A Guide to SNARK

Mark E. Stickel Richard J. Waldinger
Vinay K. Chaudhri

May 12, 2000

Contents

1 Preface

SNARK, SRI's New Automated Reasoning Kit, is a theorem prover intended for applications in artificial intelligence and software engineering. SNARK is geared toward dealing with large sets of assertions; it can be specialized with strategic controls that tune its performance; and it has facilities for integrating special-purpose reasoning procedures with general-purpose inference.

SNARK has been used as the reasoning component of SRI's High Performance Knowledge Base (HPKB) system, which deduces answers to questions based on large repositories of information. It constitutes the deductive core of the NASA Amphion system, which composes software from components to meet users' specifications, e. g., to perform computations in planetary astronomy. SNARK has also been connected to Kestrel's SPECWARE environment for software development.

SNARK is a resolution-and-paramodulation theorem prover for first-order logic with equality—in this sense, it is in the same category as Argonne's OTTER ([McCune]). SNARK has provisions for both clausal and nonclausal reasoning, and it has an optional sort mechanism. It incorporates associative/commutative unification and a built-in decision procedure for reasoning about temporal points and intervals. It has no special facilities for proof by mathematical induction. It has some capabilities for abductive reasoning, which have been used in natural-language applications. SNARK is implemented in an easily portable subset of COMMON LISP.

SNARK is a refutation system; in other words, rather that trying to show directly that some assertions imply a desired conclusion, it attempts to show that the assertions and the negation of the conclusion imply a contradiction. It is an agenda-based system; that is, in seeking a refutation, it will put the assertions and the negation of the conclusion on an agenda. An agenda a list of formulas. When a formula reaches the top of the agenda, SNARK will perform selected inferences involving that formula and the previously processed formulas. The consequences of those inferences are added to the agenda. This process continues until the propositional symbol `false` is derived; this means that a contradiction has been deduced and the refutation is complete. The user has considerable control over the position at which a newly derived formula is placed on the agenda; this is one way in which a knowledgeable user can tailor SNARK's search strategy to a particular application.

This document is an example-driven tutorial introduction to SNARK that will allow the reader to experiment with the system. It does not purport

to introduce mathematical logic or resolution theorem proving; [Chang] provides an introduction to this style of theorem proving that does not assume any prior knowledge of logic. The guide also uses some notions introduced more fully in [Manna] and [Waldinger], particularly nonclausal resolution, quantifier force, and answer extraction. Knowledge of LISP syntax and basics is also assumed here (e. g., see [Graham] or [Pitman]). Nevertheless, it is intended that a reader who is willing to suspend incomprehension will be able to read this document without consulting other sources.

The guide is divided into several sections. In Getting Started (Section 2) we give the minimum information necessary to allow a user to get started with SNARK. We show how to enter axioms, and pose questions and set SNARK parameters. In Language and Logic (Section 3) we introduce the aspects of the SNARK language, including sorts, equality, and quantifiers, and we discuss mechanisms for dealing with them, including the paramodulation rule and skolemization. In Section 4, Topics in Answer Extraction, we discuss how SNARK is used to provide answers to questions (other than a simple Yes), and how answers are managed. In Efficiency Considerations (Section 5) we describe SNARK features that help us direct the search, to find an answer more quickly.

In Temporal Reasoning (Section 6) we describe the feature that allows SNARK to reason efficiently about temporal points and intervals. In Procedural Attachment (Section 7) we talk about how SNARK can do inferences using special-purpose external procedures supplied by the user. In Support for KIF/OKBC Users (Section 8) we describe KIF+C (KIF plus Classes), the facility that enables SNARK to understand assertions and queries phrased in KIF, the Knowledge Interchange Format, with some extensions from OKBC ("Open Knowledge-Base Connectivity"), an object-oriented framework for representing knowledge. This dialect was used in the HPKB project and other knowledge representation efforts.

In future versions of this guide we intend to include a description of SNARK's abduction mechanism, a reference manual, and a description of SNARK's internal workings. The guide is meant to be a living document, that will change as SNARK develops.

2 Getting Started

In this section, we give minimum information necessary to get started with SNARK.

2.1 SNARK language and theories

We begin by introducing the language of SNARK and then show how to describe subject matter as a logical theory. As a running example in this document, we illustrate of the use of SNARK to reason in a theory of family relationships.

2.1.1 The SNARK language

Terms stand for entities, such as things and people, while *formulas* stand for truth-values, either true or false. *Constants*, such as `carol` and *variables*, such as `?x`, are terms. A constant such as `betty` or `carol` refers to only one thing, while a variable may refer to different things, depending on context. (Note that SNARK is usually case-insensitive; it will not distinguish between the constants `carol`, `CAROL`, and `CaRoL`.)

Function symbols, such as `father` and `mother`, can be applied to a term to yield another term, such as `(father carol)` and `(mother ?x)`, standing for the father of Carol and the mother of `?x`, respectively. Because these function symbols take only one argument, we say that are *unary*, or of *arity* 1.

Predicate symbols, such as `parent` or `grandparent`, are applied to terms but yield *atoms*, which are elementary formulas, not terms. Thus

```
(parent carol ?x)
```

is an atomic formula that stands for a truth-value, which is true if Carol is indeed a parent of `?x`, and false otherwise. Predicate symbols stand for a relation between entities; `parent` stands for the parenthood relation. Because `parent` and `grandparent` take two arguments, we say they are *binary* predicate symbols, of arity 2. There are also *propositional* symbols, such as `it-is-raining`, which are themselves atoms, and can be either true or false. Finally, there are two special propositional symbols, the *truth symbols* `true` and `false`, which always stand for the truth-values true and false, respectively.

Other formulas are built up from atoms by successive application of logical *connectives* (`and`, `or`, `not`, `implies`, `implied-by`, `iff`, `xor` `nand`, and `nor`), the *universal quantifier* `forall` and the *existential quantifier* `exists`. Various synonyms are accepted for these symbols.

Thus, if `<Form1>` and `<Form2>` are formulas, so are `(and <Form1> <Form2>)`, `(exists (?x) <Form1>)`, and so forth. For example,

```
(forall (?x)
 (implies
  (parent ?x (mother carol))
  (grandparent ?x carol))
```

is a formula, which means that anyone who is a parent of the mother of Carol is a grandparent of Carol. (In the theory we have in mind, this formula stands for the truth-value true.) We will say that the occurrences of `?x` in the above formula are *bound* by the quantifier `(forall (?x)...)`. Variables that are not bound by any quantifier are said to be *free*.

Any symbol in the list following a quantifier is taken to be a variable, whether or not it is prefixed by `?`. Thus, the above formula could just as well have been written

```
(forall (x)
 (implies
  (parent x (mother carol))
  (grandparent x carol))
```

An occurrence of `x` not bound by a quantifier is taken to be a constant, while any symbol prefixed by `?` is automatically a variable, whether or not it is bound by a quantifier; we call these *?-variables*.

Together, terms and formulas make up the *expressions*. No expression is both a term and a formula. A string of symbols, such as

```
(father (parent ?y carol))
```

obtained by applying a function symbol to a formula is not a legal expression in the SNARK language.

2.1.2 SNARK Theories

A *theory*, then is a vocabulary of constant, function, and predicate symbols, and a set of formulas, called *assertions*, which describe the properties of the

entities mentioned in the theory. For example, in the family theory we are discussing, we can introduce the assertion

```
(parent betty carol)
```

to say that Betty is a parent of Carol.

The fact that fathers and mothers are parents is expressed by the two SNARK assertions

```
(forall (?x) (parent (father ?x) ?x))

(forall (?x) (parent (mother ?x) ?x))
```

In other words, the father of `?x` is a parent of `?x`, and the mother of `?x` is a parent of `?x`.

There is a convention that, in assertions, free ?-variables `?x` are actually bound by an invisible universal quantifier `(forall (?x)...)`. Thus, the above assertions could have been written

```
(parent (father ?x) ?x)

(parent (mother ?x) ?x).
```

Had we used variables without question-marks, however, we could not have omitted the quantifier `(forall (x)...)`; `x` would have been taken to be a constant.

The fact that a parent of a parent is a grandparent is expressed by the following SNARK assertion:

```
(implies
 (and (parent ?x ?y) (parent ?y ?z))
 (grandparent ?x ?z)).
```

That is, if ?x is a parent of ?y, and ?y is a parent of ?z, then ?x is a grandparent of ?z.

Once we have formulated a theory, we can pose a *query*, which is a SNARK formula treated as a question rather than an assertion. SNARK will attempt to show that a query follows from all the assertions of the theory. For example, if we want to ask if Carol has a grandparent, we may pose the query

```
(exists (?z) (grandparent ?z carol)).
```

In other words, we want to deduce that there exists `?z` such that `?z` is a grandparent of Carol.

The convention for queries is the opposite of that for assertions. In a query, a free ?-variable `?z` is bound by an invisible existential quantifier `(exists (?z)...)`. Thus, the above query could have been written

```
(grandparent ?z carol).
```

In attempting to prove this query, SNARK will assert the negation of the query and attempt to derive a contradiction from that and the other assertions. Thus, SNARK will assert

```
(not (grandparent ?z carol))
```

and attempt to deduce the truth symbol `false`. If it succeeds, we know that all the assertions together must be contradictory.

2.1.3 Clauses

SNARK is more efficient when dealing with formulas in *clausal form.* A `clause` is a special kind of formula that consists of a disjunction of `literals`, where each literal is either an atom or the negation of an atom. For instance, the formula

```
(or
 (not (parent ?person ?person1))
 (not (parent ?person1 ?person2))
 (grandparent ?person ?person2))
```

is in clausal form; it is the disjunction (`or`-ing) of three literals: one atom and two negations of atoms. This clause is equivalent to the assertion that the parent of a parent is a grandparent. Much resolution theorem proving is based on the fact that the problem of refuting any formula is equivalent to one of refuting a formula that is in clausal form, i. e., that consists of a conjunction (`and`-ing) of clauses. As its default, SNARK translates all formulas into clausal form, which it can deal with most efficiently.

2.2 Resolution

In searching for a refutation, SNARK will apply the resolution rule to the formulas on its agenda. In its simplest form, the rule can be applied to two given clauses of form

```
(or (not P) Q)
```

and

```
(or P R)
```

and will deduce the corresponding clause of form

```
(or Q R)
```

But the rule is more complex than that. The two given clauses do not need to have identical subatoms `P`. It is enough if the subatoms are *unifiable*—in other words, if one can make them identical by applying a substitution, replacing their variables by other terms.

The order of the disjuncts is irrelevant; `(not P)` and `P` do not need to be first. Also it is possible that either or both of the other subformulas `Q` and `R` are actually several disjuncts. Or they may be absent (i. e., one or both may be taken to be `false`); the given formulas are then simply `(not P)` or `P`, respectively, and the deduced formula will then be `R` or `Q`, respectively. If both `Q` and `R` are absent, the deduced formula is `false`, and the refutation is complete.

We shall see an application of the resolution rule in the following section.

2.3 Invoking SNARK

Let us see how to invoke SNARK to solve the above problem.

We start with a LISP environment and load the SNARK system; how to do this depends on your particular installation. We start by typing:

```
(initialize)
(use-resolution t)
```

The `initialize` function will clear out any previous formulas SNARK has derived and ready it to accept a new theory and perform a new proof. After initialization, we indicate which inference rules we would like to use, in this case resolution.

We then introduce the assertions of our theory as SNARK assertions:

```
(assert '(parent (father ?x) ?x) :name 'father-is-parent)

(assert '(parent (mother ?x) ?x) :name 'mother-is-parent)

(assert
 '(implies
   (and (parent ?x ?y) (parent ?y ?z))
   (grandparent ?x ?z))
 :name 'parent-of-parent-is-grandparent)
```

Note that each assertion is given a name, a string that follows the *keyword* `:name`; these names have no effect on the proof, but will be used in the trace to indicate where the assertion is used. If the name is omitted, a number will be used instead. The assertion, its name or number, and some other information are stored in a structure called a "row."

A query is initiated by the SNARK `prove` function:

```
(prove '(grandparent ?z carol)
       :name 'does-carol-have-a-grandparent)
```

Here we have given the query a name, for our own convenience in following the proof.

The query has the effect of putting the negation of the formula

```
(grandparent ?z carol)
```

into a row and then invoking SNARK to find a refutation.

SNARK will respond by first giving a list of the options that have been selected. Some of these (e. g., resolution) we have selected; most of them are defaults:

```
; The current SNARK option values are
;       (USE-RESOLUTION T)
;       (USE-HYPERRESOLUTION NIL)
; ...
```

This means that we have selected to use resolution and not hyperresolution. (Hyperresolution is a variant of resolution that can do several resolution steps at once and is often more efficient.) SNARK will print out a complete list of the options it offers, and the settings that have been selected. If the user has

not selected a setting, SNARK will choose the default. This printing, and the other printing reported here, can be altered by the user—we are reporting SNARK's default behavior.

