watched Literals (3WL). It is an extension of the Two Watched Literals scheme (2WL), prominent in SAT solvers. We extended 2WL for a third truth value *must-be-true*, which was found to greatly improve search performance for lazy-grounding in previous works. We introduced watch structures that can be easily implemented as the core data structure to perform 3WL.

Soundness of the new approach was analyzed in great detail and describes how assigning one atom initiates a series of propagation that restore coherence between assignment and watch structure at the point of termination.

Our evaluation shows that 3WL does not strictly perform better than a naïve approach, but yields considerable improvements when the portion of unit nogoods during search is small, which often happens for search intense problems. Here, a run-time improvement of several orders of mangnitude can be expected.

5.1 Related Work

This work heavily relies on previous results as cited. Most notably Moskewicz et al. [25], Zhang and Malik [36] described 2WL and Faber et al. [10] introduced *must-be-true* to ASP solvers.

5.2 Open Questions and Further Work

The algorithm presented in Chapter 3 does not consider guessing of assignments for atoms and backtracking in case of guesses leading to conflicts. One possibility for future work therefore is formalization of the behavior of 3WL including backtracking. Note that the implementation in the Alpha system, which was evaluated in Chapter 4 implements

backtracking, so there is some discrepancy between what was described and analysed here, and the concrete implementation. Based on this work, 3WL supporting backtracking was developed and is presented at [21].

3WL is expected to be complete in the sense that it computes all assignments that can be inferred through unit propagation. Cross-checking the implementation in Alpha in that regard was successful thus far. However, this work did not address completeness, and so a proof remains open as further work.

Evaluation results show that 3WL is not effective for instances where a "large" subset of nogoods propagates. Future work might address this issue and attempt to approximate threshold levels for the effectiveness of 3WL based on parameters of the input program. This line of research might lead to adaptive implementations that use heuristics to decide which method of propagation should be used. This proposes interesting challenges in the engineering aspects of solver systems, e.g. interfacing with different unit propagation modules within the same solver system.

[23] surveys propagation schemes in SAT solvers. An extension for *must-be-true* similar to the one in this work could be done to find out whether the results also hold for three truth values.

One issue that is not directly connected to 3WL, but to addressing the grounding bottleneck more generally, is dynamic deletion of nogoods. By deleting "inactive" nogoods during search, additional memory might be freed, allowing solvers to work with larger search spaces. As the watch structure holds all nogoods, integrating nogood deletion might well be tightly coupled with propagation.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Chandrabose Aravindan, Jürgen Dix, and Ilkka Niemelä. Dislop: A research project on disjunctive logic programming. *AI Commun.*, 10(3-4):151–165, 1997. URL http://content.iospress.com/articles/ai-communications/aic130.

[2] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. Asp-core-2: Input language format. *ASP Standardization Working Group, Tech. Rep*, 2015. https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03c.pdf.

[3] Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The dlv system: Model generator and advanced frontends (system description). In *WLP*, page 0, 1997.

[4] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, 1977.*, Advances in Data Base Theory, pages 293–322, New York, 1977. Plemum Press. ISBN 0-306-40060-X.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. doi: 10.1145/800157.805047. URL http://doi.acm.org/10.1145/800157.805047.

[6] Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. Omiga : An open minded grounding on-the-fly answer set solver. In Luis Fariñas del Cerro, Andreas Herzig, and Jérôme Mengin, editors, *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings*, volume 7519 of *Lecture Notes in Computer Science*, pages 480–483. Springer, 2012. ISBN 978-3-642-33352-1. doi: 10.1007/978-3-642-33353-8_38. URL http://dx.doi.org/10.1007/978-3-642-33353-8_38.

[7] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.*, 51(2-4):123–165, 2007. doi: 10.1007/s10472-008-9086-5. URL https://doi.org/10.1007/s10472-008-9086-5.

[8] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009. ISBN 978-3-642-03753-5. doi: 10.1007/978-3-642-03754-2_2. URL http://dx.doi.org/10.1007/978-3-642-03754-2_2.

[9] Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, 2009. Springer. ISBN 978-3-642-04237-9. doi: 10.1007/978-3-642-04238-6. URL https://doi.org/10.1007/978-3-642-04238-6.

[10] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pushing goal derivation in DLP computations. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Logic Programming and Nonmonotonic Reasoning, 5th International Conference, LPNMR'99, El Paso, Texas, USA, December 2-4, 1999, Proceedings*, volume 1730 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 1999. ISBN 3-540-66749-0. doi: 10.1007/3-540-46767-X_13. URL https://doi.org/10.1007/3-540-46767-X_13.

[11] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982. doi: 10.1016/0004-3702(82)90020-0. URL https://doi.org/10.1016/0004-3702(82)90020-0.

[12] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007. ISBN 978-3-540-72199-4. doi: 10.1007/978-3-540-72200-7_23. URL https://doi.org/10.1007/978-3-540-72200-7_23.

[13] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012. doi: 10.1016/j.artint.2012.04.001. URL http://dx.doi.org/10.1016/j.artint.2012.04.001.

[14] Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the sixth answer set programming competition. In Francesco Calimeri, Giovambattista Ianni, and

Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *Lecture Notes in Computer Science*, pages 531–544. Springer, 2015. ISBN 978-3-319-23263-8. doi: 10.1007/978-3-319-23264-5_44. URL https://doi.org/10.1007/978-3-319-23264-5_44.

