A Versatile Intermediate Language for Answer Set Programming: Syntax Proposal

Martin Gebser [1] Tomi Janhunen [2] Max Ostrowski [1]
Torsten Schaub [1] Sven Thiele [1]

[1] Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam, Germany
{gebser,ostrowsk,torsten,sthiele}@cs.uni-potsdam.de

[2] Helsinki University of Technology, Department of Information and Computer Science,
P.O. Box 5400, FI-02015 TKK, Finland
Tomi.Janhunen@tkk.fi

September 16, 2008

Abstract

The attractiveness of Answer Set Programming (ASP) and related paradigms for declarative problem solving is considerably due to the availability of highly efficient yet easy-to-use implementations. Standardized problem representations form a major driving force for the development and improvement of tools for several reasons. First, they relieve developers from the burden of inventing their own input formats. Second, they establish interoperability between separate tools, allowing users to easily compare and exchange them without extensively converting their problem representations. Third, they facilitate the acquisition of problem descriptions from distinct sources, which is useful for benchmarking and assessment purposes. Historically, however, standards for representing logic programs, serving as inputs to ASP systems, were mainly dictated by few available tools. In fact, there currently are two quasi-standard formats, viz. the formats of *lparse* and *dlv*, incompatible with each other. As a first step towards overcoming this deficiency, this work proposes an intermediate format for ground logic programs, intended for the representation of problem instances as inputs to ASP solvers. The format is not designed to be a primary input language, given that ASP systems usually deploy a second component, called a grounder, to deal with the inputs provided by users. In view of this, our format is situated intermediate a grounder and a solver, guided by the example of grounder *lparse* and solver *smodels*, the latter marking the first among nowadays a variety of solvers processing the output of *lparse*. However, the output format of *lparse* has some decisive drawbacks, viz. its restrictive range and limited extensibility. We thus propose a new intermediate language, where our major design goals are flexibility in problem representation and easy extensibility to new language constructs.

Contents

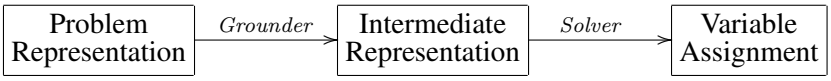

Figure 1: Basic Architecture of an ASP System

1 Introduction

Answer Set Programming (ASP; [1, 18, 31, 37]) is a declarative approach to modelling and solving search problems, represented as logic programs. As illustrated in Figure 1, an ASP system usually deploys two components: a *grounder* and a *solver*. The input to an ASP system typically consists of a non-ground problem encoding and a ground problem instance. In such *uniform* encodings, the use of first-order variables reduces the size of the representation and permits simpler, and therefore easier to write, logic programs. Furthermore, regarding the input language, several extensions have been proposed, like *aggregates*, *cardinality and weight constraints*, and *optimize statements* [6, 26, 43]. A grounder translates such a problem representation, which typically consists of a pair of an encoding and an instance in the input language, into a ground logic program, represented in a simplified, solver-readable form. Starting from the grounder's output, the solver then searches for answer sets, corresponding to solutions of the original problem. The most common solving approaches are based on the Davis-Putnam-Logemann-Loveland (DPLL; [4, 5]) algorithm, like in *dlv* [26] and *smodels* [43], or Conflict-Driven Clause Learning (CDCL; [32, 33, 35]), e.g., used in *clasp* [15].

There currently is a single intermediate language accessible to ASP solvers, namely, the output format of grounder *lparse* [44].[1] However, the format is not standardized and might thus change over different *lparse* versions, which is a delicate issue since no version information is included, e.g., for backward compatibility. The latter also makes ad-hoc extensions of *lparse*'s output format intricate and error-prone.[2] Furthermore, the fact that *lparse*'s output format is designed to match *smodels*' internal data structures necessitates program transformations incurring a loss of structural information [30]. We thus consider the restrictedness, on the one hand, and the limited extensibility, on the other hand, of *lparse*'s output format as serious drawbacks, making it unsuitable as a general standard.

This work proposes a new intermediate format, called *ASPils* ("ASP intermediate language standard"), for the use in-between grounders and solvers. Important design goals are:

- simplicity and efficiency in outputting and parsing;
- independence of grounder and solver implementations;
- support of the existing (input) language constructs (cf. [6, 26, 44]);
- support of version information, meta-information, and user comments; and
- flexibility and easy extensibility.

[1] The *dlv* system uses an internal grounder that is directly coupled with the solver.

[2] For instance, the tool *dencode* [21] includes functionality for making conversions between the disjunctive rule output formats of old (up to version 1.0.14) and new versions of *lparse*.

The development of *ASPils* is inspired by experiences made in related fields, such as Boolean Satisfiability (SAT), having standardized problem description languages, e.g., DIMACS format [7]. For one, such standardized languages establish interoperability between solvers and further tools, for instance, tools generating solver inputs. As a concrete example in ASP, a standardized intermediate language might enable (arbitrary) solvers to process the output of *dlv*'s grounding component, and also *dlv*'s solving component to process the output of an external grounder. From a user's point of view, interoperability facilitates running different solvers, as a problem encoding written in the input language of a particular grounder could after grounding be processed by an arbitrary solver. Furthermore, a standardized intermediate language supported by all solvers would greatly foster ASP solver competitions, where in the past the different input formats of *dlv* and other solvers have been a major bottleneck [16]. In fact, as a secondary benefit, a common language eases collecting challenging benchmarks from distinct sources and might thus push the further development of ASP solvers, like it has been experienced in SAT.

In order to put this document in perspective, let us stress that *ASPils* is proposed as a standard for the transmission of ground logic programs from grounders to solvers. At this stage, our proposal aims at recording the language constructs currently supported by *lparse*-based solvers as well as *dlv*'s solving component and at integrating them into a common framework, also anticipating future extensions to a certain extent. Of course, an input format for ASP solvers can only turn into a standard if it is widely supported and used in practice, which requires a community effort, especially, from ASP system developers. Our proposal of *ASPils* thus aims at providing a starting point for a community-wide discussion of a standardized input format for ASP solvers. Even if such a standard is successfully established, it will not instantly abolish all differences and peculiarities of ASP systems. For instance, grounders and integrated ASP systems may further (have to) use proprietary input languages, and any incremental [14] or even a system not following the computational pattern shown in Figure 1 might not be readily supplied with an appropriate input format. However, before succeeding to standardize the simplest element in the workflow of ASP systems, namely, the intermediate language used in-between a grounder and a solver, any attempts to standardize more diversified matters would most likely be prone to fail. Hence, even though the scope of *ASPils* is limited to an intermediate representation according to Figure 1, we think that it may initiate a worthwhile discussion on language standards in ASP.

The remaining sections of this document are organized as follows. We start by reviewing existing (intermediate) languages for ASP and SAT in Section 2, thereby, highlighting and discussing their pros and cons. In Section 3 and 4, we describe the design of *ASPils* in general and in detail, respectively. Section 5 specifies a number of normal forms, reflecting language fragments of *ASPils* corresponding to different syntactic primitives of existing ASP solvers. Finally, we present the conclusions of this proposal in Section 6. The grammar of *ASPils* in Extended Backus-Naur Form (EBNF; [8]) can be found in Appendix A.

2 Review of Intermediate Languages for ASP and SAT

The following comparison of existing intermediate languages for SAT and ASP builds upon [22], that is, we here review the same intermediate languages and in addition also MidL Normal Form, but we sometimes draw different conclusion, for instance, as regards the extensibility

of SModels format. Note that we do not consider the input language of *dlv* here, as it is a textual format primarily aimed at the grounding component of *dlv* and thus too complex for a solver-oriented intermediate language. For the same reason, we omit a discussion of the Rule Interchange Format (RIF; [41]) and the Satisfiability Modulo Theories Library (SMT-LIB; [40]) format, both dealing with (fragments of) first-order logic.

2.1 BC Format

The BC file format [23] is used by the solver *bcsat* [24] for Boolean circuit satisfiability. The first line of a file in BC format provides the format version. In the remainder, the file defines the structure of the circuit in terms of *gate definitions* each of which associates a propositional formula with a name in a recursive fashion. Such formulas may involve prefix operators, like `AND`, `OR`, etc., or infix operators, like `&`, `|`, etc. Thereby, symbolic names can be used to refer to particular gates in the circuit. The newline symbol is used as delimiter between gate definitions.

Advantages and Disadvantages. The file format supports version information, thus, changes in the language can be recognized easily. Furthermore, the format is human-readable as symbolic names are used for gates and operators. However, symbolic names and different notations (prefix and infix) for operators make parsing more complex and therefore fault-prone.

2.2 DIMACS Format

There are two different DIMACS input formats [7], one for propositional formulas in CNF and one for (arbitrary) propositional formulas. In addition, there is a particular output format, a modification of which is used in SAT competitions [42]. All three formats are numerical and support comments via lines starting with letter `c`.

CNF

A problem description in CNF format starts with a so-called problem line, whose first letter `p` is followed by constant string `cnf` and the number of variables and clauses in the problem. The remainder of the problem description consists of clauses, that is, lists of literals terminated by number `0`. Literals are non-zero integers whose absolute values stand for atoms.

SAT

A problem description in SAT format also starts with a problem line similar to CNF format, but replacing `cnf` by `sat` and without providing a number of clauses. After the problem line, a formula is specified in a single statement composed of infix operators $-$, $\star$, and $+$ along with parentheses. (In the extended `satex` format, `xor` and = are additional infix operators.)

Output

Although it is not an intermediate format, the DIMACS output format is oriented at CNF and SAT formats. The output format allows one to report information about the satisfiability of a

problem as well as a solution in a standardized way, so that it can be post-processed automatically. Also, timing information (how long has it taken to compute the solution) can be included.

Advantages and Disadvantages. The simplicity of the formats, in particular, CNF and output formats, make them easy to parse or to produce by SAT solvers. The use of number `0` as clause delimiter in CNF format allows clauses to span over multiple lines, and it avoids the interpretation of newline as an implicit delimiter. However, as the formats do not support symbolic information, it can only be included in an application-specific way via comments. The number of variables (and clauses) in problem lines can be useful for preparing the internal data structures within SAT solvers, but tools generating the formats need to know these numbers in advance. In ASP, providing such information at the start of a problem description would compel grounders to store the complete ground instantiation of a logic program before outputting it.