SNARK will then print out the assertions it has been given, and the negation of the conclusion:

```
(Row father-is-parent
   (parent (father ?x) ?x)
   assertion)
(Row mother-is-parent
   (parent (mother ?x) ?x)
   assertion)
(Row parent-of-parent-is-grandparent
   (or (not (parent ?x ?y)) (not (parent ?y ?z))
       (grandparent ?x ?z))
   assertion)
(Row does-carol-have-a-grandparent
   (not (grandparent ?x carol))
   ~conclusion)
```

For each row, SNARK provides its name or number, its formula, and an explanation of its origin. SNARK freely renames variables according to its own conventions—thus, in the negation of the conclusion, the variable `?z` has been renamed `?x`. SNARK provides control options to suppress the printing of formulas.

Next, SNARK gives an account of the steps it takes in searching for the proof. If it finds a refutation, it will then reprint only those rows that play a role. Here is the first of these steps:

```
(Row 12
   (or (not (parent ?x ?y))
       (not (parent ?y carol)))
   (resolve does-carol-have-a-grandparent
            parent-of-parent-is-grandparent))
```

Note that the row contains some information about how its formula was deduced, if it was not an assertion or the negation of the conclusion. Here Row 12 was obtained by applying the resolution rule to the negation of the conclusion and the assertion that the parent of a parent is a grandparent.

The unifying substitution replaced the variable `?z` with the constant `carol`; this substitution is reflected in Row 12.

Here is the next step:

```
(Row 23
   (not (parent ?x carol))
   (resolve 12 father-is-parent))
```

Note that the fact that `?x` occurs in both Row 12 and in the assertion `father-is-parent` is not logically significant—variables are systematically renamed before unification to avoid such coincidences. Otherwise the two rows would not be unifiable

Finally, we obtain

```
(Row 24
   false
   (resolve 23 father-is-parent))
```

The formula `false` indicates that a refution has been obtained. The intuitive idea behind this proof is that Carol's father is her parent, Carol's father's father is *his* parent, and, since the parent of a parent is a grandparent, Carol's father's father is her grandparent.

SNARK will then report on statistics involved in the proof search, e. g.:

```
; Summary of computation:
;     15 formulas have been input or derived (from 7 formulas).
;     12 (80%) were retained.  Of these,
;          1 ( 8%) were simplified or subsumed later,
;          0 ( 0%) were deleted later because
                   the agenda was full
;         11 (92%) are still being kept.
;
; Run time in seconds excluding printing time:
;      0.01   5%   Resolution
;      0.01   5%   Forward subsumption
;      0.17  89%   Other
;      0.19        Total
; ....
```

To conclude, SNARK will return a value,

```
:PROOF-FOUND.
```

If the desired conclusion does not follow from the assertions, SNARK will fail to find a proof. There are two ways it may behave in this case. It may exhaust all possible inferences without finding a refutation; in this case it will report that the agenda is empty, and halt, reporting

```
; All agendas are empty.
:AGENDA-EMPTY
```

Or it may continue to run on indefinitely without ever finding a refutation.

For instance, if we do not give SNARK the assertion that the parent of a parent is a grandparent, SNARK will print the two assertions and the negation of the conclusion, give the statistical summary, and then halt with an empty agenda. In this configuration SNARK is a *logically complete* theorem prover; that is, if conclusion does follow from the assertions, SNARK will find a proof. Therefore, we know that in this case the assertions do not imply the conclusion.

On the other hand, if SNARK runs on for longer than we expect, there is no way, in general, to determine if the conclusion is not valid or if we simply haven't given it enough time. This is not a particular weakness of SNARK; it is a theoretical limitation on all theorem provers that are logically complete for first-order logic or more expressive logics.

If SNARK is running longer than we expect, we may interrupt it by typing a carriage return at the keyboard. (In most implementations we may type any character for this purpose.) SNARK will then ask

```
Stop now?
```

We may answer `yes`, meaning to interrupt the proof, or `no`, meaning to continue the proof. Even if we choose to interrupt the proof, we may continue later by calling the function `(closure)`, the SNARK function that computes the logical consequences of the current set of rows. This function is also invoked automatically when we call `prove`. We say more about calling it explicitly in Section 4.1.

Now let us see how to change SNARK's strategies and options.

We indicated that SNARK was to use the (binary) resolution rule by executing

```
(use-resolution t)
```

after calling `initialize` but before calling `prove`. If we change our mind and decide to try hyperresolution instead, we can invoke

```
(use-resolution nil)
(use-hyperresolution t)
```

before starting the proof.

We have mentioned that clause form is the default for SNARK. If we decided to try using nonclausal resolution instead, we may say

```
(use-resolution t)
(use-clausification nil)
```

after initialization but before beginning the proof. In this case, SNARK will invoke a nonclausal version of the resolution rule. (Caution: no nonclausal version of hyperresolution has been implemented.)

This illustrates a pattern with setting SNARK options. Henceforth, when we say that we select SNARK option `some-option`, we shall mean that we execute

```
(use-some-option t)
```

after initializing SNARK but before beginning the proof. As an abbreviation, we may simply say

```
(use-some-option)
```

to select `some-option`. To turn the option off, we say

```
(use-some-option nil).
```

Most of the options that SNARK lists when it begins a proof can be turned on or off in this manner.

If we want to ask SNARK what its setting is for a particular option `some-option`, we may invoke

```
(use-some-option?)
```

For instance, if we want to see if SNARK is currently using the resolution rule, we invoke

```
(use-resolution?)
```

Invoking `initialize` causes SNARK to revert to its default options. Later (in Section 2.4), we shall see how to change those defaults.

Exercise: Resolution versus Hyperresolution. Run SNARK on the problem of finding the grandparents of Carol; compare its performance using resolution and hyperresolution. Also compare the results using the clausal and the nonclausal versions of resolution. See what happens if you request the nonclausal version of hyperresolution, nonexistent at the time of this writing.

The following section, on how to change SNARK's default behavior, may be omitted on first reading.

2.4 Changing SNARK's Defaults

Each time we initialize SNARK, we restore the setting of all its options to its defaults. We have seen how, after initialization, we can reset SNARK's options. This is appropriate if we want to change SNARK's behavior for a particular query, but want it to resume its normal behavior subsequently, the next time it is initialized. If we want a more permanent change, it is convenient to change SNARK's default behavior; there is a convention about how to go about this.

Here is an example: as we have seen, use of the resolution rule is not the default in SNARK; if we want to turn it on, we must say

```
(use-resolution t)
```

after initializing SNARK and before beginning the proof. Subsequent calls to `(initialize)` will restore the default, i. e., will turn resolution back off. If we normally do want to use resolution (as opposed to hyperresolution, say), we can change SNARK's default; we may invoke

```
(default-use-resolution t).
```

This in itself will not change SNARK's setting on the resolution option, but subsequent invocation of `initialize` will then turn on resolution. Thus, typically, we will set the default before we invoke initialize.

In general, calling

```
(default-use-some-option ...)
```

changes the default setting for `some-option`. Calling `initialize` returns SNARK to its default setting for all options. Calling

```
(use-some-option ...)
```

changes the setting for `some-option` temporarily, without changing the default. Subsequent calls to `initialize` return the setting to its default again. This enables us to experiment with different settings for the options without changing the default.

2.5 Basic Answer Extraction

In the example of Section 2.3, we have established that Carol has at least one grandparent, but we have not answered the question "Who is Carol's grandparent?" For this purpose, we may use SNARK's answer-extraction mechanism. We include in the `prove` statement an indication of what constitutes an answer, should the proof succeed:

```
(prove '(grandparent ?z carol) :answer '(ans ?z))
```

Notice that we have given our query a new component, *the answer formula*, which follows the keyword `:answer`. This means we are asking SNARK to tell us what value of `?z` will allow us to show that `?z` is Carol's grandfather. In response, SNARK proves the conclusion again, reporting in each row what substitution was made for `(ans ?z)` at that stage of the proof:

```
(Refutation
(Row father-is-parent
   (parent (father ?x) ?x)
   assertion)
(Row parent-of-parent-is-grandparent
   (or
    (not (parent ?x ?y))
    (not (parent ?y ?z))
    (grandparent ?x ?z))
   assertion)
(Row does-carol-have-a-grandparent
   (not (grandparent ?x carol))
   ~conclusion
   Answer (ans ?x))
(Row 12
   (or (not (parent ?x ?y)) (not (parent ?y carol)))
   (resolve
     does-carol-have-a-grandparent
```

```
     parent-of-parent-is-grandparent)
   Answer (ans ?x))
(Row 23
   (not (parent ?x carol))
   (resolve 12 father-is-parent)
   Answer (ans (father ?x)))
(Row 30
     false
     (resolve 23 father-is-parent)
     Answer (ans (father (father carol))))
```

At Row 23, we see that if `?x` is a parent of Carol, then the father of `?x` is a suitable answer (that is, `?x` is a grandparent of Carol.) At the end of the proof, we see that the father of the father of Carol is a grandparent of Carol. Later (Section 4.1), we shall see how to obtain more than one answer for a given query.

Use of the predicate symbol `ans` is arbitrary; we can give any formula as the answer formula. We could not, however, say simply

```
(prove '(grandparent ?z carol) :answer '?z)
```

because the answer must be a formula, not a term.

This concludes the Getting Started section, which acquaints the user with the bare minimum necessary to experiment with the system. We now give a more detailed introduction to SNARK.

3 Language and Logic

In this section we give additional SNARK language constructs and the mechanisms it has for dealing with these constructs. We include the syntax of symbols, the sort mechanism, equivalence, equality, and quantifiers.

3.1 The Syntax of Symbols

The syntax of the basic SNARK symbols can be described as follows.

A *proposition symbol*, *predicate symbol*, or *function symbol* can be any LISP symbol that

- is at least one character long,

- does not begin with the character "?",
- is not the symbol NIL,
- is not the (keyword) symbol :none, and
- is not another keyword symbol, such as :name or :answer. [1]

A *constant symbol* can be a LISP number, character, string, or any LISP symbol that

- is at least one character long,
- does not begin with the character "?",
- is not the (keyword) symbol :none, and
- is not another keyword symbol. [2]

SNARK compares constant symbols using the LISP EQL function. Thus, 3 and 3.0 are different constant symbols. Ordinarily, (EQL "ABC" "ABC") may be false, but SNARK will use a single copy of string constants so that such tests will succeed.

A *variable symbol* can be any LISP symbol that

- is at least two characters long,
- begins with the character "?", and
- is not a keyword symbol.

A variable symbol bound by a quantifier can also be any LISP symbol that

- is at least one character long,
- does not begin with the character "?",
- is not the symbol NIL, and
- is not a keyword symbol.

Whether X is a variable symbol or a constant symbol depends on whether the occurrence of X is bound (or being bound) by a quantifier or not.

[1] Unless SNARK option allow-keyword-proposition-symbols, allow-keyword-predicate-symbols, or allow-keyword-function-symbols is true.

[2] Unless SNARK option allow-keyword-constant-symbols is true.

3.2 Sorts

Sorts provide a mechanism to restrict the possible instantiations of a variable in a formula. Sorts can substantially reduce the cost of inference by restricting the search space. Using sorts some formulas can be stated more compactly.

In our theory of family relationships, everything is tacitly assumed to be a person. If an assertion has a global variable `?x`, that variable is assumed to stand for a person, but that assumption has not been made explicit.

If we were to combine this theory with another theory of, say, inanimate objects, we would have to differentiate between those assertions that are meant to apply to people and those that are meant to apply to objects. For example, the assertion that says that everyone's father is his parent holds only for people, not objects.

We could do this by introducing predicate symbols `person` and `object`, and augmenting our axioms with antecedents that classify the variables. For instance, our axiom that fathers are parents could be written

```
(assert
 '(implies
   (person ?x)
   (parent (father ?x) ?x))
 :name 'father-is-parent)
```

However, such axioms are more cumbersome to write, particularly if they contain many variables. Moreover, including conditions such as `(person ?x)` makes proofs longer and, consequently, more difficult to find. Every time SNARK uses the axiom to prove that a term `(father <term>)` is a parent, it must prove `(person <term>)`.

In fact, we might wish to limit the `parent` and `father` symbols so that their arguments are forced to be people. We might choose to make it illegal to write an assertion that mentioned `(father the-maltese-falcon)`, since `the-maltese-falcon` is not a person.

The sort mechanism achieves this by introducing new symbols, called *sorts*, which stand for nonempty sets of entities. The terms of our vocabulary are classified as being *of* certain sorts.

For instance, we may introduce two new sorts, `person` and `object`, standing for sets of people and inanimate objects, respectively, by the declarations

```
(declare-sort 'person)
(declare-sort 'object).
```

If a term is of sort `person`, that means that it must stand for a person.