[15] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988. ISBN 0-262-61056-6.

[16] Carla P. Gomes, Henry A. Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In van Harmelen, Lifschitz, and Porter [33], pages 89–134. ISBN 978-0-444-52211-5. doi: 10.1016/S1574-6526(07)03002-7. URL https://doi.org/10.1016/S1574-6526(07)03002-7.

[17] Claire Lefèvre and Pascal Nicolas. The first version of a new ASP solver : Asperix. In Erdem, Lin, and Schaub [9], pages 522–527. ISBN 978-3-642-04237-9. doi: 10.1007/978-3-642-04238-6_52. URL https://doi.org/10.1007/978-3-642-04238-6_52.

[18] Claire Lefèvre and Pascal Nicolas. A first order forward chaining approach for answer set computing. In Erdem et al. [9], pages 196–208. ISBN 978-3-642-04237-9. doi: 10.1007/978-3-642-04238-6_18. URL https://doi.org/10.1007/978-3-642-04238-6_18.

[19] Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. Asperix, a first-order forward chaining approach for answer set computing. *TPLP*, 17(3): 266–310, 2017. doi: 10.1017/S1471068416000569. URL https://doi.org/10.1017/S1471068416000569.

[20] Nicola Leone and Wolfgang Faber. The DLV project: A tour from theory and research to applications and market. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2008. ISBN 978-3-540-89981-5. doi: 10.1007/978-3-540-89982-2_10. URL https://doi.org/10.1007/978-3-540-89982-2_10.

[21] Lorenz Leutgeb and Antonius Weinzierl. Techniques for efficient lazy-grounding ASP solving. In *Applications of Declarative Programming and Knowledge Management - 21st International Conference, INAP 2017, Würzburg, Germany, September 19-22, 2017*, Lecture Notes in Computer Science. Springer, 2017. to appear.

[22] Lengning Liu, Enrico Pontelli, Tran Cao Son, and Miroslaw Truszczynski. Logic programs with abstract constraint atoms: The role of computations. In Verónica

Dahl and Ilkka Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2007. ISBN 978-3-540-74608-9. doi: 10.1007/978-3-540-74610-2_20. URL https://doi.org/10.1007/978-3-540-74610-2_20.

[23] Inês Lynce and João P. Marques Silva. Efficient data structures for backtrack search SAT solvers. *Ann. Math. Artif. Intell.*, 43(1):137–152, 2005. doi: 10.1007/s10472-005-0425-5. URL https://doi.org/10.1007/s10472-005-0425-5.

[24] Jack Minker. *On indefinite databases and the closed world assumption*, pages 292–308. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982. ISBN 978-3-540-39240-8. doi: 10.1007/BFb0000066. URL https://doi.org/10.1007/BFb0000066.

[25] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. ISBN 1-58113-297-2. doi: 10.1145/378239.379017. URL http://doi.acm.org/10.1145/378239.379017.

[26] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Answer set programming with constraints using lazy grounding. In Patricia M. Hill and David Scott Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2009. ISBN 978-3-642-02845-8. doi: 10.1007/978-3-642-02846-5_14. URL https://doi.org/10.1007/978-3-642-02846-5_14.

[27] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009. doi: 10.3233/FI-2009-180. URL https://doi.org/10.3233/FI-2009-180.

[28] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4. URL http://www.sciencedirect.com/science/bookseries/15746526/2.

[29] Francesca Rossi, Peter van Beek, and Toby Walsh. Constraint programming. In van Harmelen et al. [33], pages 181–211. ISBN 978-0-444-52211-5. doi: 10.1016/S1574-6526(07)03004-0. URL https://doi.org/10.1016/S1574-6526(07)03004-0.

[30] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers. Master's thesis, Simon Fraser University, Burnaby, Canada, 2004. URL https://www.cs.sfu.ca/~mitchell/papers/ryan-thesis.ps.

[31] Dietmar Seipel and Helmut Thöne. DISLOG - A system for in disjunctive deductive databases. In *DAISD*, pages 325–343, 1994.

[32] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999. doi: 10.1109/12.769433. URL https://doi.org/10.1109/12.769433.

[33] Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter, editors. *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*. Elsevier, 2008. ISBN 978-0-444-52211-5. URL http://www.sciencedirect.com/science/bookseries/15746526/3.

[34] Antonius Weinzierl. Learning non-ground rules for answer-set solving. In *Proceedings of the Second Workshop on Grounding and Transformation for Theories with Variables (GTTV'13)*, pages 25–37, 2013.

[35] Antonius Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In Marcello Balduccini and Tomi Janhunen, editors, *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, volume 10377 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2017. ISBN 978-3-319-61659-9. doi: 10.1007/978-3-319-61660-5_17. URL https://doi.org/10.1007/978-3-319-61660-5_17.

[36] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002. ISBN 3-540-43931-5. doi: 10.1007/3-540-45620-1_26. URL https://doi.org/10.1007/3-540-45620-1_26.

[37] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In Rolf Ernst, editor, *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, pages 279–285. IEEE Computer Society, 2001. ISBN 0-7803-7249-2. doi: 10.1109/ICCAD.2001.968634. URL https://doi.org/10.1109/ICCAD.2001.968634.