2.3 GCore Format

The GCore format [36] for ground logic programs is a fragment of *lparse*'s input language [44] in which all atoms are ground and referred to by names of the form $\mathtt{v}i$. If $\mathtt{v}i$ and $\mathtt{v}j$ occur in a program, then all atoms in-between $\mathtt{v}i$ and $\mathtt{v}j$ must occur as well, that is, gaps are not allowed.

Advantages and Disadvantages. Due to the similarity with the input language of *lparse*, the format is human-readable, but with meaningless atom names. The drawback of textual format is that it is harder to parse by solvers than a numerical format. In comparison to the output language of *lparse* (see SModels format below), an advantage is that the structure of logic programs is preserved, since extended rules of *lparse*'s input language are directly supported.

2.4 MidL Normal Form

MidL Normal Form (MNF; [34]) is a format for propositional ID-Logic, modifying DIMACS format (cf. Section 2.2). In the problem line, constant string `midl` is used as the format name, and in contrast to other DIMACS input formats, the number of variables is not provided. The actual problem is represented by clauses as in CNF format, and in addition, statements of the form `C atom clause 0` or `D atom clause 0`, where `atom` is a positive integer and `clause` is a list of non-zero integers. Furthermore, `C` and `D` indicate whether `atom` is equivalent to the conjunction or disjunction of `clause`, respectively. Via an optional footer, MNF allows for providing symbolic information. Rather than using a table for mapping integers to names, the domains of functions and predicates are given via types, and individuals can be reconstructed from base numbers. For instance, the footer

```
type T : A B C
pred P 1 : T T
pred Q 10 : T
```

declares that predicate `P` is of type $\mathtt{T}\times\mathtt{T}$, and its instances are numbered from `1=P(A,A)` to `9=P(C,C)`. Similarly, we have `10=Q(A)`, `11=Q(B)`, and `12=Q(C)`. By applying this convention, the symbolic names of atoms can be reconstructed and provided to users.

Advantages and Disadvantages. As MNF extends DIMACS format, it is numerical and thus easy to parse. In contrast to DIMACS format, symbolic information is supported and can be represented in a succinct way, as the instances of a function or predicate are not explicitly listed. However, base numbers can become very large, as every syntactically constructible instance of a function or predicate implicitly has a dedicated number, even if it is not referred to.

2.5 PB Format

The Pseudo-Boolean (PB; [39]) format may be viewed as a textual version of CNF format (cf. Section 2.2). In fact, the format supports comments via lines starting with symbol `*`, and according to the rules of the 2007 PB evaluation, the first line of an input file provides the numbers of atoms and constraints inside a comment. The first non-comment line of an input file can specify an objective function, starting the line with `min:`. The remainder of the file specifies Pseudo-Boolean constraints, that is, inequalities and equalities, using connectives +, −, >=, and =. Each (in)equality is provided within a single line terminated by symbol `;`. Analogously to GCore format (cf. Section 2.3), variables occurring within inequalities and equalities are named schematically by $\texttt{x}i$ where i ranges from 1 to the number of variables declared in the first line of an input file.

Advantages and Disadvantages. The format is human-readable, but as with GCore, atom names are meaningless. Similar to DIMACS formats (cf. Section 2.2), providing the numbers of atoms and constraints can help solvers to prepare their internal data structures; however, the strategy to provide such information inside comments is questionable. Finally, the format specification [39] grants certain conditions only within the 2007 PB evaluation environment, but not for user inputs. A simpler (numerical) format might exclude such sources of parsing errors.

2.6 SModels Format

The input language of the *smodels* solver, usually generated by means of grounder *lparse*, is described in [44]. It allows for the representation of ground logic programs in a numerical format, not supposed to be human-readable. Numerical program representations consists of three blocks, separated from each other by a single line containing only number `0`. The first block provides the rules of a program, supporting eight different rule types identified via a dedicated number at the beginning of a line. The remainder of the line specifies an individual rule in a type-specific way, e.g., providing the head and the body of a normal rule. The second block contains the symbol table of the program, which maps the names of atoms to unique positive integer IDs that have been used in the first block to refer to atoms. Note that an atom used in the first block, that is, its corresponding ID, does not have to occur in the symbol table, in which case the atom is considered to be *hidden*. The third and last block contains "compute statements" in which atoms can be declared to be either true or false. The last item of the third block is the number of answer sets to compute.

Advantages and Disadvantages. The SModels format is recognized by a variety of ASP solvers and used in ASP contests [16]. By means of the symbol table, it is possible to reconstruct

a human-readable format and to represent answer sets in terms of symbolic atom names, which is essential in applications. However, the supported rule types are designed to match *smodels*' internal data structures and thus impose limitations that might lead to a loss of structural information. For instance, a cardinality or weight constraint can only specify a lower bound. If it originally had an upper bound too, it is translated into another constraint, not belonging to the input program. Moreover, negative weights and bounds for weight constraints are not allowed, and due to complexity reasons, available compilations to get rid of them [43] may lead to counter-intuitive results [12]. The block structure of the format compels grounders to accumulate the whole symbol table in main memory before outputting it, which might be unnecessary if a grounder were allowed to output parts of the symbol table in-line. This limitation of the SModels format is partly circumvented by *modlist* and *lpcat* (see [21] for descriptions) in terms of *module sequences*, which are streams of programs in the SModels format. However, the definition of *input* atoms that are defined outside a module is based on an ad-hoc assumption[3] since input atoms are not distinguished by the SModels format. Yet another issue is the lack of version and meta-information, so that changes in the language cannot be recognized easily.

2.7 Conclusions

In [22], several aspects of intermediate languages, like the file format (text or binary), version and meta-information, comments, symbolic names, and extensibility, are discussed. Let us take up some of these issues along with the observations made in the previous subsections:

- All of the discussed intermediate languages use a text format, and not a binary one. In fact, a text format has the advantage of being system-independent, but it might be of greater size than a binary format. However, common compression methods are able to reduce the size of text and binary formats of the same contents to approximately the same size. Thus, we think that system independence is more important than an a priori reduction of size.

- Most of the formats use a numerical representation for atoms and their interactions. From the viewpoint of a solver, this eases parsing and thus reduces the likelihood of parsing errors. Via symbol tables, symbolic information can be reconstructed even from a numerical format. In the previous subsections, we have seen two alternative approaches to represent symbol tables: in MNF (cf. Section 2.4), a naming scheme is used to compact symbolic information, while SModels format (cf. Section 2.6) provides symbolic names for atoms individually. Both ideas have their pros and cons, as base numbers used in MNF can become large, and symbol tables in SModels format may contain many entries (but still bound linearly in the size of ground logic programs). We note that numerical formats are not considered to be human-readable, thus, they cannot be changed manually once they have been generated. This drawback seems marginal in the context of ASP, since the input language visible to users is that of a grounder, but not the intermediate language.

[3] An input atom has a name in the symbol table of the program but no defining rules. It is worth noting that, by default, such atoms would be assigned false under the stable model semantics, but proper input atoms can take any truth value [38].

- Some formats, that is, CNF and SModels format (cf. Section 2.2 and 2.6), impose restrictive normal forms for their statements. In the process of generating such formats, this often incurs a loss of structural information, which even led to approaches dealing with restoring the lost structure [9, 30]. To abolish the need for such workarounds, an intermediate format should be expressive and flexible enough to retain problem structure.

- An important detail lies in the choice of a statement delimiter, where we have seen two alternative solutions: using explicit delimiters (such as `0`, `()`, `;`, or `.`) or newline as an implicit delimiter. The second choice forbids statements to span over multiple lines, which can be problematic if a computational environment does not allow for arbitrarily long lines. Furthermore, the use of an explicit delimiter seems less fault-prone, to the price of adding one more symbol to statements where otherwise newline would be sufficient.

- For enabling extensibility, it is crucial to include version information, so that solvers can adjust their parsers or terminate in a well-defined way in case of an unrecognized (potentially too new) version. The provision of a version numbers is also necessary in order to address backwards compatibility in various tools. Another difficulty in ASP are different language fragments supported by solvers, for instance, solver *assat* [29] does not handle extended rules [43], and *smodels*$_{cc}$ [45] does not process weight constraints. By also including information on the language fragment, solvers not supporting it can terminate appropriately, that is, not with an error accidentally raised during parsing.

- Some formats, e.g., DIMACS format (cf. Section 2.2), include meta-information, such as the number of atoms in a problem. This can be useful for solvers, but the generation of a format becomes harder if a priori knowledge is required. In ASP, grounders may produce the intermediate format on-the-fly, that is, parts may be output before the whole ground logic program is generated. In such a setting, meta-information about the final problem description is unavailable. We thus think that including meta-information should be optional, and solvers might ignore or use the available information, depending on whether they recognize and trust it. Furthermore, we think that there is a difference between meta-information and comments, as the latter should always be ignored by solvers. That is, comments should be completely up to users, and passing solving information inside them (as done in PB format; cf. Section 2.5) must be avoided.

- For benchmarking purposes, it is advantageous if solver input can be shuffled easily. In other settings, several problem descriptions might need to be combined into a new problem. Thus, it is desirable that statements provided in problem descriptions are modular (like, e.g., clauses in CNF format; cf. Section 2.2) and that they do not rely on a particular order of the syntactic elements in the statements as far as possible. Indeed, this principle is one of the underlying assumptions of declarative programming.

3 General Design of *ASPils*

We now briefly describe the design decisions underlying *ASPils*, the new intermediate language for ASP we propose here. Our global goal is to specify a language that has the potential to

become a standard format for inputs to ASP solvers. Thus, we have to respect that different ASP solvers support different language constructs, e.g., *dlv* deals with aggregates [6] and *smodels* with extended rules [43]. In order to reflect this diversity, our language must be general and solver-independent. Furthermore, it is impossible to foresee language constructs that might evolve in the future. Hence, extensibility of the language is an important issue. We thus include a *version number* in problem descriptions of *ASPils*. In addition, *normal forms* are used to specify language fragments. Their main purpose is to reflect the variety of (syntactic) primitives supported by solvers, which are thus enabled to check whether a problem description given as input is an appropriate one before processing it any further.