The intersection of two sorts is assumed to be nonempty unless we deny it explicitly. For instance, to say that no one is both a person and an object, we must say

```
(declare-disjoint-sorts 'person 'object)
```

(If we make this declaration, we may omit the first two.) Otherwise, SNARK will assume that some entity is both a person and an object.

Let us suppose that we want a richer sort structure; we want to introduce sorts for men and women as well as people. Then, instead of (or in addition to) the earlier declaration for `person`, we could provide

```
(declare-subsorts 'person 'man 'woman).
```

This means that `person` has subsorts `man` and `woman`; every term of sort `man` is also of sort `person`, and similarly for `woman`.

Finally, if we want to say that `man` and `woman` are a disjoint partition of `person`, we can say

```
(declare-sort-partition 'person 'man 'woman).
```

This means that not only are `man` and `woman` disjoint subsorts of `person`, but also that every term of sort `person` must be either of sort `man` or of sort `woman`.

It is also possible to use boolean operators to define new sorts or to express relationships between sorts. For example, if we want to introduce a new sort that includes objects and men, we can say

```
(declare-sort 'object-or-man :iff '(or object man))
```

This means that every term of sort `object-or-man` is of sort `object` or of sort `man`, and every term that is of one of these two sorts is also of sort `object-or-man`. Other boolean operators, such as `and` and `not`, may also be used. SNARK is able to deduce relationships between sorts declared in this way. For instance, it will know that the sort `object-or-man` is the same as the sort `same-sort` declared by

```
(declare-sort 'same-sort
              :iff '(and (or object person) (not woman)))
```

In this way, SNARK's sort mechanism is unusually versatile.

Once we have established a sort structure, we can classify our vocabulary accordingly. For instance, we can declare Bob to be a man, Carol to be a woman, and the-maltese-falcon to be an object:

```
(declare-constant-symbol 'bob :sort 'man)
(declare-constant-symbol 'carol :sort 'woman)
(declare-constant-symbol 'the-maltese-falcon :sort 'object)
```

Because we have declared that `man` and `woman` are disjoint sorts, this also means that `bob` is not of sort `woman` and `carol` is not of sort `man`. Because both `man` and `woman` are subsorts of `person`, and because we have declared that `person` and `object` are disjoint, this means further that neither `bob` nor `carol` can be of sort `object`.

We can declare the function symbol `father` to take a person as an argument and yield a man as a value, as follows:

```
(declare-function-symbol 'father 1 :sort '(man person))
```

Note that for the declaration of a function, the sort for the value of the function comes before the sorts of its arguments. This declaration must be made before we use the symbol `father` in an assertion or conclusion. Also, it is required to include the number of arguments (arity) `1` in the declaration, because if `father` is sometimes used with more than one argument, those occurrences stand for different functions. A separate declaration could be given for a two-argument symbol `father`, with different sorts for its arguments.

If we want to give a function or relation an alternative name, we use the *alias* mechanism. For instance, we can give the one-argument function `father` the alternative name `father-1` by providing a declaration

```
(declare-function-symbol 'father 1 :sort '(man person)
  :alias 'father-1)
```

Then we can use the name `father-1` when we need to refer unambiguously to the one-argument father. The two-argument `father` could be given the alias `father-2`. Usually, however, we do not need to distinguish between them because SNARK can see count how many arguments are provided. In the trace of the proof, both functions will be represented as `father`.

The fact that the predicate symbol `parent` takes two arguments of sort `person` is expressed in its declaration:

```
(declare-predicate-symbol 'parent 2
  :sort '(boolean person person))
```

Note that predicate symbols are declared as if they were function symbols that return a value of sort `boolean`.

It is also possible to declare the sorts of variables, such as

```
(declare-variable-symbol '?some-guy :sort 'man)
```

If an assertion or query contains free (unquantified) occurrences of the variable `?some-guy`, they will then automatically be of sort `man`.

If a formula contains explicit quantifiers, we can declare the sort of a variable within the quantifier itself. For example, in a formula

```
(forall ((?this-man :sort man)
         (?that-man :sort man)) ...),
```

the variables `?this-man` and `?that-man` are of sort `man`.

Symbols such as `?man`, `?man1`, ..., are automatically variables of sort `man`. It is illegal to define certain symbols, such as `u`, ..., `z`, `?u`, ..., or `?z`, perhaps followed by any number of digits, as sorted variables; by convention, these are reserved to be unsorted.

The syntax of sorts can be described more exactly as follows:

A *sort symbol* can be any LISP symbol that

- is at least two characters long,
- does not begin with the character "`?`,
- is not the symbol `NIL`,
- is not the (keyword) symbol `:none`, and
- is not a keyword symbol. [3] *TRUE* and *FALSE* are predefined as the topmost bottommost sorts.

[3] Unless SNARK option `allow-keyword-sort-symbols` is true.

It is permissible to intermix sorted and unsorted symbols; unsorted terms are implicitly of sort `true`, and every sort is a subsort of `true`. Thus the unsorted version of SNARK can be regarded as a special case of the sorted version.

Once a sort structure has been declared, it becomes illegal to apply a function or a predicate symbol to a term whose argument is of an unsuitable sort. Thus SNARK will give an error if we say

```
(parent the-maltese-falcon bob),
```

because `parent` takes arguments of sort `person`, and `the-maltese-falcon` is not of sort `person`.

Thus, the sort mechanism serves as a debugging aid. In a sorted theory, many errors in formulating assertions and queries show up as sort errors, and are easily detected. Without the sort mechanism, some of these conceptual errors would only become evident when SNARK failed to find a proof; when that happens, it is difficult to decide which assertion contains the error that is the cause of the failure.

A more important effect of the sort structure is that it limits the application of resolution and other inference rules. SNARK will never allow two terms to be unified if their sorts are incompatible. For example, the variable `?person1` may be unified with a constant `carol` of sort `woman`, because `woman` is a subsort of `person`—we can always take `?person1` to stand for a woman. But a variable `?woman1` may not be unified with a constant `the-president` of sort `person`, because we cannot be certain that `the-president` is a woman.

Two variables of different sorts can be unified if we have not declared the sorts to be disjoint—the unified variable will have as its sort the conjunction of the two original sorts. A variable `?man1` cannot be unified with a variable `?woman2`, because we have declared the sorts `man` and `woman` to be disjoint; no one can be both a man and a woman. But a variable `?person1` can be unified with a variable `?man2` and vice versa; the unified variable will be of sort `man`. SNARK will block application of resolution, paramodulation, and other rules if they violate these restrictions.

The sort mechanism gives us abbreviated ways of saying things about subsets of our universe of discourse. For example, we can introduce an assertion

```
(assert '(not (brother ?woman ?person))
 :name 'women-are-not-brothers)
```

to mean that a women cannot be the brother of any person. As we indicated earlier, if we had no sort mechanism, we would be forced to write this in terms of predicate symbols, as

```
(assert
 '(implies
   (and (woman ?x) (person ?y))
   (not (brother ?x ?y)))
 :name 'women-are-not-brothers)
```

The restrictions on unification makes SNARK behave as if the conditions `(woman ?x)` and `(person ?y)` were actually there, when they are not.

Not only is the latter form of assertion more cumbersome to write, but also it is less efficient for SNARK. The sort mechanism allows us to make in one step an inference that would require several steps if people and women were represented by predicate symbols.

Furthermore, the version of `women-are-not-brothers` with sorts is a unit assertion—it has no connectives. Unit clauses have a beneficial effect on the search space. In particular, applying the resolution rule to a unit and a formula yields a smaller formula; applying the resolution rule to a nonunit and a formula may yield a larger formula, and produce a correspondingly larger search space.

In declaring a sort structure, we are saying what expressions in the language are meaningful, not what expressions are true. For instance, we do not declare the predicate symbol `brother` with the sort declaration

```
(declare-predicate-symbol 'brother 2
  :sort '(boolean man person)).
```

Even though we expect that someone's brother will always be a man, we do not want to exclude from the language formulas in which the first argument of `brother` is of sort `woman`; otherwise, we would have no way of saying that the expression `(brother betty bob)` is false. Merely being false is not the same as being meaningless. We declare `brother` on sorts for which the brother relationship is meaningful, certainly on all people.

Note that there is no way to include in an assertion the explicit condition that something is of a particular sort, or that it is not of a particular sort. For instance, we cannot say something like

```
(implies
 (brother ?person1 ?person2)
 ``?person1 is of sort man'')
```

in the SNARK language. In choosing to represent gender as a sort, we have decided that we will know in advance whether a person is male or female, and it won't be necessary to deduce such things.

Note that is it permissible to use the same symbol to stand for a sort and a predicate symbol; however, SNARK will regard that as a coincidence; it will not ensure that there is any relationship between the meanings of the two symbols. (KIF+C, the KIF-OKBC dialect of SNARK, described in Section 8, does enforce a relationship between sorts and predicate symbols of the same name.)

Example: Grandmothers. In Section 2.5, we showed how to find a grandparent of Carol. Suppose we would like to find a grandmother of Carol, not just any grandparent. Assume we have introduced sorts `man` and `woman` and have declared the sorts for `father` and `mother` accordingly, so that they yield people of sorts `man` and `woman` respectively. Then we may pose the query

```
(prove '(grandparent ?woman carol) :answer '(ans ?woman)).
```

In other words, we are asking for a person of sort `woman` who is a grandparent of Carol. We will then obtain an answer, either `(mother (father carol))` or `(mother (mother carol))`, depending on which proof is found first. (In Section 4.1, we shall see how to obtain more than one answer to such queries.)

3.3 Skolemization

If a formula contains explicit quantifiers, SNARK will remove the quantifiers by skolemization. In particular, existentially quantified variables in assertions will be replaced by functional terms, where the function is a newly introduced *skolem function* or *skolem constant*. We shall call the entire functional term a *skolem term*. Universal quantifiers will be removed too, but their variables will remain variables. (Actually, whether a quantifier is treated as universal or existential depends on whether it appears within a negation in the assertion—see the remark on Quantifier Force on Page 29).

For example, in Section 2.1 we introduced an assertion, later called `parent-of-parent-is-grandparent`, that said that any parent of a parent is a grandparent. Suppose now we need to state the converse, that any grandparent is the parent of a parent. Then we may formulate this assertion in terms of an explicit existential quantifier, as (in the sorted theory)

```
(assert
 '(implies
   (grandparent ?person1 ?person2)
   (exists (?person)
           (and
            (parent ?person1 ?person)
            (parent ?person ?person2))))
 :name 'grandparent-is-parent-of-parent)
```

Note that we cannot omit the existential quantifier, because unquantified variables have tacit universal quantification. If the quantifier were missing, we would be saying that every grandparent is a parent of every person, and that every person is in turn the parent of the grandchild.

The new assertion will be translated by SNARK (if we turn off the clausification option, Section 2.3) into

```
(Row grandparent-is-parent-of-parent
   (implies
    (grandparent ?person1 ?person2)
    (and (parent ?person1 (#:person-sk1 ?person1 ?person2))
         (parent (#:person-sk1 ?person1 ?person2) ?person2)))
   assertion)
```

Here `#:person-sk1` is the skolem function SNARK introduced in replacing the existential quantifier. Intuitively speaking, if `?person1` is a grandparent of `?person2`, then

```
(#:person-sk1 ?person1 ?person2)
```

is the child of `?person1` who is a parent of `?person2`. Skolem function symbols are prefixed by the sort of the variable whose quantifier is being removed—in this case `person`. Note again that SNARK renames the variables in a row as it sees fit.

Note that, instead of using an explicit existential quantifier, we could have phrased the assertion as follows:

```
(assert
 '(implies
   (grandparent ?person1 ?person2)
   (and (parent ?person1 (link ?person1 ?person2))
        (parent (link ?person1 ?person2) ?person2)))
        :name 'grandparent-is-parent-of-parent)
```

Here `link` is our own name for the skolem function that SNARK introduced automatically. This formulation may be more mnemonic—the link is the person who connects a grandparent and grandchild in the family tree (either the grandchild's mother or father). Also, we are free to use the function `link` in other assertions and queries, if we so choose.

It is possible to provide SNARK with a name to use when it skolemizes a quantifier, using the keyword `conc-name` in the quantifier, as follows:

```
(assert
 '(implies
   (grandparent ?person1 ?person2)
   (exists ((?person :conc-name link))
           (and
            (parent ?person1 ?person)
            (parent ?person ?person2))))
 :name 'grandparent-is-parent-of-parent)
```

(Note that the double parentheses in the list of quantified variables are required; otherwise, SNARK will expect `:conc-name` and `link` to be additional variables of the quantifier and report an error.) This formula will be skolemized (assuming clausification is turned off) as

```
(Row grandparent-is-parent-of-parent
   (implies
    (grandparent ?person ?person1)
    (and
     (parent ?person (#:link1 ?person ?person1))
     (parent (#:link1 ?person ?person1) ?person1)))
   assertion)
```

Here SNARK has not used the exact name we have provided, but it has incorporated the string `link` into the name `#:link1` it has constructed. This is because SNARK needs to invent a unique symbol every time it constructs a

skolem function. If we gave the same conc-name `link` to another quantifier as well, SNARK would invent a different skolem function symbol for the new quantifier, but both symbols would incorporate the string `link`.

If an existential quantifier to be removed is within the scope of some universal quantifiers, the new skolem function will have arguments—the variables of those quantifiers. (We include the invisible quantifiers that bind the free variables of the assertion.) Otherwise, we introduce skolem constants, not skolem functions. For example, if we introduce the assertion

```
(assert
 '(exists (?person) (parent betty ?person))
 :name 'betty-has-a-child),
```

SNARK will transform this into the assertion

```
(Row betty-has-a-child
   (parent betty #:person-sk9)
   assertion)
```

Here SNARK has made up its own name, the skolem constant `#:person-sk9`, for Betty's child.

Remark: Quantifier Force. If an existential quantifier is within the scope of a single negation, it will behave as a universal quantifier and will be treated accordingly. In particular, during skolemization, its variables will remain as variables. This happens even if the negation is only implicit, when the quantifier is in a query or in the antecedent of an implication, say. A quantifier that behaves as a universal, even if it is syntactically an existential, is said to have *universal force*.

Similarly, it can happen that a universal quantifier will have *existential force* if it is within the scope of a single explicit or implicit negation—in this case, its variable will be replaced by a skolem term during skolemization.

Thus, the way quantifiers are treated in queries is precisely the reverse of the way they are treated in assertions—universally quantified variables are replaced by skolem terms, while existentially quantified variables remain variables when quantifiers are removed. Additional negations reverse the force of the quantifiers yet again; thus, an existential quantifier that is within the scope of precisely two explicit or implicit negations does have existential force.

The skolemization procedure is justified by showing that a set of rows is contradictory precisely when the skolemized version of those rows is contradictory.

3.4 Equivalence

We could have combined the two assertions that defined the notion of a grandparent into a single SNARK assertion using equivalence, as follows:

```
(assert '(iff
          (grandparent ?person1 ?person2)
          (exists (?person)
                  (and (parent ?person1 ?person)
                       (parent ?person ?person2))))
        :name 'grandparent-iff-parent-of-parent)
```

SNARK will break this down into two implications; if clausification has not been turned off, these will then be transformed into clauses.

Exercise: In-Laws. In a sorted theory of families, introduce predicate symbols for the relationships `siblings`, `sister`, `brother`, `spouse`, `wife`, `husband`, `sister-in-law`, and `brother-in-law`. Introduce assertions that relate these notions to each other. Prove that if a man is a woman's brother-in-law, she is his sister-in-law. (In other words, the `brother-in-law` and `sister-in-law` relations are *inverses*.)

Hints. A man is a brother-in-law to another person if he is husband to a sibling of that person, or if he is brother to a spouse of that person. Similarly for sister-in-law.

You may introduce assertions to express the following facts:

A husband is a spouse; a wife is a spouse.

A male spouse is a husband; a female spouse is a wife.

A brother is a sibling; a sister is a sibling.

A male sibling is a brother; a female sibling is a sister.

Solution. The brother-in-law relation is defined by the assertion

```
(assert
 '(iff
   (brother-in-law ?person1 ?person2)
   (or
    (exists (?person)
            (and (husband ?person1 ?person)
                 (sibling ?person ?person2)))
    (exists (?person)
            (and (brother ?person1 ?person)
                 (spouse ?person ?person2)))))
 :name 'brother-in-law)
```

The definition of `sister-in-law` is analogous.

The problem is stated as follows:

```
(prove
 '(forall (?man ?woman)
       (implies
        (brother-in-law ?man ?woman)
        (sister-in-law ?woman ?man)))
 :name 'brother-and-sister-in-law-are-inverses)
```

Note that we cannot omit the universal quantifier `(forall (?man ?woman) ...)` here, because then the query would be treated as if it were surrounded by an invisible existential quantifier `(exists (?man ?woman)...)`, which would change its meaning.

The fact that all husbands are spouses is expressed by the assertion

```
(assert
 '(implies (husband ?person1 ?person2)
           (spouse ?person1 ?person2))
 :name 'husbands-are-spouses)
```

The fact that male spouses are husbands is expressed by the assertion

```
(assert
 '(iff (spouse ?man ?person)
       (husband ?man ?person))
 :name 'male-spouses-are-husbands)
```

Similar assertions apply to wives, and to brothers, sisters, and siblings. There are other assertions that could be made (e. g., the definition of `sibling`), but they are not necessary to solve this problem.

3.5 Equality and Paramodulation

The equality relation `(= ?x ?y)` means that `?x` and `?y` stand for the same thing.

Although it is possible to describe the equality relation by giving its axioms as SNARK assertions, if we want to reason about the relation it is usually best to include the paramodulation rule among our rules of inference. The rule allows us to replace equals with equals. In its simplest form, the rule can be applied to two clauses of form

```
(or P[s])
```

and

```
(or (= s t) Q),
```

where `s` and `t` are terms and the clause `(or P[s])` contains at least one occurrence of `s`. Then the paramodulation rule will deduce the corresponding clause of form

```
(or P[t] Q),
```

where the literals `P[t]` are obtained from the literals `P[s]` by replacing all occurrences of `s` with `t`. The roles of `s` and `t` may be reversed, so that the rule is applied right to left to replace occurrences of `t` with `s`. As with the resolution rule, the order of literals is not meaningful, and the replaced terms need not be identical to `s` or `t`, but merely unifiable.

We can include paramodulation in our arsenal by selecting the option `use-paramodulation`. Then no axioms for equality need be provided by the user.

Let us look at an example of the application of the paramodulation rule.

Example: Grandchildren of Alice. Suppose we wish to tell SNARK that Carol's mother is Betty and Betty's mother is Alice; then we may add the new assertions

```
(assert '(= (mother carol) betty))

(assert '(= (mother betty) alice))
```

We assume that we have previously declared `alice`, `betty`, and `carol` to be constants of sort `woman`, and `mother` to be a function that takes a `person` into a `woman`.

If we then want to ask who is a grandchild of Alice, we may invoke SNARK with the query

```
(prove '(grandparent alice ?person) :answer '(ans ?person)).
```

If paramodulation has been selected, SNARK will complete the proof with the answer `carol`. Here is the entire refutation:

```
(Refutation
(Row mother-is-parent
   (parent (mother ?person) ?person)
   assertion)
(Row mother-of-carol-is-betty
   (= (mother carol) betty)
   assertion)
(Row mother-of-betty-is-alice
   (= (mother betty) alice)
   assertion)
(Row who-is-the-grandchild-of-alice?
   (not (grandparent alice ?person))
   ~conclusion
   Answer (ans ?person))
(Row grandparent-iff-parent-of-parent-39
   (or (not (parent ?person ?person1))
       (not (parent ?person1 ?person2))
       (grandparent ?person ?person2))
   assertion)
(Row 42
   (parent betty carol)
   (paramodulate mother-is-parent mother-of-carol-is-betty))
(Row 43
   (parent alice betty)
```

```
   (paramodulate mother-is-parent mother-of-betty-is-alice))
(Row 90
   (or (not (parent alice ?person))
       (not (parent ?person ?person1)))
   (resolve who-is-the-grandchild-of-alice?
    grandparent-iff-parent-of-parent-39)
   Answer (ans ?person1))
(Row 277
   false
   (rewrite (resolve 90 42) 43)
   Answer (ans carol))
)
```

The derivations of Rows 42 and 43 illustrate the use of paramodulation. For instance, Row 42 has been obtained by applying paramodulation to the assertion that a mother is a parent,

```
(parent (mother ?person) ?person),
```

and the assertion that the mother of Carol is Betty,

```
(= (mother carol) betty).
```

The variable `?person` has been unified with `carol`.

The final step, which actually combines two resolution steps, will be explained in a later discussion on rewrite rules (Section 5.6.2).

4 Answer Management

We have seen (in Section 2.5) how answers to queries may be extracted from proofs. In this section, we consider some of the nuances of answer extraction—how to obtain multiple answers and conditional answers, and how to restrict SNARK to avoid unwanted answers.

4.1 Multiple Answers

In Section 2.5 we saw how to use SNARK to answer the query "Who is Carol's grandfather?" obtaining a single answer, such as

```
(father (father carol))
```

Of course, Carol has more than one grandparent. To find the others, we may reinvoke SNARK to find a different proof. For this purpose, we execute the function

```
(closure).
```

This is the function that computes computes logical consequences of the current set of rows, stopping when SNARK finds a new proof. In calling `prove`, we have implicitly been invoking `closure` on the initial set of rows. If we call `closure` after a proof has been interrupted, it will pick up where it left off and try to complete the proof, as we mentioned in Section 2.3. But if we call `closure` after a proof has been completed, SNARK will try to find a different proof.

Since the example was introduced in the unsorted version of the family theory, we continue in that version here. SNARK finds a second proof, which differs from the first only at the last step:

```
(Refutation
 ...
(Row mother-is-parent
   (parent (mother ?x) ?x)
   assertion)
 ...
(Row 23
   (not (parent ?x carol))
   (resolve 12 father-is-parent)
   Answer (ans (father ?x)))
(Row 31
   false
   (resolve 23 mother-is-parent)
   Answer (ans (father (mother carol))))
)
```

In the last step, we have used the assertion that a mother is a parent—in the first proof, we used the assertion that a father is a parent instead.

This new proof gives us a different answer: the father of the mother of Carol. Repeated invocation of SNARK via `closure` gives another two answers: `(mother (father carol))` and `(mother (mother carol))`. We have thus found four grandparents for Carol.

There is no guarantee, however, that a different proof will yield a different answer. If we reinvoke SNARK once more, we find yet another proof, but the answer, `(father (father carol))`, is the same as the first one we discovered. Reinvoking SNARK again yields more duplicate answers.

4.2 Avoiding Duplicate Answers

If we want to avoid this sort of duplication, we may rely on SNARK's subsumption mechanism. Subsumption is a strategic control mechanism which avoids duplication of effort. If the subsumption strategy is in operation (and it is the default) and two formulas are derived such that one is logically more general than the other, the less general formula is discarded. In the clausal case, this happens when the subsumed clause contains an instance of the subsuming clause among its disjuncts.

For example, if we have derived the two clauses

```
(parent ?x carol)
```

and

```
(or (parent (father ?y) carol)
    (grandparent ?w ?z)),
```

the latter ("subsumed") clause will be discarded; only the former clause will be retained. This is because one of the disjuncts of the latter clause, `(parent (father ?y) carol)` is an instance of the former; any proof that uses the latter clause will correspond directly to a proof that uses the former, so there is no need to retain both.

If one clause subsumes another and both have answer formulas, we cannot discard the subsumed clause without risking loss of answers; a proof using the subsumed clause may yield a different answer from a proof using the subsuming clause. For example, if we pay no attention to answers, a row

```
(Row 12
   (or (not (parent ?x ?y)) (not (parent ?y carol)))
   . . .
   Answer (ans ?x))
```

is subsumed (as SNARK observes) by the row

```
(Row 23
   (not (parent ?x carol))
   (resolve 12 father-is-parent)
   Answer (ans (father ?x)))
; Subsumed 12
```

However, if we discard Row 12, we may be losing proofs which yield answers other than `(father ?x)`; Row 23 may be more general but its answer is not.

If we want to get multiple answers for our query, we must be sure that the option `use-answers-during-subsumption` is selected; in fact, this is the default. In this case, the subsumed clause will be discarded only if its answer is also a special case of the answer associated with the subsuming clause. In the preceding example, if answers are used during subsumption, Row 23 will not be regarded as subsuming Row 12.

SNARK does not normally do subsumption on the final formula `false`; this formula would subsume all the formulas derived previously or subsequently. However, if we are interested in extracting multiple answers from proofs without duplicating answers, we must select the option `use-subsumption-by-false`; this option is not selected by default. Then the final `false` formula of the first proof will subsume all derived rows whose answer formula is an instance of (or identical to) the answer already found; formulas with different answers will be retained.

If we select the option `use-subsumption-by-false` for the problem of finding Carol's grandparents, the final row of the first proof,

```
(Row 30
   false
   ...
   Answer (ans (father (father carol))))
```

will subsume any row with the answer `(father (father carol))`. All these rows will be discarded. However, rows that contain distinct answers, such as `?x`, `(father ?x)`, and `(father (mother ?x))`, will be retained. If we invoke SNARK four times, we get the four different grandparents of Carol, as before. If we have not introduced any assertions other than the ones presented in Section 2.5, we exhaust the agenda with the fifth invocation of SNARK. All the duplicate answers have been subsumed.