The body of *ASPils* consists of *entries*, mainly defining *objects* of particular *types*. The idea is similar to the output format of *lparse* [44], using several rule types. However, *ASPils* goes further than *lparse* by not restricting types to rules, rather, all entities in a ground logic program, e.g., atoms, conjunctions, disjunctions, etc., are objects having a type. An advantage of this is that complex structures occurring in a program, e.g., conjunctions of cardinality and weight constraints, can be represented in a modular and structure-preserving way. In contrast to this, *lparse* would have to introduce new atoms and rules to represent complex structures in its restrictive output format. Another advantage of types is the easy embedding of new language constructs, as it only requires the definition of a type identifier and a syntax for objects of the type. To avoid clashes of custom type identifiers, we use numbers consisting of a "major" and a "minor" slot (similar to IP addresses), where the major slot ought to be related to a research group defining the type. If newly introduced types turn into a standard, they can be integrated into *ASPils* via the introduction of an additional normal form or even a new language version.

As mentioned above, every *object* occurring in a problem description has an associated type. In addition, each object has a unique *object identifier*, or *object ID* for short. An object ID is a positive integer that gives an identity to the object and hence enables subsequent references to the object. This makes the language modular because, on the syntactic level, other objects make use only of the object ID of an referenced object, but not of its internal structure. Hence, if new language constructs are introduced, their objects can immediately be used within available structures, without needing to exchange them. A second potential benefit of object IDs is the possibility to re-use them if the same object has multiple occurrences in a ground logic program, thus compacting the representation in analogy to gate definitions in the specifications of Boolean circuits. Note that this accounts for ordinary propositional atoms as well as for non-atomic structures, such as conjunctions and disjunctions.

On the technical level, our language shall be easy to parse, independent of system environments, and resistant against potential parsing errors. To this end, we use a numerical text format and number `0` as an explicit delimiter for entries. As `0` can also occur within an entry (not as delimiter), each entry must specify its number of consecutive (numeric and/or symbolic) parameters directly after its type. This shall enable the correct syntactic decomposition of *ASPils* sentences, even without truly understanding the contents, and it deliberately introduces a layer of redundancy in order to avoid parsing errors. Also, note that most parameters occurring in entries are numeric and thus represented by integers, which should facilitate their recognition by solvers. In particular, negative integers are used to denote the *default negations* of objects having the absolute values as their IDs. Of course, the use of numbers makes *ASPils* less human-readable, but obtaining human-readability is not among our desiderata anyway. Symbolic information can

still be included, most likely, for defining atom names, but usually such information needs not be interpreted by solvers. Furthermore, arbitrary meta-information as well as user comments can be provided using dedicated types. Note that comments are the only kind of objects not having an ID, as they ought to be ignored by solvers. In contrast, meta-information may be exploited by solvers, but it should not be mandatory for solvers to recognize it. We here do not suggest using any kind of meta-information, however, including information such as whether a ground logic program has an answer set or whether it is tight [11, 10] might be useful for particular purposes.

4 Language Description

This section details the elementary constituents of our proposed intermediate language *ASPils*. We focus on intuitions and examples because the formal specification of *ASPils* is given by the grammar in Appendix A. Below, *italic* and `typewriter` fonts indicate non-terminals and terminals, respectively, in the grammar. Consecutive terminal symbols (other than digits of *integer*s or characters of *string*s and *verbal*s) must be separated by whitespace (including characters for space, tabulation, carriage return, line feed, and form feed).

4.1 Header

Every sentence of *ASPils* starts with a *header*.[4] For example, header

```
1 3 1 3 0 0
```

consists of a `1` indicating the *type header*, a `3` providing the number of parameters before the delimiter, the second `1` stating that this is the first *version* of *ASPils*, the second `3` indicating conformance to *normal form* "SModels" (cf. Section 5.4), a `0` stating that there are no *additional headers*, and the second `0` delimiting the header. Note that language version `1` of *ASPils*, defined in this document, does not specify any additional headers, hence, their number will always be `0`. The pattern that a type number is followed by the number of parameters before delimiter `0` recurs in each of the types described below, while the parameters themselves are specific.

4.2 End Of File

Every sentence of *ASPils* is terminated with an occurrence of *object eof*,[4] looking as follows:

```
0 0 0 .
```

The first `0` indicates the object's type, the second its number of parameters, and the third one delimits it. (The full stop sign "." is part of the surrounding text, but not of *ASPils*.)

4.3 Entries

In-between the header and end of file, a ground logic program is specified by entries, providing meta-information, comments, or defining elements of the program at hand. Each *entry* starts

[4] Only *comment*s are allowed before *header* and after *object eof*.

with its type number, an up to eight digits long hexadecimal cipher. The first four digits denote the research group that has developed the type; for the core types described below, the four leading digits are zeros and can thus be omitted (as done in Appendix A). The last four digits of a type number must not evaluate to zero (the value reserved for *object eof*). Each type number is followed by the number of consecutive parameters before the *entry* is delimited by an occurrence of `0`. Each entry type imposes particular parameters, that is, the slots specified in the grammar (see Appendix A) must be filled with appropriate terminals.

Meta-Information. Objects of type `2` can be used to provide meta-information. Although we do not suggest any particular meta-object, the following one is syntactically valid:

```
2 3 42 "tight program" 23 0 .
```

Among the `3` parameters of the entry, `42` is the *object ID*, `"tight program"` is a *safe verbal*, that is, a list of strings and whitespaces enclosed in double quotes, and `23` is the single element of a list of *meta-option*s (that is, *integer*s). Meta-information may be exploited by solvers, but any such information should not affect the semantics of the specified ground logic program, so that it is admissible for solvers to simply ignore unrecognized meta-objects.

Comments. Comments of type `3` do not define any object (that is, they do not have an object ID as parameter). An example comment looks as follows:

```
3 1 "grounded by GrinGo version 2" 0 .
```

As *comment*s are not associated with an ID, they cannot be referenced by objects defined in an *ASPils* problem description. In fact, comments are understood to be completely up to user information, such as the author of a problem description or the grounder that generated it. Unlike meta-information, comments must always be ignored by solvers.

Atoms. Objects of type `4` define atoms. For instance, consider:

```
4 2  8   p(a,1)         0
4 5 15 "-p(a,1)" 1 2 8 0 .
```

The first entry specifies that *object ID* `8` stands for an *atom* whose name is `p(a,1)`, and the second entry defines *object ID* `15` to represent another atom with name `-p(a,1)`.[5] Furthermore, *atom option* `1`, provided for `-p(a,1)`, declares the atom as *hidden*, that is, the atom name shall be suppressed in the output of an ASP solver. We include this option in order to reflect the effect of hide declarations in the input language of *lparse* [44]. However, while *lparse* suppresses atom names by not including them in the symbol table, we choose to keep the symbolic names of atoms and to signal their hidden nature via an option. In this way, it stays possible to recover a symbolic representation from the intermediate format, as it is done by tool *lplist* [21] for *lparse*'s

[5] Note that the name of the second atom must be provided as a *safe verbal*, enclosed in double quotes. Double quotes may only be omitted for *atom name*s starting with a *letter*. Due to this requirement, the first symbols of *atom name*s and *integer*s become unambiguous.

output format using the symbolic information still available there. The second *atom option* `2` for `-p(a,1)` declares the atom to be the classical negation of the object with ID `8`, viz., of atom `p(a,1)`. Classical negation is understood in the sense of [19]. In our example, it means that atoms `p(a,1)` and `-p(a,1)` cannot jointly belong to a stable model of the program at hand. Furthermore, *module option*s, `3` and `4`, may be used to declare an atom to be an *input* atom or a *local* one, respectively, with respect to the concept of logic program modules in [38]. Intuitively speaking, input atoms are defined by some other module whereas local atoms are defined in the current module and hence they cannot be referenced by other modules.

Rules, Facts, and Integrity Constraints. In ASP, a logic program is a set of rules, each rule consisting of a *head* and a *body*. Either the head or the body may be constant, in which case the rule is called a *fact* or an *integrity constraint*, respectively. Let us consider the following example logic program containing a rule, a fact, and an integrity constraint:

```
-p(a,1) :- not p(a,1).
 p(a,1).
        :- -p(a,1).
```

Reusing *object ID*s `8` and `15` for `p(a,1)` and `-p(a,1)`, respectively, corresponding entries in *ASPils* can be represented as follows:

```
5 3 55 15 -8 0
6 2 66  8  0
7 2 77 15  0 .
```

Here, type numbers `5`, `6`, and `7` indicate that the objects with IDs `55`, `66`, and `77` are a *rule*, a *fact*, and an *integrity constraint*, respectively. Note that `-8` in the first entry refers to the default negation of the atom with ID `8`, viz., of atom `p(a,1)`. Furthermore, observe that the second entry specifies only a head *literal* and the third one only a body *literal*, while the rule defined by the first entry contains both a head and a body *literal*. The choice of introducing three different types is motivated by the goal of not imposing any hard-wired assumptions on the structure of heads and bodies, while still being able to identify the role of particular *literal*s. Importantly, workarounds such as the `_false` atom, introduced by *lparse* [44] as the head of integrity constraints, ought to be avoided. Due to the generic design of entries for rules, facts, and integrity constraints, allowing to refer to arbitrary and possibly default negated objects, there are no restrictions on the structure of heads and bodies (rules might even reference each other) a priori. In Section 5, the issue of ensuring certain formats is dealt with via normal forms.

Conjunctions and Disjunctions. In order to express more complex rules than the ones given above, entries may specify conjunctions and disjunctions of *literal*s. For instance,

```
8 3 89 -8 -15 0
```

defines an object with ID `89` as the *conjunction* of the default negations of the objects with IDs `8` and `15`. Similarly,

```
9 3 98 8 15 0
```

defines a *disjunction* of the objects with IDs `8` and `15`. Assuming that ID `8` stands for atom `p(a,1)` and `15` for `-p(a,1)`, the entries

```
7 2 78 89 0
6 2 67 98 0
```

describe the following integrity constraint and disjunctive fact:

```
:- not p(a,1), not -p(a,1).
p(a,1) | -p(a,1).
```

Finally, note that the normal forms specified in Section 5 will restrict conjunctions to occur in bodies of rules and disjunctions to being used in rule heads.