On the other hand, if we have introduced the assertion

```
(assert
 '(implies
   (grandparent ?x ?z)
   (and (parent ?x (link ?x ?z))
        (parent (link ?x ?z) ?z)))
        :name 'grandparent-is-parent-of-parent)
```

as in Section 3.3, we will obtain an endless stream of additional answers, such as

```
(father (link (father (father carol)) carol))
```

This is another way of describing Carol's paternal grandfather,

```
(father (father carol)).
```

It is not a new answer, just a different representation for one of the answers we have seen previously. We shall call this sort of repetition of answers *semantic* duplication, to distinguish it from *syntactic* duplication, in which literally the same answer appears more than once.

Of course, we could simply remove the assertion `grandparent-is-parent-of-parent`, but it may be necessary for some other proofs. In Section 4.3, which follows, we shall see a mechanism for avoiding this sort of duplication.

4.3 Constructive Answer Restriction

There is no way of eliminating semantic duplication of answers altogether, but SNARK's "constructive-answer restriction" mechanism is one way of reducing semantic duplication. More important, it ensures that answers are provided in terms of a useful vocabulary. Let us see how this mechanism operates.

Suppose, in the theory we have been developing, we have provided the assertion that Carol's mother is Betty, i. e.,

```
(assert '(= (mother carol) betty))
```

If we want to ask who is Carol's mother, our query is then

```
(prove '(= ?person (mother carol)) :answer '(ans ?person))
```

As it stands now, we will get two answers, `betty` and `(mother carol)`. The second answer is correct but unhelpful—if we ask "Who is the mother of Carol?", we are not happy to be told "the mother of Carol," even though the mother of Carol is certainly the mother of Carol. One way to avoid such unhelpful answers is to indicate that the answer must be expressed without mentioning the function symbol `mother`. To do this, we select the option `use-constructive-answer-restriction` (as in Section 2.3) and, after initialization but before beginning the proof, we make the declaration

```
(declare-function-symbol 'mother 1 :allowed-in-answer nil)
```

This means that we will reject all rows that contain the function symbol `mother` in the answer formula. If we like, we can combine this declaration with the sort declaration:

```
(declare-function-symbol 'mother 1 :sort '(woman person)
 :allowed-in-answer nil)
```

If we ask who is the mother of Carol with the above declarations, we get only the single answer `betty`. The proof-steps that lead to the answer `(mother carol)` are rejected because their answer contain the forbidden symbol `mother`.

The example in Section 4.2 affords another example of the use of the constructive answer restriction to avoid semantically duplicated answers. In finding the grandparents of Carol, we obtain an endless stream of answers, such as

```
(father (link (father (father carol)) carol)),
```

in terms of the function `link`. If we use the constructive answer restriction and prohibit the function symbol `link` to appear in an answer, we obtain only the four distinct grandparents of Carol.

Constants and predicate symbols may also be declared to be disallowed in the answer, although the reason for disallowing a predicate symbol in the answer will not become apparent until we discuss conditional answers (Section 4.5). If a symbol has no declaration, the default is that it is permitted in the answer.

Whether a symbol is allowed in the answer or not is a context-dependent decision. If we were to ask who a person's grandparents are, we might be perfectly happy to learn that their mother's mother is one of them; in that case, we would not declare that the symbol `mother` is not allowed in the answer.

Exercise: Riddle. There is an old riddle

> Brothers and sisters have I none,
> But this man's father is my father's son.
>
> Who am I?

Using the vocabulary we have already introduced, formulate this as a SNARK problem, add any missing knowledge about family relations as assertions, and use SNARK to solve the riddle.

Hint. Define two people to be siblings if they are distinct and have at least one parent in common.

Phrase "Brothers and sisters have I none" as an assertion "I have no siblings." (Declare `i` to be a constant that is not allowed in the answer.)

Phrase "This man's father is my father's son" as an assertion "My father is a parent of this man's." (Declare `this-man` to be a constant symbol of sort `man`.)

Express as an assertion the fact that "The male parent is the father."

Phrase "Who am I?" as the conclusion to be proved, "I am equal to ?person1", where ?person1 is the answer.

Solution. Here are the declarations of `i` and `this-man`:

```
(declare-constant-symbol 'i
                         :allowed-in-answer nil
                         :sort 'person)
(declare-constant-symbol 'this-man :sort 'man)
```

The first verse "Brothers and sisters have I none" is expressed by the assertion

```
(assert '(not (sibling i ?person))
        :name 'brothers-and-sisters-have-i-none)
```

The second verse, "This man's father is my father's son" is expressed by the assertion

```
(assert '(parent (father i) (father this-man))
        :name 'this-mans-father-is-my-fathers-son)
```

The fact that a male parent is the father is expressed by the assertion

```
(assert
 '(implies
   (parent ?man ?person)
   (= ?man (father ?person)))
 :name 'male-parent-is-father)
```

The definition of sibling is given by

```
(assert
 '(iff
   (sibling ?person1 ?person2)
   (and
    (not (= ?person1 ?person2))
    (exists (?person)
         (and
          (parent ?person ?person1)
          (parent ?person ?person2)))))
 :name 'siblings-share-a-parent)
```

Note that is is necessary to state explicitly that siblings are distinct; even though I have the same parents as myself, I am not my own sibling. Note also that the solution doesn't work if we insist that siblings means full siblings—we must include half siblings.

The question is phrased as the conjecture

```
(prove '(= i ?person1)
       :name 'who-am-i
       :answer '(ans ?person1))
```

Note that if we hadn't declared `i` to be disallowed in the answer, we could have obtained `i` as the answer—correct but not helpful.

Remark: Skolem functions in answers. In the preceding riddle exercise (Section 4.3), suppose we had included the assertion `grandparent-iff-parent-of-parent`, with the explicit existential quantifier. If we invoked SNARK again, we could have obtained the additional answer

```
(#:person-sk1 (father (father this-man)) this-man)
```

In other words, I am the person who is the parent of this man and the child of his grandfather. Another answer we can obtain by a further invocation of SNARK is

```
(#:person-sk1 (mother (father this-man)) this-man).
```

Here `#:person-sk1` is the skolem function that resulted when SNARK removed the existential quantifier from the assertion. These answers are semantic duplicates—they are both equal to `(father this-man)`.

If we prefer that a particular skolem function not be allowed in the answer, we may mark the variable of the quantifier in the same way that we would mark a constant, function, or predicate symbol. For instance, if we replaced the existential quantifier `exists (?person)` of the assertion `grandparent-iff-parent-of-parent` with

```
exists ((?person :allowed-in-answer nil))
```

we would obtain only the first answer `(father this-man)`, not the other two, in answering the riddle. (Note again that the double parentheses are required.) On there other hand, there will be times when we do want to have skolem functions in answers; see, for example the problem of the red-headed grandmother, in the forthcoming Section 4.5.

Exercise: Cousins. Introduce assertions that define notions of `siblings` and `cousins`. These should express the facts that siblings have at least one parent in common, as in the riddle exercise, Section 4.3) and that cousins are children of siblings. (We consider only first cousins here.) Use SNARK to prove that cousins have a grandfather in common.

Solution. The definition of `sibling` is given by the assertion

```
(assert
 '(iff
   (sibling ?person1 ?person2)
   (and
    (not (= ?person1 ?person2))
    (exists (?person)
         (and
          (parent ?person ?person1)
          (parent ?person ?person2)))))
 :name 'siblings-share-a-parent).
```

The definition of `cousin` is given by the assertion

```
(assert
 '(iff
   (cousin ?person1 ?person2)
   (exists (?person3 ?person4)
           (and
            (parent ?person3 ?person1)
            (parent ?person4 ?person2)
            (sibling ?person3 ?person4))))
 :name 'cousins-are-children-of-siblings)
```

The conclusion to prove is

```
(prove
 '(forall (?person1 ?person2)
       (implies
        (cousin ?person1 ?person2)
        (exists (?person)
                (and
                 (grandparent ?person ?person1)
                 (grandparent ?person ?person2)))))
 :name 'cousins-share-a-grandparent)
```

Note that we cannot omit the existential quantifier

```
(exists (?person)...),
```

because it is within the scope of the universal quantifier

```
(forall (?person1 ?person2)...).
```

Omitting the explicit existential quantifier would create an invisible existential quantifier that would surround the entire query—the order of the quantifiers would be reversed and the meaning of the query would be changed. We would be trying to prove the existence of a single person who is a common grandparent of all pairs of cousins, but no such person exists.

4.4 Existentially Quantified Variables in Answers

Normally the answer field contains question-mark variables that are unquantified in the query. Although these variables have tacit existential quantification, they are not inside the scope of any explicit quantifier.

Sometimes, however, we would like to have the variable of an explicit existential quantifier in a query appear in an answer. A special mechanism is required to do this, because the answer formula is not within the scope of the existential quantifier.

For instance, in the Cousins exercise (Section 4.3), suppose we would like to find the cousins' common grandparent, as well as to prove that this person exists. We might be tempted to phrase the query like this:

```
(prove
 '(forall (?person1 ?person2)
       (implies
        (cousin ?person1 ?person2)
        (exists (?person)
                (and
                 (grandparent ?person ?person1)
                 (grandparent ?person ?person2)))))
 :name 'cousins-share-a-grandparent
 :answer '(ans ?person))
```

However, because the variable `person` in the answer is outside the scope of the existential quantifier, it does not refer to the same person.

To remedy this, we may mark the variable as `global`, by writing it as `(?person :global t)`. The query is then phrased as

```
(prove
 '(forall (?person1 ?person2)
       (implies
        (cousin ?person1 ?person2)
        (exists ((?person :global t))
                (and
                 (grandparent ?person ?person1)
                 (grandparent ?person ?person2)))))
 :name 'cousins-share-a-grandparent
 :answer '(ans ?person))
```

This tells SNARK that the variable `?person` refers to the same thing throughout the row, whether it is inside or outside the scope of the existential quantifier.

From this query, we obtain the answer

```
(ans
 (#:common-parent1
  (#:a-parent2 #:person-sk8 #:person-sk9)
  (#:b-parent3 #:person-sk8 #:person-sk9)))
```

Here we have used the `:conc-name` feature to give names to the skolem functions in the assertions. Thus `#:common-parent1` is the common parent of two siblings; we have rephrased the assertion as

```
(assert
 '(iff
   (sibling ?person1 ?person2)
   (and
    (not (= ?person1 ?person2))
    (exists ((?person :conc-name common-parent))
         (and
          (parent ?person ?person1)
          (parent ?person ?person2)))))
 :name 'siblings-share-a-parent)
```

Similarly, `#:a-parent2` and `#:b-parent3` are the siblings who are parents of two cousins, in the assertion `cousins-share-a-grandparent`. The skolem constants `#:person-sk8` and `#:person-sk9` are the two given cousins.

4.5 Conditional Answers

Sometimes it is not possible to find a single definite answer to a query, but it can be shown that one of several entities will satisfy the query, depending on contingencies. In this case, SNARK can produce a conditional answer. This is particularly useful if we are using SNARK to construct a program, because it allows for the introduction of conditional expressions, or tests, into the program being constructed.

For example[4], in the theory of family relations, suppose we are told that Alice is a grandparent of Carol, that Alice is red-headed, but that Carol is

[4]This is a reformulation of a blocks-world problem of Robert C. Moore.

not red-headed. We would like to find a pair `?person1` and `?person2` such that `?person1` is a parent of `?person2`, where `?person1` is red-headed but `?person2` is not.

Recall that (in Section 3.3) we introduced an assertion that a grandparent is always the parent of a parent, or, more precisely,

```
(assert
 '(implies
   (grandparent ?person1 ?person2)
   (and (parent ?person1 (link ?person1 ?person2))
        (parent (link ?person1 ?person2) ?person2)))
 :name 'grandparent-is-parent-of-parent)
```

This version of the assertion uses the explicit function `link`; later we shall consider what happens if we use the version with the existential quantifier.

The assertion tells us that if Alice is a grandparent of Carol, there is a link between them, a child of Alice who is a parent of Carol. The answer to our query then depends on whether this link is red-headed or not. If so, the link and Carol satisfy the query, because the link is red-headed but Carol is not. Otherwise, Alice and the link satisfy the query, because Alice is red-headed and the link is not. Thus, a conditional answer is called for.

To construct conditional answers, we must select the SNARK option `use-conditional-answer-creation` (See Section 2.3); it is not the default. We introduce the assertions that specify the situation:

```
(assert '(grandparent alice carol)
        :name 'alice-is-grandparent-of-carol)

(assert '(red-headed alice)
        :name 'alice-is-red-headed)

(assert '(not (red-headed carol))
        :name 'carol-is-not-red-headed)
```

Then we pose the query:

```
(prove '(and
         (parent ?person1 ?person2)
         (red-headed ?person1)
         (not (red-headed ?person2)))
```

```
        :name 'red-headed-parent
        :answer '(parent ?person1 ?person2))
```

SNARK proves the conclusion and yields that answer

```
(answer-if
 (red-headed (link alice carol))
 (parent (link alice carol) carol)
 (parent alice (link alice carol)))
```

In other words, if the link between Alice and Carol is red-headed, then the link and Carol are the desired pair; otherwise, Alice and the link are the pair. This agrees with our informal reasoning.

If we fail to invoke `use-conditional-answer-creation`, SNARK will not form a conditional answer. Instead it will form a disjunction of the instances of the answer formula that satisfy the query, without indicating which instance holds in which situation. In this case, it will produce the answer

```
(or
 (parent (link alice carol) carol)
 (parent alice (link alice carol))))
```

5 Efficiency Considerations

In this section we discuss several mechanisms for improving SNARK's performance, including associative and commutative unification (Section 5.1), the set-of-support strategy (Section 5.2), the recursive-path and predicate ordering strategies (Sections 5.3 and 5.4), and rewrite rules (Section 5.6).

When appropriate options are selected, SNARK is a logically complete theorem prover; in other words, if a conclusion follows from the assertions, SNARK, equipped with the appropriate inference rules, will eventually find a proof. However, as the length of the proof or the number of assertions grows, the truth of the preceding sentence relies more and more on the word "eventually".

SNARK's strategic mechanisms allow us to tune its performance for a particular subject domain or theory. If used properly, they can have a dramatic effect on the size of the search space and on the time necessary to discover a proof. Whereas none of them is necessary for the examples in this guide, their use is essential if we are dealing with a large theory or searching

for a long or difficult proof. However, some care is required in the use of control strategies. Subtle misuse of rewrite rules, for example, can lead to incompleteness or even infinite looping. While the set-of-support and the recursive-path ordering strategies retain completeness when used separately, the two can cause loss of completeness when used together. Any of them may be incompatible with hyperresoluton or the constructive answer restriction. While it may be justifiable to sacrifice completeness for the sake of improved performance, the user should be aware that if SNARK fails to find a proof, the cause may be unforeseen interactions with control strategies. In such a case, we can experiment with disengaging the strategies.

5.1 Commutative and Associative Symbols

SNARK has special-purpose unification algorithms that allow us to reason about symbols with special properties, without introducing assertions that express these properties.

For example, the `sibling` relation we introduced in the riddle exercise (Section 4.3) is commutative: if Arthur and Susan are siblings, then so are Susan and Arthur. The order is irrelevant.

We could express this property by an assertion

```
(assert
 '(iff
   (sibling ?person1 ?person2)
   (sibling ?person2 ?person1))
 :name 'symmetry-of-sibling).
```

(It would suffice to use `implies` rather than `iff` here.) However, that assertion would have many consequences; the resolution rule applies to it and any formula that mentions the `sibling` relation. Most of these consequences would be irrelevant to the problem at hand, and each of them would have its own consequences in turn, and so on. We could be swamped by the proliferation of irrelevant clauses.

Instead we choose to declare that `sibling` is a commutative predicate, and drop the assertion. For this purpose, we include the declaration

```
(declare-predicate-symbol 'sibling 2 :commutative t)
```

anywhere after initialization and before use of the symbol. This means that the commutative unification algorithm will be used when we attempt to

unify two atoms with the predicate symbol `sibling`. Although commutative unification is slower than ordinary unification, its employment here prevents SNARK from generating lots of unnecessary consequences.

We can declare function symbols as well as predicate symbols to be commutative. Function symbols can also be declared to be associative. For associativity, we use the keyword `:associative`.

If SNARK sees an assertion that is an instance of associativity or commutativity, it will remove the assertion automatically and instead declare the appropriate symbol to be associative or commutative. Thus, if we did chose to include the assertion `symmetry-of-sibling` as an assertion, SNARK would simply remove the assertion and declare the relation `sibling` to be commutative.

5.2 Set of Support

The set-of-support strategy causes SNARK to pay special attention to a particular subset of the rows derived in searching for a proof. Typically this *set of support* includes the desired (negated) conclusion, and often it includes special hypotheses required for that conclusion to hold, but it may exclude all the other assertions. In observing the strategy, we only apply a rule of inference to a set of rows if at least one of the rows is *supported*, i. e., belongs to the set of support. The newly inferred row is then added to the set of support.

If we have a large set of assertions and the set of support is relatively small, applying the strategy may allow us to ignore most of the assertions and help us focus our attention on inferences that are related to the desired conclusion.

Selecting the set of support strategy is a bit different from the selection of other SNARK options. The strategy is always turned on—there is no way to turn it off—but initially every row is included in the set of support, which means that the strategy has no effect; no inferences are excluded.

To actually benefit from the strategy, we must indicate some rows to be excluded. Normally, we choose the option

```
(assert-supported nil)
```

to exclude all the assertions from the set of support. The option must be selected before any assertions are made. Inferences can still be made from these assertions, but only if some member of the set of support takes part

in the inference step. For instance, we can apply the resolution rule to any assertion and a member of the set of support, but not to two assertions.

The negation of the conclusion (obtained from the argument of the `prove` function) will normally be included in the set of support. If there are certain special assertions that we wish to to include in the set of support, even if assertions are generally excluded, we can assert them by saying (for a formula `<Form>`)

```
(assert <Form> :supported t)
```

This will override the general policy that assertions are not to be supported.

For instance, if we execute

```
(assert
  (implies
   (and
    (parent ?person1 ?person2)
    (parent ?person2 ?person3))
   (grandparent ?person1 ?person3))
  :name 'parent-of-parent-is-grandparent
  :supported t)
```

then this assertion will be supported. We might do this because we want believe this assertion will be relevant to the proof, or class of proofs, we are seeking.

If there are two unsupported assertions,

```
(parent alice betty)
```

and

```
(parent betty carol),
```

we can still use resolution and the assertion `parent-of-parent-is-grandparent` to deduce

```
(grandparent alice carol).
```

If none of these assertions were supported, the inference would be illegal.

Normally a formula that is deduced from a supported formula is also supported; in other words, being supported is *inherited*. For example, the

formula `(grandparent alice carol)` we just deduced will be supported, even though the two facts it was deduced from, `(parent alice betty)` and `(parent betty carol)`, are not. We can declare that a formula is to be supported but that its supportedness is not to be inherited by tagging it with the keywords

```
:supported :uninherited
```

instead of

```
:supported t
```

If the assertion `parent-of-parent-is-grandparent` were so tagged, the new deduced formula would not be supported.

Another way to force a particular assertion `<Form>` to belong to the set of support is to execute

```
(assume <Form>)
```

in place of

```
(assert <Form>).
```

The construct `assume` is synonymous with `assert`, but when we exclude assertions made via `assert` from the set of support by saying

```
(assert-supported nil),
```

we have no effect on those assertions made via `assume`; they will still be supported. Typically, these *assumptions* are special hypotheses for the theorem being proved, and hence are particularly likely to be relevant to the proof.

If we choose to, we can even exclude assertions made via `assume` from the set of support by saying

```
(assume-supported nil).
```

This is unusual because the point of having two synonymous constructs, assertions and assumptions, is that we can support assumptions while excluding ordinary assertions. We can even exclude the negation of the conclusion from the set of support by saying

```
(prove-supported nil).
```

The reason for doing this sort of thing is to explore the effect of different search strategies on the same problem.

In some theorem provers and other rule-based systems, a distinction is made between forward- and backward chaining rules. Forward-chaining rules allow us to reason forward from assertions; backward-chaining rules allow us to reason backward from the desired conclusion. The support mechanism in SNARK allows us to mimic the behavior of forward and backward chaining. When we applied the assertion `parent-of-parent-is-grandparent` to two other assertions, we were using it as a forward-chaining rule. In general, forward-chaining behavior is obtained by supporting assertions and assumptions; backward-chaining behavior is obtained by supporting the desired conclusion.

The set-of-support strategy is complete if the complement of the set of support is satisfiable, that is, if the complement contains no contradictory formulas. For instance, if all of our assertions are consistent (satisfiable), the strategy will be complete if only the desired conclusion is supported. Caution must be exerted in using the strategy, however, because it is incompatible with other strategies, as mentioned in the introduction to this section (Section 5). The default in SNARK is that all assertions, assumptions, and desired conclusions are supported—in other words, the set-of-support strategy does not impede any inference. This is a conservative choice—completeness will not be lost because of incompatibilities between set-of-support and other strategies—but for some problems it may not be the most efficacious decision.

The value of the set-of-support strategy is illustrated by the proof for the riddle exercise of Section 4.3, which puts 218 rows on the agenda if the default settings are used, but only 50 rows if assertions are excluded from the set of support and only the negation of the conclusion is included; the time required is less than half. Exact figures depend on how many assertions are in the theory, of course. The more irrelevant assertions in the theory, the more important the strategy becomes.

5.3 Recursive-Path Ordering Strategy

SNARK has a number of mechanisms by which we can impose an ordering on the symbols in our vocabulary, which can give SNARK a sense of direction in searching for a proof. These orderings say that certain expressions are preferable to certain others, and will restrict the action of SNARK's rules so that they will tend to remove the less favored expressions. We discuss

two ordering strategies: the recursive-path ordering strategy, which restricts application of the paramodulation rule, and the predicate ordering strategy, which restricts application of the resolution rule. Both of these are referred to as symbol-ordering strategies.

The recursive-path ordering strategy uses an ordering on the constant and function symbols of our vocabulary. It is invoked by

```
(use-term-ordering :recursive-path).
```

The format of this command is unusual because SNARK also allows other such "term-ordering" strategies, which may be stipulated here instead.

As we have observed, the paramodulation rule can be applied in two directions; if the equality is `(= s t)`, it can be used left-to-right to replace `s` with `t` or right-to-left to replace `t` with `s`. If we do not select an ordering strategy, SNARK will attempt applying the rule in both directions, with some redundancy. If we employ the recursive-path ordering strategy, SNARK can often avoid one of these two directions and hence reduce the branchiness of the search space.

The recursive-path ordering strategy compares `s` and `t` according to an ordering before allowing a replacement. If `s` is greater than `t`, it will not allow us to replace `t` with `s`; we will only be permitted to apply paramodulation in left-to-right order. If `t` is greater than `s`, we will only be allowed to apply the rule right-to-left. In either of these cases, we say that the equality can be *ordered*. Otherwise, if no order exists between `s` and `t`, the strategy will allow us to apply the rule in either direction. For example, the recursive-path ordering will not allow us to order a commutativity axiom, such as

```
(= (f ?x ?y) (f ?y ?x)).
```

As we have seen (Section 5.1), SNARK deals with associativity and commutativity, in particular, by invoking a special-purpose unification algorithm, but other equalities may be unorderable as well.

To define the ordering on terms, we declare an ordering on the constant and function symbols. For instance, suppose we prefer that when we have assertions such as

```
(assert '(= (mother carol) betty))
```

we always wish to replace the constant `betty` with the term `(mother carol)` so that we can use properties of the function `mother`. If so, we can declare the orderings

```
(declare-ordering-greaterp 'betty 'mother)
(declare-ordering-greaterp 'betty 'carol)
```

With this declaration, the equality will be applied in the right-to-left direction.

The precise definition of recursive-path ordering is complicated—see, for example, [Dershowitz]—but to order a ground (variable-less) equality it suffices that all the constant and function symbols on one side precede all the constant and function symbols on the other.

If (for ordering other equalities) we want to say that `carol` is also to be preferred to `mother`, we can do it in a single declaration,

```
(declare-ordering-greaterp 'betty 'carol 'mother).
```

The ordering we provide on symbols must be loop-free, and every symbol must previously be declared—otherwise SNARK will give an error message. Also, the name must be unambiguous. If we have introduced a constant `betty` and a function symbol `betty`, SNARK will not know which one we mean and will give an error message. For this purpose, we should use the alias mechanism (Section 3.2) to give the two `betty`'s distinct aliases, e. g., `betty-con` and `betty-fun`. The declaration can then refer unambiguously to `betty-con`.

5.4 Predicate Ordering Strategy

The predicate ordering strategy, which controls the resolution rule, uses an ordering on the predicate symbols of our vocabulary and extends the ordering on terms to apply to atomic formulas. It is selected by `use-literal-ordering-with-resolution`. (There is also a version of the restriction that applies to the paramodulation rule.)

Let us give an example: consider the exercise (Section 4.3) in which we showed that cousins have a common grandparent. Let us see how to introduce an ordering on the predicate symbols.

We use four predicate symbols here, `parent`, `sibling`, `cousin`, and `grandparent`. Note that `grandparent` and `sibling` are defined in terms of `parent`, and `cousin` is defined in terms of `sibling` and `parent`. Therefore, it makes sense during a proof to paraphrase `cousin` in terms of `sibling` and `parent`. Then `grandparent` and `sibling` can be paraphrased in terms of `parent`.