Default Negation. Programs in "canonical form" [25, 28] permit double (default) negation of atoms. Rather than permitting multiple occurrences of "−" at the beginning of a *literal* (which would make *literal*s and *integer*s syntactically different), we introduce entries defining default negation objects for representing such nested expressions. This allows us to express a "choice"

```
p(a,1) :- not not p(a,1).
```

in terms of the following entries:

```
a 2 44 -8  0
5 3 45  8 44 0 .
```

Observe that the object defined by the first entry stands for the *default negation* of *literal* `-8` and, thus, for the double negation of the object with ID `8`, viz., of atom `p(a,1)`. Finally, note that, under answer set semantics, rules using an atom and those using its double negation are not necessarily equivalent [28], so that double negation constitutes a proper syntactic feature. The normal form in Section 5.8 will thus permit *default negation* objects for expressing double negation of atoms.

Cardinality and Weight Constraints. Cardinality and weight constraints [43, 44] permit expressing conditions on sets of literals, that is, true literals can be counted or their weights can be summed up in order to compare the result against a lower and an upper bound. Letting *object ID*s `1`, `2`, `3`, and `4` stand for atoms `a`, `b`, `c`, and `d`, we can represent the expressions

```
2{a, b, not c, not d}3
-1[a=-2, b=1, not c=3, not d=-4]2
```

in *ASPils* as follows:

```
b  7 5  2 3 1 2 -3 -4  0
c 11 6 -1 2 1 2 -3 -4 -2 1 3 -4 0 .
```

Here, the *object ID*s `5` and `6` of the *cardinality* and *weight constraint*, respectively, are immediately followed by their lower and upper bounds. Afterwards, the *literal*s are provided, and for weight constraint `6`, also a list of *weight*s, exactly one *weight* per *literal*. The primary constituents of a logic program, viz., *rule*s, *fact*s, and *integrity constraint*s, can incorporate *cardinality* and *weight constraint*s just like *atom*s, simply by referencing their IDs directly or indirectly through literals. Observe that, in general, we allow for upper bounds as well as for negative weights, both of which can occur in the input language, but not in the output language, of *lparse*. In fact, *lparse* performs a number of transformations and introduces new atoms to remove them. Some of the normal forms in Section 5 impose similar restrictions, thus, specifying *lparse*-like fragments of *ASPils*. In such fragments, only *trivial upper bounds* are permitted, obtained by summing up all (positive) weights, where weight `1` is used for the literals of cardinality constraints.

Weighted Literals. *Weighted literal*s are auxiliary concepts, devised for the use with aggregates and optimization statements (see below), thus, establishing a uniform way of referencing objects (simple or complex ones) that evaluate to numbers. For instance, we may associate *weight*s to *literal*s, as in the above weight constraint, via the following entries:

```
d 3 11  1 -2 0
d 3 12  2  1 0
d 3 13 -3  3 0
d 3 14 -4 -4 0 .
```

Aggregates. We adopt the aggregates supported by *dlv* [6], allowing for five operations, viz., *count*, *sum*, *max*, *min*, and *times*. While *count* applies to Boolean operands, that is, to *literal*s, the other four aggregates operate on numerical values, thus, they require *object ID*s rather than *literal*s as parameters. For instance, *count* is used to count satisfied literals whereas *sum* yields the sum of weights associated with satisfied literals. Reusing atoms `a`, `b`, `c`, and `d` as well as weighted literals as specified above, we may define a *count* and a *sum* aggregate as follows:

```
e 5 21  1  2 -3 -4 0
f 5 22 11 12 13 14 0 .
```

Observe that *count* applies to (possibly negative) *literal*s, while *dlv*'s aggregates are restricted to atoms. As such a restriction does not significantly simplify dealing with aggregates, we do not adopt it here, and the *weighted literal*s used by *sum* may also apply to negative *literal*s (as it is the case for the objects with IDs `13` and `14`). However, in order to reasonably apply an aggregate, its operands must have appropriate types, being an issue to the normal forms that will be addressed and collected in Section 5.

Operators. Arithmetic comparison operators can be applied to *weighted literal*s and *aggregate*s in order to retrieve Boolean values from them. Thus, *operator*s can be referenced by *rule*s, *fact*s, and *integrity constraint*s in the same way as *atom*s. We provide two kinds of operators: (binary) *operator*s of type in-between `13` and `17` can be used to compare the numerical values

of two *object*s with one another, while *unary operator*s of type in-between `18` and `1c` allow for comparing an *object*'s numerical value to an *integer*. Both kinds of *operator*s support the following comparison operations: *eq* ("equal"), *leq* ("less or equal"), *lt* ("less than"), *geq* ("greater or equal"), and *gt* ("greater than"). For instance, the following entry describes the application of the (binary) *operator eq* to the aggregates with IDs `21` and `22` as defined above:

```
13 3 31 21 22 0 .
```

An application of *unary operator leq* to the aggregate with ID `22` and integer `0` can be specified as follows:

```
19 3 91 22 0 0 .
```

Finally, note that current ASP solvers do not support (binary) *operator*s of type in-between `13` and `17`, hence, they will not be permitted by the normal forms in Section 5.

Optimization. Amongst the most common optimization techniques in ASP are "minimize statements" [43] supported by *smodels* and "weak constraints" [26] supported by *dlv*. For accommodating them, we introduce an *optimize* object whose underlying *strategy* is minimization of (arbitrarily many) objective functions of distinct priorities, the priorities forming a strict total order. Other strategies than lexicographic ordering of objective functions, e.g., Pareto optimality, would also be possible but are currently not in use, so we do not (yet) consider them. In what follows, we detail how "minimize statements" and "weak constraints" can be represented in *ASPils*.

Minimize Statements of *smodels* [43]: We start with an example. Assume that an input program provided to *lparse* contains two minimize statements (in order):

```
minimize {not a, b, c}.
minimize [a=4, not b=3, c=2].
```

The first statement expresses that a minimum number of its literals should be true, while the second one is about minimizing the sum of weights of true literals. The corresponding objective functions are expressed by a *count* and a *sum* aggregate (over *weighted literal*s), respectively, where we assume IDs `1`, `2`, and `3` for atoms `a`, `b`, and `c`:

```
d 3 11  1  4  0
d 3 12 -2  3  0
d 3 13  3  2  0
e 4 21 -1  2  3 0
f 4 22 11 12 13 0 .
```

Note that the *count* aggregate with ID `21` gives the objective function minimized by the first statement over literals, and the *sum* aggregate with ID `22` takes the weights provided in the second minimize statement into consideration. We are now ready to define a single *optimize* object that incorporates both aggregates:

```
1d 4 43 1e 22 21 0 .
```

The type of this object is `1d`, `4` the number of parameters, and `43` the ID. Furthermore, `1e` specifies the lexicographic optimization strategy, which is the only *strategy* included in the first version of *ASPils*. Finally, minimizing the numerical value of the *sum* aggregate with ID `22` takes higher priority than minimizing the value of the *count* aggregate with ID `21`. This priority, reverse to the order of minimize statements in the input, is indeed applied by *smodels*.

In what follows, we provide a general scheme of how to represent multiple minimize statements in *ASPils*. For uniformity, we assume that all minimize statements include multiple literals along with weights. (Without explicit weights, *count* may be used instead of *sum*, and for a minimize statement over a single literal, a *weighted literal* would already express its objective function.) Consider m minimize statements in reverse order of priority:

`minimize` $[l_{1_m} = w_{1_m}, \ldots, l_{n_m} = w_{n_m}]$. $\ldots$ `minimize` $[l_{1_1} = w_{1_1}, \ldots, l_{n_1} = w_{n_1}]$.

Note that $l_{1_1}, \ldots, l_{n_m}$ can be represented as *literal*s in *ASPils*. We keep notation l_{i_j} to refer to these literals for defining *weighted literal*s to associate each l_{i_j} with w_{i_j}:

`d 3` x_{1_1} l_{1_1} w_{1_1} `0`
$\vdots$
`d 3` x_{n_1} l_{n_1} w_{n_1} `0`
$\vdots$
`d 3` x_{1_m} l_{1_m} w_{1_m} `0`
$\vdots$
`d 3` x_{n_m} l_{n_m} w_{n_m} `0 .`

Note that the above x_{i_j}s are placeholders for the *object ID*s of the defined *weighted literal*s. We then proceed by defining *sum* aggregates for individual minimize statements:

`f` n_1`+1` s_1 x_{1_1} $\ldots$ x_{n_1} `0`
$\vdots$
`f` n_m`+1` s_m x_{1_m} $\ldots$ x_{n_m} `0 .`

The s_js are again placeholders for the *object ID*s of the defined *sum* aggregates. We use these object IDs to define the final *optimize* object:

`1d` m`+2` o `1e` s_1 $\ldots$ s_m `0 .`

Again, *o* stands for the *object ID* of the defined *optimize* object, and recall that `1e` specifies the lexicographic optimization *strategy*. While *lparse* and *smodels* derive the (reverse) priorities of minimize statements implicitly from the order in the input, in the *ASPils* representation, priorities are explicit because the objective functions to be minimized are combined within an *optimize* object.

Weak Constraints of *dlv* [26]: We again start with an example. Consider an input program containing the following weak constraints:

```
:~ a, not b. [1:2]
:~ not c. [1:1]
:~ b, c. [2:1]
```

Note that the numbers in brackets describe weights and levels, respectively. As level `2` of the first weak constraint is greater than `1`, the first weak constraint is of higher priority than the second and the third one. Among the last two weak constraints, the priority of the third one is greater simply because its weight `2` is greater than `1`. In order to express the non-singleton bodies of weak constraints, we define *conjunction*s as follows:

```
8 3 81 1 -2 0
8 3 83 2  3 0 .
```

The *conjunction* with ID `81` stands for the body of the first weak constraint, and the one with ID `83` for the body of the third weak constraint. We can now proceed by defining *weighted literal*s, one per weak constraint:

```
d 3 11 81 1 0
d 3 21 -3 1 0
d 3 22 83 2 0 .
```

Observe that the *weight* of each weak constraint is the weight given in the input. Finally, we use a *sum* aggregate for multiple weak constraints at the same level and define an *optimize* object as follows:

```
f  3 24 21 22 0
1d 4 35 1e 11 24 0 .
```

The last entry expresses that we minimize weights starting with the weak constraint at level `2` and, secondarily, for the weak constraints at level `1`.

We now provide a general scheme of how to represent multiple weak constraints of distinct levels in *ASPils*. Consider the following weak constraints ordered by their levels l_j, where we assume $l_i > l_j$ if $i < j$ in order to respect level priorities:

$$\begin{array}{llll} \texttt{:}\sim\ body_{1_1}.\ [w_{1_1} : l_1] & \ldots & \texttt{:}\sim\ body_{n_1}.\ [w_{n_1} : l_1] \\ & \vdots & \\ \texttt{:}\sim\ body_{1_m}.\ [w_{1_m} : l_m] & \ldots & \texttt{:}\sim\ body_{n_m}.\ [w_{n_m} : l_m]\,. \end{array}$$

In *ASPils*, the bodies $body_{1_1}, \ldots, body_{n_m}$ of weak constraints can be defined by *literal*s either over *atom*s (for singleton bodies) or over *conjunction*s, which can be defined as usual. Thus, we keep notation $body_{i_j}$ in the following *weighted literal*s:

d 3 x_{1_1} $body_{1_1}$ w_{1_1} 0

$\vdots$

d 3 x_{n_1} $body_{n_1}$ w_{n_1} 0

$\vdots$

d 3 x_{1_m} $body_{1_m}$ w_{1_m} 0

$\vdots$

d 3 x_{n_m} $body_{n_m}$ w_{n_m} 0 .

Except for the use of $body_{i_j}$ instead of l_{i_j}, the *weighted literal*s defined above are similar to those used in the representation of minimize statements. In fact, as $body_{i_j}$ and l_{i_j} are both *literal*s, it makes no difference in *ASPils* whether they refer to *atom*s or to *conjunction*s. The next step of defining level-wise *sum* aggregates and a single *optimize* object is similar to minimize statements:

f n_1+1 s_1 x_{1_1} ... x_{n_1} 0

$\vdots$

f n_m+1 s_m x_{1_m} ... x_{n_m} 0

1d m+2 o 1e s_1 ... s_m 0 .

The given embeddings in *ASPils* indicate that minimize statements and weak constraints are handled likewise, using *weighted literal*s, *sum* aggregates (sometimes, simpler constructs are sufficient), and an *optimize* object.

5 Normal Forms

This section describes seven normal forms corresponding to different language fragments handled by existing ASP solvers. The normal forms stand in a hierarchy, as illustrated in Figure 2. Each of the normal forms is identified via a corresponding number, given in parentheses in Figure 2, to be provided within the *header* (cf. Appendix A) of a problem description in *ASPils*.

The marks in Table 1 (see page 21) indicate which entry types (represented by columns) are admissible in particular normal forms (indicated by letters denoting rows). Consult Table 3 for the abbreviations used. Note that (binary) *operator*s of types in-between 13 and 17 are not supported by any of the normal forms, that is, they cannot occur in problem descriptions.

Table 2 summarizes requirements on the types of objects (in columns) permitted as parameters in entries of certain types (in rows), that is, the cells contain the normal forms in which an entry of the row type can refer to an entry of the column type by using its *object ID* as a parameter, either directly or through a negative *literal*. For instance, the cell in row 6, column 9, indicates that a *disjunction* object may occur as parameter of a *fact* in normal forms other than "Simple" and "SModels." Note that an *atom* may refer to another *atom* only through option *classical negation*, otherwise, row 4 would be empty. Furthermore, *comment*s of type 3 are not listed in the second table as they do not have IDs and can themselves not refer to any object. One can check that the requirements on reference relationships among objects of particular types exclude nesting of complex types, e.g., a *conjunction* may neither directly nor indirectly be defined in terms of another *conjunction*. In this way, a normal form imposes a certain structure of ground

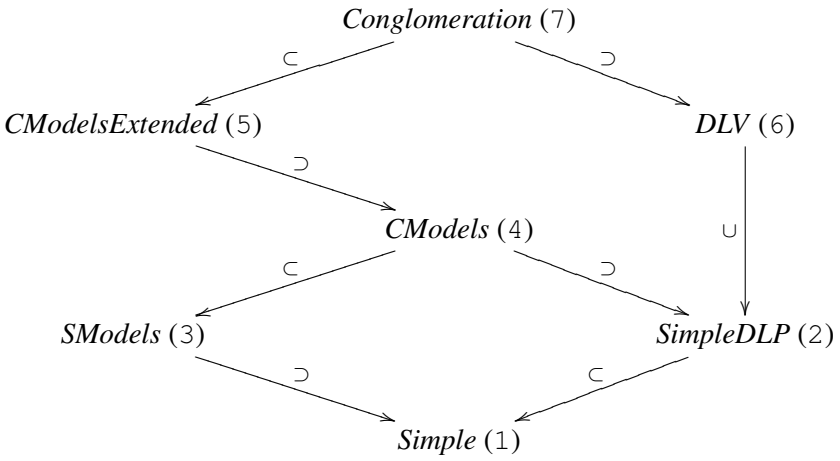

Figure 2: Normal Form Hierarchy

logic programs, which become thereafter easier to handle by solvers. Finally, note that further restrictions are given in footnotes, they will be explained below along with the normal forms subject to restrictions in question. Before we describe the particular normal forms, we start with global requirements valid for all of them.

5.1 Global Requirements

To ease parsing and the semantic recognition of problem descriptions in *ASPils*, we introduce some requirements to be met by all normal forms. Accordingly, the following global requirements apply automatically and will thus not be repeated for particular normal forms:

- Every problem description in *ASPils* is a *program*, starting with a *header* and ending with an occurrence of *object eof*.
- In a *header*, *object eof*, and all *entries*, the parameter immediately after the type must correctly specify the number of further parameters occurring before delimiter `0`.
- We say that an *object* is "defined" by an *entry* if its *object ID* occurs immediately after the number of further parameters. (Note that *comments* do not define any object as they do not have a parameter *object ID*.) Every *object* occurring in a problem description of *ASPils* may be defined by at most one *entry*, that is, every defined *object* has a unique *object ID*.
- We say that an *object* is "referenced" by an *entry* if its *object ID* occurs directly or as the absolute value of a *literal* in a parameter distinct from the defined object's ID. Every *object* referenced by an *entry* must have been defined earlier by some distinct *entry* of the problem description, that is, every referenced *object* is (non-recursively) defined.

	2	3	4	5–7	8	9	a	b	c	d	e–f	10–12	18–1c	1d
A	•	•	•	•	•									
B	•	•	•	•	•	•								
C	•	•	•	•	•			•[6]	•[6,7]	•[7]	•			•
D	•	•	•	•	•	•		•[6]	•[6,7]	•[7]	•			•
E	•	•	•	•	•	•		•	•	•	•			•
F	•	•	•	•	•	•				•[7]	•	•	•	•
G	•	•	•	•	•	•	•	•	•	•	•	•	•	•

Table 1: Summary of admissible objects

	2	4	5–7	8	9	a	b	c	d	e–f	10–12	18–1c	1d
2	A–G	A–G	A–G	A–G	B,D–G	G	C–E,G	C–E,G	C–G	C–G	F–G	F–G	C–G
4		A–G											
5		A–G		[A–G][8,9]	[B,D–G][10]	G[8,9]	C–D[11],E,G	C–D[8],E,G				F[8],G	
6		A–G			B,D–G		C–D[11],E,G	E,G				G	
7		A–G		[A–G][9]		G[9]	C–E,G	C–E,G				F–G	
8		A–G				G[9]	C–E,G	C–E,G				F–G	
9		[B,D–G][9]											
a		G[12]											
b		C–E,G				G[9]							
c		C–E,G				G[9]							
d		C–G		[F–G][9,13]		G[9]							
e		C–G		[F–G][9,13]		G[9]							
f									[C–G][9]				
10–12									[F–G][9]				
18–1c									[F–G][9]	[F–G][9]	[F–G][9]		
1d									[C–G][9]	[C–G][9]			

Table 2: Overview of admissible reference relationships

2 *Meta-Object*	4 *Atom*	9 *Disjunction*	e *Count Aggregate*	18 *EQ Operator*
3 *Comment*	5 *Rule*	a *Default Negation*	f *Sum Aggregate*	19 *LEQ Operator*
1d *Optimize Object*	6 *Fact*	b *Cardinality Constraint*	10 *Max Aggregate*	1a *LT Operator*
	7 *Integrity Constraint*	c *Weight Constraint*	11 *Min Aggregate*	1b *GEQ Operator*
	8 *Conjunction*	d *Weighted Literal*	12 *Times Aggregate*	1c *GT Operator*
A Simple	C SModels	D CModels	F DLV	G Conglomeration
B SimpleDLP		E CModelsExtended		

Table 3: Abbreviations used in Table 1 and Table 2

- Objects need not be defined in the strict order of their IDs (as long as they are defined before they get referenced). Furthermore, object IDs do not have to be consecutive, that is, gaps are allowed.

[6] Only with trivial upper bound.
[7] Only with non-negative weights.
[8] Only referable as rule body.
[9] Only referable as positive literal.
[10] Only referable as rule head.
[11] Only with lower bound 0 when referenced as head.
[12] Only referable as negative literal.
[13] Only referable if not in the scope of any operator of type in-between 18 and 1c.

- The *literal* standing for the head of a *rule* or *fact* must be positive, in other words, heads must not be (default) negated. The reason for this design decision is that default negation of a head can be eliminated by using an *integrity constraint* with the head in its body, thus, this requirement reduces ambiguity in the syntactic representation of integrity constraints.
- A problem description of *ASPils* may define at most one *optimize* object of type `1d`.

We below explain and illustrate the normal forms shown in Figure 2 on examples. A detailed list of the restrictions they impose is collected in Tables 1 and 2 (cf. Table 3 for abbreviations).