For example, we include the declaration

```
(declare-predicate-symbol 'parent 2).
```

We can then include the ordering

```
(declare-ordering-greaterp 'cousin 'sibling 'parent)
(declare-ordering-greaterp 'grandparent 'parent)
```

This means that we will favor inferences that replace `cousin` with `sibling` or `parent`, that replace `sibling` with `parent`, or that replace `grandparent` with `parent`. (We do not say whether we prefer `cousin` or `sibling` to `grandparent`, since we have no opinion on that at this time.)

In proving this result without a symbol ordering, SNARK may generate more than a thousand rows before discovering a proof. Selecting this ordering will allow SNARK to find the proof after generating fewer than a hundred rows. The time required was about about a twentieth of the time without the ordering strategy.

In comparing two formulas which have the same predicate symbol, the ordering restriction will use the recursive-path ordering to compare the argument terms of the two predicate symbols. It is possible to control the order in which the arguments are compared, but we shall not discuss this technicality here.

If we want to use an ordering strategy but want SNARK to invent an ordering for us, we can select the option `use-default-ordering`. SNARK's ordering will agree with any orderings we have chosen, but SNARK will fill in its own ordering when we have not made any decision. For example, since we have not decided whether we prefer `sibling` or `grandparent`, SNARK will decide one way or another, according to its own criteria. Even if we do not indicate any ordering at all, we may get improved performance if we use the ordering strategy and the default ordering, because the search space will be narrower: fewer inferences are legal at each step.

These ordering restrictions must be used with some care. They are logically complete in isolation, but they are not in general complete when used in combination with other strategies, or even with the constructive answer restriction.

5.5 Obtaining Left-to-Right Behavior

SNARK inference rules can operate on any atom in a formula. Logic-programming languages, such as PROLOG, on the other hand, commonly operate on a formula in left-to-right order. This behavior can be mimicked in SNARK by

tagging a formula to be *sequential*. If a formula is sequential, only its leftmost atom is available to be operated on.

We can indicate that a formula is to be processed sequentially by a keyword argument, as in

```
(assert
  (implied-by
   (grandparent ?person1 ?person3)
   (and
    (parent ?person1 ?person2)
    (parent ?person2 ?person3)))
   :sequential t).
```

This means that to apply the resolution rule, say, to this formula, we must unify the leftmost atom,

```
(grandparent ?person1 ?person3),
```

rather than either of the two subsequent atoms. If we do succeed in deriving a consequence from this formula, the derived formula is also sequential. In particular, the instance of the second atom

```
(parent ?person1 ?person2)
```

must be operated on before the instance of the third atom

```
(parent ?person2 ?person3)
```

can be. If we do not want formulas derived from a sequential formula to be inherited, we can tag it

```
:sequential :uninherited
```

instead of `t`.

If the keyword `:sequential` is not specified, whether a formula is to be treated sequentially is determined by the current value of `assert-sequential`, `assume-sequential`, or `prove-sequential`. For instance, if we declare

```
(assert-sequential t),
```

all assertions will be sequential. SNARK's default value is `nil` for all these options.

To approximate the behavior of PROLOG—backward chaining and left-to-right solution of goals, we declare all assertions to be sequential and unsupported and we specify

```
(prove goal :supported t :sequential t).
```

Because sequentiality is inherited, all the formulas we derive will then be treated in left-to-right order, as in PROLOG.

Like the support restriction, the sequentiality restriction must be used with care to preserve completeness; it is only complete in special cases. When using sequentiality, the user is viewing the assertions as programs, to be executed in order. Predicate-ordering strategies provide a safer way of indicating which atoms of a formula to operate on first.

5.6 Rewrite Rules

SNARK has a rewrite rule mechanism that allows us to treat certain designated equalities or equivalences as rewrite rules. This means that whenever an occurrence of the left side of the rule appears in a row (henceforth called the *target*), it will be replaced by the corresponding instance of the right side of the rule, immediately and automatically.

For example, in Section 3.4, we used an assertion to define the grandparent relation:

```
(assert
 '(iff
   (grandparent ?person1 ?person2)
   (exists (?person)
           (and (parent ?person1 ?person)
                (parent ?person ?person2))))
 :name 'grandparent-iff-parent-of-parent)
```

If we chose to represent the same equivalence by a rewrite rule, we would instead say

```
(assert-rewrite
 '(iff
   (grandparent ?person1 ?person2)
```

```
(exists (?person)
        (and (parent ?person1 ?person)
             (parent ?person ?person2))))
     :name 'grandparent-iff-parent-of-parent)
```

There are several differences between these two statements in the way they are treated by SNARK. The rewrite rule will immediately replace any subformula of form `(grandparent ?person1 ?person2)`, in any SNARK row, with a corresponding formula

```
(and (parent ?person1 term) (parent term ?person2))
```

where `term` is either a variable ?person or a skolem term, depending on the force of the quantifier (see Section 3.3). The assertions and proof will appear as if the `grandparent` symbol did not exist.

Phrasing the statement as an assertion, on the other hand, will not interfere with SNARK's usual operation. Formulas involving the `grandparent` relation will be placed on the agenda to be processed in due course. Resolution with the assertion that defines the relation will not cause the symbol to be replaced—instead, new rows will be created and added to the agenda.

The fact that rewrite rules cause a replacement instead of an addition to the agenda means that they can drastically reduce the search space. In the problem of showing that cousins have a grandparent in common (Section 4.3), if we replace the assertions that define grandparents, cousins and siblings with rewrite rules, we obtain a proof in a quarter of the time, generating a third the number of rows.

Another difference between a rewriting and an ordinary inference is that SNARK will only do one-way matching, not full unification, in performing a rewriting. It will instantiate variables in the left side of the rule to force them to be identical to terms in the target formula, but it will not instantiate variables in the target formula to force them to be identical to terms in the rule.

For example, suppose we have an ordinary assertion (not a rewrite rule)

```
(assert '(= (mother carol) betty)).
```

If we also have in our theory the assertion

```
(assert '(parent (mother ?person) ?person)
        :name 'mother-is-parent),
```

Then we can apply paramodulation to the two assertions, instantiating `person` be `carol`, to obtain

```
(parent betty carol),
```

The instantiation was discovered by the unification algorithm, in unifying the left side of the equality, the term `(mother carol)`, with the subterm `(mother ?person)` of the target assertion, `mother-is-parent`.

Now suppose we rephrase the theory so that the fact that Carol's mother is Betty is expressed instead by the rewrite rule

```
(assert-rewrite '(= (mother carol) betty)).
```

Then we cannot apply the rewrite rule to the assertion `mother-is-parent`, because SNARK will not instantiate the variable `?x` in the assertion to create an instance of the left side of the rule.

5.6.1 Proceed with Caution

Rewrite rules can replace one formula with another, rather than merely adding a new row, because they do not instantiate variables in the target formula; this means that no proof that requires a different instantiation is being lost due to the replacement. However, caution must be exercised in using rewrite rules instead of assertions. Poor choice of a rewrite rule can lead to incompleteness—SNARK may fail to prove valid conclusions if a fact is represented as a rewrite rule rather than as an assertion.

For example, in the theory in which the fact that Betty is Carol's mother is represented by an assertion, SNARK can answer the query

```
(prove '(parent betty ?person) :answer '(ans ?person)
 :name 'who-is-bettys-child?).
```

The negation of the conclusion directly contradicts the assertion

```
(parent betty carol),
```

which was obtained by the paramodulation step, giving the answer `carol`.

But if the fact that Betty is Carol's mother is represented by a rewrite rule, we cannot deduce that Betty is a parent of Carol, and we cannot answer the query.

Another caution on the use of rewrite rules is that it is the user's responsibility to see that they terminate. If we chose to represent the symmetry of the sibling relation by a rewrite rule

```
(assert-rewrite
 '(iff (sibling ?person1 ?person2)
       (sibling ?person2 ?person1))
 :name 'symmetry-of-sibling)
```

we would have an infinite computation whenever we used the sibling relation.

Finally, even if the use of a rewrite rule does not lead to incompleteness or to an infinite loop, there is no guarantee that it will speed up the search. For instance, if a rewriting replaces a simple atomic formula with a complex sentence, it may even make some proofs more complex.

5.6.2 Rewrite Rules May Be Introduced Automatically

SNARK will sometimes introduce rewrite rules automatically, to increase its efficiency, when it observes a suitable opportunity. If we select the option `use-simplification-by-equalities`, SNARK will introduce rewrite rules from asserted or deduced equalities. For example, if we are using the recursive-path ordering strategy and deduce an equality that can be ordered by the strategy, SNARK will introduce a rewrite rule that will have the same effect.

SNARK, in contrast with a human being, will never introduce rewrite rules that lead to incompleteness or looping. Therefore, rather than introducing a rewrite by hand, it is usually preferable for a user to introduce an ordinary assertion and to provide a symbol ordering that will cause SNARK to introduce the rewrite automatically, if it is safe.

Even if we have not selected special options, if we have a unit assertion `P`, SNARK will introduce a rewrite rule

```
(assert-rewrite (iff P true)).
```

Application of the rewrite rule will produce some of the effects of the resolution rule; for instance, if we have a clause

```
(or (not P') Q),
```

where `P'` is an instance of `P`, applying the rule will (after simplification) yield the clause `Q`; this same clause would ultimately have been obtained by resolution with the unit clause `P`, if the rewrite rule had not been in place. But the rewrite rule will be applied immediately and automatically, when it becomes applicable as the result of applying other rules.

This explains why we have seen SNARK traces in which rewriting has been applied, even when we have not introduced any rewrite rules. For example, the last step of the proof in the Grandchildren-of-Alice example, Section 3.5, is a resolution followed by a rewrite that has the effect of a second resolution with a unit assertion—SNARK has effectively combined two resolution steps into one.

In cases in which SNARK invents a rewrite rule corresponding to an assertion, it will also leave the assertion in place, so that other rules, such as paramodulation and resolution, can be applied.

Exercise: Efficiency Experiment with the previous exercises to see if you can improve the time required or the number of rows produced by using the set-of-support strategy, symbol orderings, or rewrite rules.

6 Temporal Reasoning

Reasoning about time plays a central role in many knowledge-representation applications. SNARK's temporal representation supports two kinds of temporal entities, time points and time intervals, and relationships between them, and is an extension of the temporal interval logic of Allen [Allen]. It also supports times of day and calendar dates. Many of the relation names in SNARK's temporal representation were adapted from Cyc's upper ontology [Cyc-UL].

A possible approach to reason with time is to provide the axioms for the desired temporal inferences, and rely on SNARK's general-purpose inference methods, such as resolution and paramodulation, to draw conclusions. This approach can involve a good deal of time-consuming search. Alternatively, one may use special-purpose inference methods for reasoning with temporal knowledge. A special-purpose reasoner can significantly improve the speed of inference, but a suitable interface to the general-purpose inference procedure must be designed. SNARK supports a time and date procedure with an interface to the ordinary resolution rule, based on the *constraint resolution* framework [Burckert].

The SNARK temporal reasoning facility is switched on by selecting the option `use-temporal-reasoning`, e. g., by evaluating

```
(default-use-temporal-reasoning)
```

before initialization; henceforth, in this section, we assume that the facility has been turned on. When using temporal reasoning, special meaning is given to certain symbols, such as `time-point`; it is possible for the user to rename these symbols.

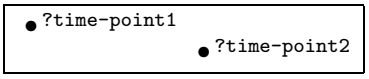
We begin with the relations that apply to time points; later we will consider time intervals and relations between time points and intervals.

6.1 Time Points

A time point is a single moment of time; these are of sort `time-point`. Terms of this sort can be treated like any other terms; they may be arguments of predicate and function symbols, for example. There are some built-in operations that are already declared on time points.

There are three primitive relations between pairs of time points:

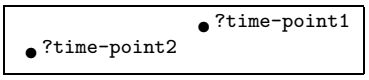
- `(before ?time-point1 ?time-point2)`:

  `?time-point1` is (strictly) earlier than `?time-point2`.

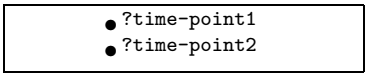
  (In general, we use the word *strict* to apply to relations that do not allow time points to occur simultaneously.) Here moving from left to right corresponds to the passing of time.

- `(after ?time-point1 ?time-point2)`:

  `?time-point1` is later than `?time-point2`.

- `(simultaneous-with ?time-point1 ?time-point2)`:

  `?time-point1` is simultaneous with `?time-point2`.

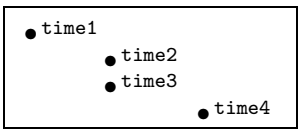
Note that these three relations are mutually exclusive and exhaustive: for any two time points, precisely one of these is true. Also note that the relation `simultaneous-with` is not the same as ordinary equality: one could have two time points that occur simultaneously but are different in other respects, e.g.,

one is the beginning of a new millennium and the other is the end of the old one. One of these time points could be regarded as happy, the other as sad, so they are not equal. Temporally, however, they are treated as identical. If one wants true equality, one should use instead the relation `=`, which is stronger than `simultaneous-with` and has the same effect in temporal reasoning.