5.2 Normal Form *Simple*

This *normal form* corresponds to the input language used in the *SCore* category of the ASP system competition [16]. It allows for representing ground normal logic programs without any extended constructs (like aggregates, etc.). For example, consider the following input program:

```
a :- not b.
b :- not a.
:- b.
c :- d, not b.
d.
#hide a.
```

This program can be translated into the following *ASPils* representation:[14]

```
3 1 "header, language version =: 1, normal form =: 1" 0
1 3 1 1 0 0
3 1 "a =: 1 and hidden, b =: 2, c =: 3, d =: 4" 0
4 3 1 a 1 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "a :- not b. =: 5" 0
5 3 5 1 -2 0
3 1 "b :- not a. =: 6" 0
5 3 6 2 -1 0
3 1 ":- b. =: 7" 0
7 2 7 2 0
3 1 "(d, not b) =: 8" 0
8 3 8 4 -2 0
3 1 "c :- d, not b. =: 9" 0
5 3 9 3 8 0
3 1 "d. =: 10" 0
6 2 10 4 0
```

[14] We indicate the meanings of *objects* in comments preceding their definitions.

```
3 1 "end of file" 0
0 0 0 .
```

Note that the above representation is not unique, for instance, we could have assigned different *object ID*s or changed the order of entries (provided that referenced *object*s are already defined).

5.3 Normal Form *SimpleDLP*

This *normal form*, corresponding to the input language used in the *SCore*$^{\vee}$ category of the ASP system competition [16], extends normal form "Simple" by allowing *disjunction*s over *atom*s to occur in heads of *rule*s and *fact*s. For instance, consider the following disjunctive program:

```
a | b.
b | c | d :- a, not d.
```

This program can be represented in *ASPils* as follows:

```
3 1 "header, language version =: 1, normal form =: 2" 0
1 3 1 2 0 0
3 1 "a =: 1, b =: 2, c =: 3, d =: 4" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "(a | b) =: 5" 0
9 3 5 1 2 0
3 1 "a | b. =: 6" 0
6 2 6 5 0
3 1 "(b | c | d) =: 7" 0
9 4 7 2 3 4 0
3 1 "(a, not d) =: 8" 0
8 3 8 1 -4 0
3 1 "b | c | d :- a, not d. =: 9" 0
5 3 9 7 8 0
3 1 "end of file" 0
0 0 0 .
```

As in the previous subsection, this representation in *ASPils* is not unique.

5.4 Normal Form *SModels*

This *normal form* is inspired by the input language of the *smodels* solver [43], that is, it extends "Simple" by *cardinality* and *weight constraint*s as well as *optimize* objects (cf. tables on page 21). Note that the *weight*s used in *weight constraint*s and *weighted literal*s have to be non-negative. Furthermore, the upper bounds of *cardinality* and *weight constraint*s must be trivial, that is, they cannot be smaller than the number of literals or the sum of weights, respectively,

in a constraint. If a *cardinality constraint* occurs as the head of a *rule* or *fact*, its lower bound must also be trivial, viz., it must be `0`, while *weight constraint*s are not permitted as heads. Finally, note that *weighted literal*s as well as *count* and *sum* aggregates may only be used in combination with an *optimize* object, but not as a part of a *rule*, a *fact*, or an *integrity constraint*. For illustration, we consider an input program as follows:

```
{a, b}.
c :- a, not b.
:- 3[a=2, b=1, not c=2].
minimize [not a=1, not b=2, c=2].
```

The following is a possible representation of this program in *ASPils*:

```
3 1 "header, language version =: 1, normal form =: 3" 0
1 3 1 3 0 0
3 1 "a =: 1, b =: 2, c=: 3" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
3 1 "{a, b} =: 4" 0
b 5 4 0 2 1 2 0
3 1 "{a, b}. =: 5" 0
6 2 5 4 0
3 1 "(a, not b) =: 6" 0
8 3 6 1 -2 0
3 1 "c :- a, not b. =: 7" 0
5 3 7 3 6 0
3 1 "3[a=2, b=1, not c=2] =: 8" 0
c 9 8 3 5 1 2 -3 2 1 2 0
3 1 ":- 3[a=2, b=1, not c=2]. =: 9" 0
7 2 9 8 0
3 1 "(not a=1) =: 10, (not b=2) =: 11, (c=2) =: 12" 0
d 3 10 -1 1 0
d 3 11 -2 2 0
d 3 12  3 2 0
3 1 "sum[not a=1, not b=2, c=2] =: 13" 0
f 4 13 10 11 12 0
3 1 "minimize [not a=1, not b=2, c=2]. =: 14" 0
1d 3 14 1e 13 0
3 1 "end of file" 0
0 0 0 .
```

5.5 Normal Form *CModels*

This *normal form* is closely related to the input language of solver *cmodels* [20, 27], basically, augmenting "SModels" normal form with *disjunction*s in heads of *rule*s and *fact*s.[15]

5.6 Normal Form *CModelsExtended*

This *normal form* is derived from "CModels" by dropping some of its restrictions. Non-trivial upper bounds are permitted for *cardinality* and *weight constraint*s. Furthermore, both of them can occur with non-trivial bounds as heads of *rule*s and *fact*s. Finally, negative *weight*s can be used within *weight constraint*s and *weighted literal*s being subject to *optimize* objects (cf. Table 2). The following example program makes use of these additional features:

```
0[a=1, b=-1]0 :- 0[c=-1, d=1]0.
0[c=1, d=-1]0 :- 0[a=-1, b=1]0.
minimize [not a=-1, not b=2, c=-2, d=1].
```

This program can be represented in *ASPils* as follows:

```
3 1 "header, language version =: 1, normal form =: 5" 0
1 3 1 5 0 0
3 1 "a =: 1, b =: 2, c =: 3, d =: 4" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "0[a=1, b=-1]0 =: 5" 0
c 7 5 0 0 1 2 1 -1 0
3 1 "0[c=-1, d=1]0 =: 6" 0
c 7 6 0 0 3 4 -1 1 0
3 1 "0[a=1, b=-1]0 :- 0[c=-1, d=1]0. =: 7" 0
5 3 7 5 6 0
3 1 "0[c=1, d=-1]0 =: 8" 0
c 7 8 0 0 3 4 1 -1 0
3 1 "0[a=-1, b=1]0 =: 9" 0
c 7 9 0 0 1 2 -1 1 0
3 1 "0[c=1, d=-1]0 :- 0[a=-1, b=1]0. =: 10" 0
5 3 10 8 9 0
3 1 "(not a=-1) =: 11, (not b=2) =: 12,
     (c=-2) =: 13, (d=1) =: 14" 0
d 3 11 -1 -1 0
d 3 12 -2  2 0
d 3 13  3 -2 0
d 3 14  4  1 0
```

[15] While *cmodels* does not process minimize statements, they can be expressed in "CModels" via *optimize* objects.

```
3 1 "sum[not a=-1, not b=2, c=-2, d=1] =: 15" 0
f 5 15 11 12 13 14 0
3 1 "minimize [not a=-1, not b=2, c=-2, d=1]. =: 16" 0
1d 3 16 1e 15 0
3 1 "end of file" 0
0 0 0 .
```

5.7 Normal Form *DLV*

This *normal form* is inspired by the input language of *dlv* [6, 26]. Hence, it allows for *disjunction*s over *atom*s in heads of *rule*s and *fact*s. Furthermore, aggregates may be used in bodies, under the proviso that all referenced *weighted literal*s have non-negative *weight*s.[16] As *dlv* does not deal with cardinality and weight constraints of *smodels* [43, 44], we exclude *cardinality* and *weight constraint*s from "DLV" normal form. (However, *cardinality* and *weight constraint*s can equivalently be expressed in terms of *count* and *sum* aggregates, respectively.) Finally, weak constraints [26] can be represented using *optimize* objects. Note that the restrictions in Table 2 make sure that a *weighted literal* referencing a *conjunction* can only be used within an *optimize* object, but not by aggregates in bodies, which excludes the possibility of *conjunction*s to nest. For illustration, consider an input program as follows:

```
a | b | c.
d :- sum[a=1, b=1, c=2] >= 2.
:~ d, not b. [1:2]
:~ a. [2:1]
:~ not c. [1:1]
```

The following is a possible representation of this program in *ASPils*:

```
3 1 "header, language version =: 1, normal form =: 6" 0
1 3 1 6 0 0
3 1 "a =: 1, b =: 2, c =: 3, d =: 4" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "(a | b | c) =: 5" 0
9 4 5 1 2 3 0
3 1 "a | b | c. =: 6" 0
6 2 6 5 0
3 1 "(a=1) =: 7, (b=1) =: 8, (c=2) =: 9" 0
d 3 7 1 1 0
d 3 8 2 1 0
d 3 9 3 2 0
3 1 "sum[a=1, b=1, c=2] =: 10" 0
```

[16] Moreover, the *dlv* solver requires logic programs to be "aggregate-stratified" [6], which is neglected herein.

```
f 4 10 7 8 9 0
3 1 "(sum[a=1, b=1, c=2] >= 2) =: 11" 0
1b 3 11 10 2 0
3 1 "d :- sum[a=1, b=1, c=2] >= 2. =: 12" 0
5 3 12 4 11 0
3 1 "(d, not b) =: 13" 0
8 3 13 4 -2 0
3 1 "((d, not b)=1) =: 14, (a=2) =: 15, (not c=1) =: 16" 0
d 3 14 13 1 0
d 3 15  1 2 0
d 3 16 -3 1 0
3 1 "sum[a=2, not c=1] =: 17" 0
f 3 17 15 16 0
3 1 "minimize [(d, not b)=1].
     minimize [a=2, not c=1]. =: 18" 0
1d 4 18 1e 14 17 0
3 1 "end of file" 0
0 0 0 .
```

5.8 Normal Form *Conglomeration*

This *normal form* is the most general one defined here. It results from "CModelsExtended" and "DLV" by dropping some restrictions of the latter, that is, *unary operator*s and *aggregate*s may occur in the heads of *rule*s and *fact*s, and negative *weight*s are allowed within *weighted literal*s. Furthermore, we include *default negation* objects on negative *literal*s over *atom*s in order to account for double negation. Though double negation is a syntactical feature that increases neither computational complexity nor technical difficulties of ASP solving, somewhat astonishingly, it is currently not supported by any ASP solver nor by accompanying grounders. This is why *default negation* objects were not permitted in previous normal forms. Their restriction to negative *literal*s over *atom*s excludes nesting and makes the representation of double negation unambiguous. The following program, comprising a single rule, uses the additional features:

```
sum[a=1, b=1, c=1, d=-2] == 0 :-
         2[a=-1, not b=2, not c=3]3, not not d.
```

This program can be represented in *ASPils* as follows:

```
3 1 "header, language version =: 1, normal form =: 7" 0
1 3 1 7 0 0
3 1 "a =: 1, b =: 2, c =: 3, d =: 4" 0
4 2 1 a 0
4 2 2 b 0
4 2 3 c 0
4 2 4 d 0
3 1 "(a=1) =: 5, (b=1) =: 6, (c=1) =: 7, (d=-2) =: 8" 0
```

```
d 3 5 1  1 0
d 3 6 2  1 0
d 3 7 3  1 0
d 3 8 4 -2 0
3 1 "sum[a=1, b=1, c=1, d=-2] =: 9" 0
f 5 9 5 6 7 8 0
3 1 "(sum[a=1, b=1, c=1, d=-2] == 0) =: 10" 0
18 3 10 9 0 0
3 1 "2[a=-1, not b=2, not c=3]3 =: 11" 0
c 9 11 2 3 1 -2 -3 -1 2 3 0
3 1 "(not not d) =: 12" 0
a 2 12 -4 0
3 1 "(2[a=-1, not b=2, not c=3]3, not not d) =: 13" 0
8 3 13 11 12 0
3 1 "sum[a=1, b=1, c=1, d=-2] == 0 :-
       2[a=-1, not b=2, not c=3]3, not not d. =: 14" 0
5 3 14 10 13 0
3 1 "end of file" 0
0 0 0 .
```

6 Discussion and Outlook

In this document, we have specified language version `1` of *ASPils*, which stands for "*ASP intermediate language standard*." The primary motivation for this work is making a step towards a standard input language for ASP solvers to be generated by grounders, for which we propose *ASPils*. The major design goals of *ASPils* are generality, by supporting language constructs processed by existing ASP grounders and solvers, and extensibility, by using an object-based approach and including version information. For solvers, parsing a problem description in *ASPils* should still be reasonably simple, thus, *ASPils* defines a numerical format not intended to be manually written by users. However, *ASPils* also provides means to specify symbolic information, enabling the reconstruction of a human-readable format. Beyond that, via comments and meta-information, arbitrary contents can be included in a problem description without disturbing solvers. Thus, the current proposal of *ASPils* is an appropriate response to the recommendations presented in [22] as regards extensibility and support for comments as well as symbolic information. The specification of a module architecture for problem representations in *ASPils* is still missing from this document, but atom options for identifying input atoms and local atoms are already provided in the first version of the format. However, the outcome of *joining* several *ASPils* specifications into a single specification remains to be defined.

As a proof of concept, we are currently working on a new version of grounder *gringo* [17] able to output *ASPils* format and also on an *ASPils* front-end for solver *clasp* [15]. In the course of this, we take advantage of the generic design of *ASPils* allowing us to preserve the structure of ground logic programs. For instance, *gringo* can output cardinality and weight constraints specifying both a lower and a non-trivial upper bound, and such constraints can occur both in

the bodies and in the heads of rules. In contrast, in *lparse*'s output format [44], upper bounds (and in rule heads, also lower bounds) have to be compiled away, introducing additional atoms and rules. Such structure-degrading transformations are performed by *lparse* in order to match the problem representation with the internal data structures of *smodels* [43], and in the past, tools [30] were particularly developed to undo such transformations. As regards grounding, we think that the two tasks of a grounder are, first, substituting constants for variables in an input program and, second, presenting the grounding result to a solver in some basic format that is easy to parse. Beyond these two tasks, a grounder should keep the input program intact in order to be solver-independent and to abolish the need of applying structure-restoring tools. In particular, introducing additional atoms during the grounding phase ought to be avoided, as it is very likely to spoil the desired equivalence between the input program and the result of grounding.

In long-term, we hope that our proposal of an intermediate language standard leads to the establishment of a common input format for ASP solvers, comparable to the role of DIMACS format [7] in SAT. On the one hand, it would make ASP more user-friendly if solvers could be interchanged without redoing problem encodings, given that the two main input languages, the one of *dlv* [26] and the one of *lparse* [44], are incompatible with each other. A common intermediate language would enhance the interoperability of other auxiliary tools as well. Similarly, the assessment of ASP solvers would be greatly facilitated, for instance, in future ASP solver competitions. On the other hand, the non-availability of a standardized intermediate language (as *dlv* does not supply an intermediate format and *lparse*'s output language is not standardized) makes ASP solvers and related tools satellites of particular grounders, addicted to their capabilities and supported language fragments. We think that the establishment of an extensible intermediate language standard, not dictated by the capabilities of grounders, might motivate future works on knowledge representation for applications, inventing new language constructs when they are useful and then integrating them into the standard. At the moment, incorporating new language features would mean hacking one of the few available grounding tools, making the broad acceptance and usage of the feature rather unlikely. However, the establishment of an intermediate language standard must be a community effort, requiring a representative standardization committee and developers motivated to implement the standard in their tools. In view of these requirements, our proposal of *ASPils* can serve as a starting point for future discussions within the community as well as first implementations of a real shared format.

Let us note that the establishment of *ASPils* or a comparable intermediate language standard can only be a small step in making ASP tools more general, more interoperable, and thus more user-friendly. In fact, tools are needed to perform various useful tasks on problem descriptions in the new language, similar to what the Helsinki collection [21] of tools offers for *lparse*'s output format. Let us give some examples. For the use in ASP system competitions, solver inputs must contain neither meta-information nor comments, thus, a (trivial to develop) tool to delete such information would be needed. For benchmarking, a tool like *shuffle* is desirable, in particular, considering that shuffling *ASPils* sentences requires more care than needed with *lparse*'s output format as the order among object definitions and references has to be maintained. If a grounder generates *ASPils* output on-the-fly, it is hard to predict the most restrictive normal form sufficient for the input program, hence, a post processor calculating this simplest normal form might be useful. As the last example given here, a tool like *lplist* should be made available for reconstructing a symbolic representation from the intermediate format. Finally, we note that *ASPils*

or any other intermediate language cannot establish compatibility among the input languages of grounders or integrated ASP systems (such as *dlv*). Furthermore, incremental [14] or even systems dealing with non-ground input programs are currently out of the scope of our proposal. However, we think that the relatively simple concept of an intermediate language provides a good basis for standardization efforts, whereas more sophisticated aspects of intermediate languages could be addressed in the future.

Acknowledgments. We are grateful to Marcello Balduccini, Martin Brain, Maarten Mariën, and Axel Polleres for fruitful discussions and valuable ideas that contributed to this work. Moreover, we would like to thank the anonymous reviewers of a short version of this document [13] whose comments helped us to improve the presentation.

The second author was supported by the project "*Methods for Constructing and Solving Large Constraint Models*" funded by the Academy of Finland (grant #122399).

References

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[2] C. Baral, G. Brewka, and J. Schlipf, editors. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

[3] C. Baral, G. Greco, N. Leone, and G. Terracina, editors. *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[6] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 847–852. Morgan Kaufmann Publishers, 2003.

[7] Satisfiability suggested format. URL[17], 1993.

[8] Extended Backus-Naur form. URL[18], 1996.

[17] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/`

[18] `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`

[9] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Proceedings of the Eigth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 2005.

[10] E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.

[11] F. Fages. Consistency of Clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[12] P. Ferraris. Answer sets for propositional theories. In Baral et al. [3], pages 119–131.

[13] M. Gebser, T. Janhunen, M. Ostrowski, T. Schaub, and S. Thiele. A versatile intermediate language for answer set programming. In M. Pagnucco and M. Thielscher, editors, *Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08)*, Technical Report UNSW-CSE-TR-0819, pages 150–159. University of New South Wales, September 2008.

[14] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

[15] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In Baral et al. [2], pages 260–265.

[16] M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub, and M. Truszczyński. The first answer set programming system competition. In Baral et al. [2], pages 3–17.

[17] M. Gebser, T. Schaub, and S. Thiele. GrinGo: A new grounder for answer set programming. In Baral et al. [2], pages 266–271.

[18] M. Gelfond and N. Leone. Logic programming and knowledge representation — the A-Prolog perspective. *Artificial Intelligence*, 138(1-2):3–38, 2002.

[19] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[20] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[21] T. Janhunen. ASPTOOLS. URL[19].

[22] T. Janhunen. Intermediate languages of ASP systems and tools. In M. De Vos and T. Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, number CSBU-2007-05 in Department of Computer Science, University of Bath, Technical Report Series, pages 12–25, 2007. ISSN 1740-9497.

[19] `http://www.tcs.hut.fi/Software/asptools/`

[23] T. Junttila. Constrained Boolean circuits – a file format and some tools. URL[20].

[24] T. Junttila and I. Niemelä. Towards an efficient tableau method for Boolean circuit satisfiability checking. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv, and P. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic (CL'00)*, volume 1861 of *Lecture Notes in Computer Science*, pages 553–567. Springer-Verlag, 2000.

[25] J. Lee. A model-theoretic counterpart of loop formulas. In L. Kaelbling and A. Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 503–508. Professional Book Center, 2005.

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

[27] Y. Lierler. cmodels - SAT-based disjunctive answer set solver. In Baral et al. [3], pages 447–451.

[28] V. Lifschitz, L. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):369–389, 1999.

[29] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[30] L. Liu and M. Truszczyński. Pbmodels - software to compute stable models by pseudo-boolean solvers. In Baral et al. [3], pages 410–415.

[31] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[32] J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[33] D. Mitchell. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science*, 85:112–133, 2005.

[34] Documentation of MidL Normal Form (MNF). URL[21].

[35] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001.

[36] G. Namasivayam, L. Liu, and M. Truszczyński. Syntax for ground logic programs – a proposal. URL[22], 2006.

[20] `http://www.tcs.hut.fi/~tjunttil/bcsat/`
[21] `http://www.cs.kuleuven.be/~dtai/krr/software/mnf.html`
[22] `http://www.cs.uky.edu/ai/groundlp-grammar-proposal.txt`

[37] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[38] E. Oikarinen and T. Janhunen. Modular equivalence for normal logic programs. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, pages 412–416. IOS Press, 2006.

[39] PB06: Input format. URL[23].

[40] S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. URL[24], 2006.

[41] RIF working group. URL[25].

[42] SAT competition 2002: requirements for the solvers. URL[26], 2002.

[43] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[44] T. Syrjänen. Lparse 1.0 user's manual. URL[27].

[45] J. Ward and J. Schlipf. Answer set programming with clause learning. In V. Lifschitz and I. Niemelä, editors, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, volume 2923 of *Lecture Notes in Artificial Intelligence*, pages 302–313. Springer-Verlag, 2004.

[23] `http://www.cril.univ-artois.fr/PB07/pb06_inputFormat.html`
[24] `http://combination.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf`
[25] `http://www.w3.org/2005/rules/wiki/RIF\_Working\_Group`
[26] `http://www.satcompetition.org/2004/format-solvers2004.html`
[27] `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`

A Grammar of *ASPils*

```
DIGIT      = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
POSINTEGER = DIGIT-'0', {DIGIT};
INTEGER0   = POSINTEGER | '0';
INTEGER    = ['-'], INTEGER0;

LETTER      = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' |
              'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' |
              'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' |
              'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' |
              'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
              'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | '_';
PARENTHESES = '(' | ')';
WHITESPACE  = ' ' |
              ? ISO 6429 character Horizontal Tabulation ? |
              ? ISO 6429 character Carriage Return ?       |
              ? ISO 6429 character Line Feed ?             |
              ? ISO 6429 character Vertical Tabulation ?   |
              ? ISO 6429 character Form Feed ?;
STRING      = {LETTER | DIGIT | PARENTHESES};
VERBAL      = {STRING | '-'   | WHITESPACE};
SAFE_VERBAL = '"', VERBAL, '"';

PROGRAM = {OBJECT_COMMENT, '0'},
          HEADER, ADDITIONAL_HEADERS, {ENTRY}, OBJECT_EOF,
          {OBJECT_COMMENT, '0'};

HEADER = TYPE_HEADER, POSINTEGER, VERSION, NORMAL_FORM, NUM_ADDITIONAL_HEADERS, '0';

VERSION               = POSINTEGER;
NORMAL_FORM           = POSINTEGER;
NUM_ADDITIONAL_HEADERS = INTEGER0;
ADDITIONAL_HEADERS    = {EXT_HEADER, '0'};

OBJECT_EOF = TYPE_EOF, INTEGER0, '0';


ENTRY  = OBJECT, '0';
OBJECT = OBJECT_META              | OBJECT_COMMENT                |
         OBJECT_ATOM              | OBJECT_RULE                   |
         OBJECT_FACT              | OBJECT_INTEGRITY_CONSTRAINT   |
         OBJECT_CONJUNCTION       | OBJECT_DISJUNCTION            |
         OBJECT_DEFAULT_NEGATION  | OBJECT_CARDINALITY_CONSTRAINT |
         OBJECT_WEIGHT_CONSTRAINT | OBJECT_WEIGHTED_LITERAL       |
         OBJECT_AGGREGATE_COUNT   | OBJECT_AGGREGATE_SUM          |
         OBJECT_AGGREGATE_MAX     | OBJECT_AGGREGATE_MIN          |
         OBJECT_AGGREGATE_TIMES   | OBJECT_OPERATOR_EQ            |
         OBJECT_OPERATOR_LEQ      | OBJECT_OPERATOR_LT            |
         OBJECT_OPERATOR_GEQ      | OBJECT_OPERATOR_GT            |
         OBJECT_UNARY_EQ          | OBJECT_UNARY_LEQ              |
         OBJECT_UNARY_LT          | OBJECT_UNARY_GEQ              |
         OBJECT_UNARY_GT          | OBJECT_OPTIMIZE;


OBJECT_ID      = POSINTEGER;
OBJECT_ID_LIST = OBJECT_ID, {OBJECT_ID};
LITERAL        = [-], OBJECT_ID;
LITERAL_LIST   = LITERAL, {LITERAL};
WEIGHT         = INTEGER;
WEIGHT_LIST    = WEIGHT, {WEIGHT};
```

```
OBJECT_META = TYPE_META, POSINTEGER, OBJECT_ID, SAFE_VERBAL, {META_OPTION};
META_OPTION = INTEGER;


OBJECT_COMMENT = TYPE_COMMENT, INTEGER0, [SAFE_VERBAL];

OBJECT_ATOM = TYPE_ATOM, POSINTEGER, OBJECT_ID, ATOM_NAME,
              {ATOM_OPTION}, [MODULE_OPTION];

ATOM_NAME          = (LETTER, STRING) | SAFE_VERBAL;
ATOM_OPTION        = HIDE | CLASSICAL_NEGATION;
HIDE               = '1';
CLASSICAL_NEGATION = '2', OBJECT_ID;
MODULE_OPTION      = INPUT | LOCAL;
INPUT              = '3';
LOCAL              = '4';


OBJECT_RULE                = TYPE_RULE, POSINTEGER, OBJECT_ID, LITERAL, LITERAL;
OBJECT_FACT                = TYPE_FACT, POSINTEGER, OBJECT_ID, LITERAL;
OBJECT_INTEGRITY_CONSTRAINT = TYPE_INTEGRITY_CONSTRAINT, POSINTEGER, OBJECT_ID,
                              LITERAL;


OBJECT_CONJUNCTION      = TYPE_CONJUNCTION,      POSINTEGER, OBJECT_ID, LITERAL_LIST;
OBJECT_DISJUNCTION      = TYPE_DISJUNCTION,      POSINTEGER, OBJECT_ID, LITERAL_LIST;
OBJECT_DEFAULT_NEGATION = TYPE_DEFAULT_NEGATION, POSINTEGER, OBJECT_ID, LITERAL;


OBJECT_CARDINALITY_CONSTRAINT = TYPE_CARDINALITY_CONSTRAINT, POSINTEGER, OBJECT_ID,
                                INTEGER0, INTEGER0, LITERAL_LIST;
OBJECT_WEIGHT_CONSTRAINT      = TYPE_WEIGHT_CONSTRAINT,      POSINTEGER, OBJECT_ID,
                                INTEGER,  INTEGER,  LITERAL_LIST, WEIGHT_LIST;


OBJECT_WEIGHTED_LITERAL = TYPE_WEIGHTED_LITERAL, POSINTEGER, OBJECT_ID,
                          LITERAL, WEIGHT;


OBJECT_AGGREGATE_COUNT = TYPE_AGGREGATE_COUNT, POSINTEGER, OBJECT_ID, LITERAL_LIST;
OBJECT_AGGREGATE_SUM   = TYPE_AGGREGATE_SUM,   POSINTEGER, OBJECT_ID, OBJECT_ID_LIST;
OBJECT_AGGREGATE_MAX   = TYPE_AGGREGATE_MAX,   POSINTEGER, OBJECT_ID, OBJECT_ID_LIST;
OBJECT_AGGREGATE_MIN   = TYPE_AGGREGATE_MIN,   POSINTEGER, OBJECT_ID, OBJECT_ID_LIST;
OBJECT_AGGREGATE_TIMES = TYPE_AGGREGATE_TIMES, POSINTEGER, OBJECT_ID, OBJECT_ID_LIST;


OBJECT_OPERATOR_EQ  = TYPE_OPERATOR_EQ,  POSINTEGER, OBJECT_ID, OBJECT_ID, OBJECT_ID;
OBJECT_OPERATOR_LEQ = TYPE_OPERATOR_LEQ, POSINTEGER, OBJECT_ID, OBJECT_ID, OBJECT_ID;
OBJECT_OPERATOR_LT  = TYPE_OPERATOR_LT,  POSINTEGER, OBJECT_ID, OBJECT_ID, OBJECT_ID;
OBJECT_OPERATOR_GEQ = TYPE_OPERATOR_GEQ, POSINTEGER, OBJECT_ID, OBJECT_ID, OBJECT_ID;
OBJECT_OPERATOR_GT  = TYPE_OPERATOR_GT,  POSINTEGER, OBJECT_ID, OBJECT_ID, OBJECT_ID;

OBJECT_UNARY_EQ  = TYPE_UNARY_EQ,  POSINTEGER, OBJECT_ID, OBJECT_ID, INTEGER;
OBJECT_UNARY_LEQ = TYPE_UNARY_LEQ, POSINTEGER, OBJECT_ID, OBJECT_ID, INTEGER;
OBJECT_UNARY_LT  = TYPE_UNARY_LT,  POSINTEGER, OBJECT_ID, OBJECT_ID, INTEGER;
OBJECT_UNARY_GEQ = TYPE_UNARY_GEQ, POSINTEGER, OBJECT_ID, OBJECT_ID, INTEGER;
OBJECT_UNARY_GT  = TYPE_UNARY_GT,  POSINTEGER, OBJECT_ID, OBJECT_ID, INTEGER;

OBJECT_OPTIMIZE = TYPE_OPTIMIZE, POSINTEGER, OBJECT_ID, STRATEGY;
STRATEGY        = LEX;
LEX             = TYPE_OPTIMIZE_LEX, OBJECT_ID_LIST;
```

```
TYPE_EOF                    = '0';

TYPE_HEADER                 = '1';

TYPE_META                   = '2';
TYPE_COMMENT                = '3';

TYPE_ATOM                   = '4';

TYPE_RULE                   = '5';
TYPE_FACT                   = '6';
TYPE_INTEGRITY_CONSTRAINT   = '7';

TYPE_CONJUNCTION            = '8';
TYPE_DISJUNCTION            = '9';
TYPE_DEFAULT_NEGATION       = 'a';

TYPE_CARDINALITY_CONSTRAINT = 'b';
TYPE_WEIGHT_CONSTRAINT      = 'c';

TYPE_WEIGHTED_LITERAL       = 'd';

TYPE_AGGREGATE_COUNT        = 'e';
TYPE_AGGREGATE_SUM          = 'f';
TYPE_AGGREGATE_MAX          = '10';
TYPE_AGGREGATE_MIN          = '11';
TYPE_AGGREGATE_TIMES        = '12';

TYPE_OPERATOR_EQ            = '13';
TYPE_OPERATOR_LEQ           = '14';
TYPE_OPERATOR_LT            = '15';
TYPE_OPERATOR_GEQ           = '16';
TYPE_OPERATOR_GT            = '17';

TYPE_UNARY_EQ               = '18';
TYPE_UNARY_LEQ              = '19';
TYPE_UNARY_LT               = '1a';
TYPE_UNARY_GEQ              = '1b';
TYPE_UNARY_GT               = '1c';

TYPE_OPTIMIZE               = '1d';
TYPE_OPTIMIZE_LEX           = '1e';
```