Given a set of temporal relations between time points, SNARK will be able to draw conclusions from them. For example, suppose we are given four time points such that

```
(before time1 time2)
(simultaneous-with time2 time3)
(after time4 time3).
```

This situation can be depicted as follows:



Then the temporal reasoning component within SNARK will be able to conclude that

```
(before time1 time4).
```

SNARK is complete and efficient for this kind of reasoning; in other words, if a conclusion about time points follows from a set of facts asserted using only the above three temporal relations, SNARK will be able to deduce the conclusion.

6.2 Time Intervals

A time interval is a finite connected region of time. Time intervals are of sort `time-interval`, which is disjoint from `time-point`; that is, no entity is both a time point and a time interval.

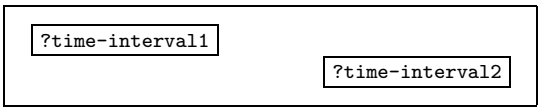
A time interval has end-points, which are time points, and it also has interior time points, which occur during the time interval. As in the Allen logic, we do not identify a time interval with a set of time points. The Allen logic is even agnostic about whether the end-points are part of the interval; that is not a meaningful question, because the original logic does not mention time points.

Each time interval has a start-point and an end-point, which are time points. Time intervals are always thought of as being nonempty, in the sense that the start-point is always strictly earlier than the end-point.

6.2.1 Allen Primitives

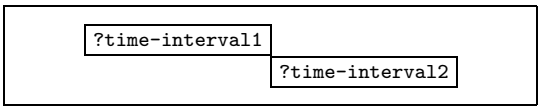
There are thirteen fundamental relations, known as the *Allen primitives*, between pairs of time intervals. We say that the relation is *strict* if it does not allow the start- or end-points to be identical.

- `(before ?time-interval1 ?time-interval2)`:

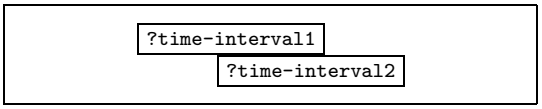
  The end-point of `?time-interval1` is (strictly) earlier than the start-point of `?time-interval2`.

- `(meets ?time-interval1 ?time-interval2)`:

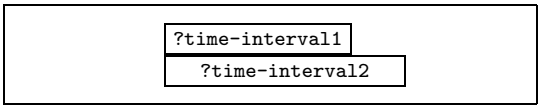
  The end-point of `?time-interval1` is simultaneous with the start-point of `?time-interval2`.

- `(overlaps ?time-interval1 ?time-interval2)`:

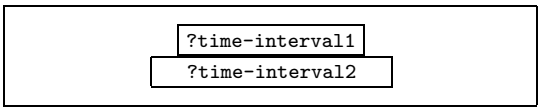
  The start-point of `?time-interval1` is earlier than the start-point of `?time-interval2`, but the end-point of `?time-interval1` is (strictly) between the start- and end-points of `?time-interval2`

- `(starts ?time-interval1 ?time-interval2)`:

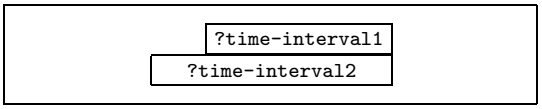
  The start-point of `?time-interval1` is simultaneous with the start-point of `?time-interval2`, but the end-point of `?time-interval1` is earlier than the end-point of `?time-interval2`.

- `(during ?time-interval1 ?time-interval2)`:



  The start-point of `?time-interval1` is later than the start-point of `?time-interval2`, but the end-point of `?time-interval1` is earlier than the end-point of `?time-interval2`.

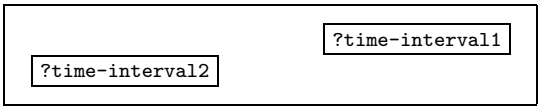
- `(finishes ?time-interval1 ?time-interval2)`:



  The end-point of `?time-interval1` is simultaneous with the end-point of `?time-interval2`, but the start-point of `?time-interval1` is later than the start-point of `?time-interval2`.

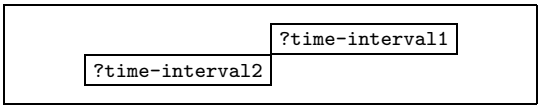
The next six Allen primitives are the inverses of the first six.

- `(after ?time-interval1 ?time-interval2)`:



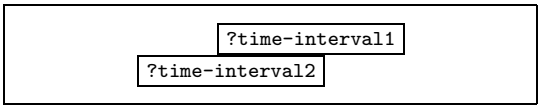
  The start-point of `?time-interval1` is later than the end-point of `?time-interval2`.

- `(met-by ?time-interval1 ?time-interval2)`:



  The start-point of `?time-interval1` is simultaneous with the end-point of `?time-interval2`.

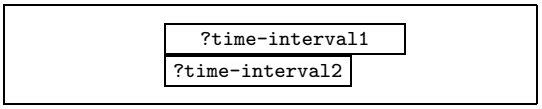
- `(overlapped-by ?time-interval1 ?time-interval2)`:



  The start-point of `?time-interval1` is between the start- and end-

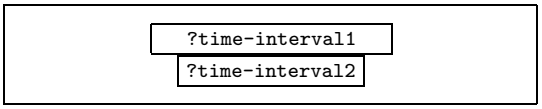
points of ?time-interval2, but the end-point of ?time-interval1 is later than the end-point of ?time-interval2.

- (started-by ?time-interval1 ?time-interval2):



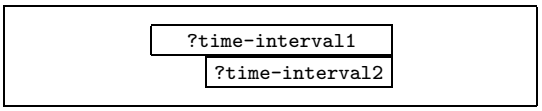
The start-point of ?time-interval1 is simultaneous with the start-point of ?time-interval2, but the end-point of ?time-interval1 is later than the end-point of ?time-interval2.

- (contains ?time-interval1 ?time-interval2):



The start-point of ?time-interval1 is earlier than the start-point of ?time-interval2, but the end-point of ?time-interval1 is later than the end-point of ?time-interval2.

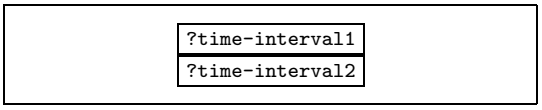
- (finished-by ?time-interval1 ?time-interval2):



The end-point of ?time-interval1 is simultaneous with the end-point of ?time-interval2, but the start-point of ?time-interval1 is earlier than the start-point of ?time-interval2.

The thirteenth Allen primitive is the equality relation for time intervals.

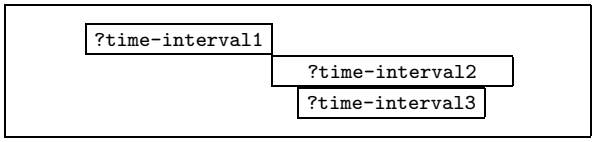
- (cotemporal ?time-interval1 ?time-interval2):



The start- and end-points of ?time-interval1 and ?time-interval2 are respectively simultaneous. The relation is also called temporal-bounds-identical.

As with the relations on time points, the Allen primitives are mutually exclusive and exhaustive: between any two time intervals, precisely one of these relations holds. Again, the relation `cotemporal` behaves like equality on time intervals for temporal reasoning, but it is not a true equality relation. In particular, it is possible for a non-temporal relation to be true for one time interval but false for another cotemporal one; two time-intervals can be cotemporal without being equal.

As with time points, SNARK can efficiently draw conclusions about time intervals from assertions expressed in terms of the Allen primitives. For example, suppose `time-interval1 meets time-interval2`, which in turn `contains time-interval3`. The situation is illustrated as follows:



Then SNARK can deduce that `time-interval1` is before `time-interval3`.

Note that if one time interval `meets` another, the end-point of the first interval must be `simultaneous-with` the start-point of the second. Because SNARK doesn't identify a time interval with a set of points, it is not meaningful to ask whether that time point is a member of either interval, or whether those intervals have a point in common.

6.2.2 Nonprimitive Relations

There are relations between time intervals other than the thirteen Allen primitives in SNARK. The ones listed here are defined as disjunctions of the Allen primitives, and hence are not logically necessary. However, they provide a useful and convenient abbreviation for commonly used notions; we used them in our HPKB work. The names are obtained from the CYC ontology [Lenat].

- `(temporally-subsumes ?time-interval1 ?time-interval2)`:

  Holds if `?time-interval2` is a (not necessarily strict) subinterval of `?time-interval1`, that is, if any of the following conditions holds:

```
(finished-by ?time-interval1 ?time-interval2)
(contains ?time-interval1 ?time-interval2)
(cotemporal ?time-interval1 ?time-interval2)
(started-by ?time-interval1 ?time-interval2)
```

In other words, it is defined as the disjunction of `finished-by`, `contains`, `cotemporal`, and `started-by`.

- `(temporally-subsumed-by ?time-interval1 ?time-interval2)`:

  The inverse of `temporally-subsumes`, it holds if `?time-interval1` is a subinterval of `?time-interval2`. It is defined as the disjunction of `starts`, `cotemporal`, `during`, and `finishes`, the inverses of the Allen primitives in the definition of `temporally-subsumes`.

- `(starts-during ?time-interval1 ?time-interval2)`:

  Holds if the start-point of `?time-interval1` is (strictly) during `?time-interval2`. It is defined as the disjunction of `during`, `finishes` and `overlapped-by`.

- `(ends-during ?time-interval1 ?time-interval2)`:

  Holds if the end-point of `?time-interval1` is during `?time-interval2`. If is defined as the disjunction of `overlaps`, `starts`, and `during`.

- `(starts-after-starting-of ?time-interval1 ?time-interval2)`:

  Holds if the start-point of `?time-interval1` is later than the start-point of `?time-interval2`. If is defined to be the disjunction of `during`, `finishes`, `overlapped-by`, `met-by`, and `after`.

- `(ends-after-ending-of ?time-interval1 ?time-interval2)`:

  Holds if the end-point of `?time-interval1` is later than the end-point of `?time-interval2`. It is defined to be the disjunction of `contains`, `started-by`, `overlapped-by`, `met-by`, and `after`.

- `(ends-after-starting-of ?time-interval1 ?time-interval2)`:

  Holds if the end-point of `?time-interval1` is later than the start-point of `?time-interval2`. It is defined to be the disjunction of all the Allen primitives except `before` and `meets`.

- `(temporally-cooriginating ?time-interval1 ?time-interval2)`:

  Holds if the start-points of `?time-interval1` and `?time-interval2` are simultaneous. It is defined to be the disjunction of `starts`, `cotemporal`, and `started-by`.

- `(temporally-coterminal ?time-interval1 ?time-interval2)`:

  Holds if the end-points of `?time-interval1` and `?time-interval2` are simultaneous. It is defined to be the disjunction of `finishes`, `cotemporal`, and `finished-by`.

- `(temporally-disjoint ?time-interval1 ?time-interval2)`:

  Holds if `?time-interval1` and `?time-interval2` have no interior points in common. It is defined to be the disjunction of the Allen primitives `before`, `meets`, `met-by`, and `after`.

- `(temporally-intersects ?time-interval1 ?time-interval2)`:

  Holds if `?time-interval1` and `?time-interval2` are not disjoint, i.e., if they have some interior points in common. It is defined to be the disjunction of all the Allen primitives except those in the definition of `disjoint`. It is also called `temporal-bounds-intersect`.

6.3 Intermixing Temporal and Relational Reasoning

While it may be interesting to infer facts about temporal points and intervals, the primary reason for including temporal reasoning in SNARK is to allow us to combine reasoning about temporal and other relations to describe a changing world. In particular, we want to be able to talk about relations that are true at some times and false at others. One way to do this is to allow those relations to have arguments that are time points or time intervals. For example, we might introduce a relation

```
(possesses ?person ?object ?time-point)
```

to mean the `?person` owns `?object` at `?time-point`, and

```
(possesses ?person ?object ?time-interval)
```

to mean the `?person` owns `?object` during `?time-interval`.

SNARK allows any relation to have arguments that are time intervals or time points, but that is not necessarily interpreted to mean that the relation is true at that time point or during that time interval. For instance, we might introduce a relation

```
(hour-long ?time-interval)
```

to mean that `?time-interval` is an hour long, not to mean that some relation `hour-long` is true during `?time-interval`.

Furthermore, SNARK does not assume that, if a relation holds for a time interval, it necessarily holds for any of the time points that occur during that interval, or for its end-points. Nor does it mean that the relation holds for any subinterval of that interval. For instance, suppose the relation

```
(profitable ?company ?time-interval)
```

holds if `?company` makes a profit over the period `?time-interval`. But that does not necessarily imply that `?company` makes a profit over every subinterval of `?time-interval`. For instance, a company can be profitable over a year but have lost money for the first quarter of that year.

Let us say that a relation is *inherited by subintervals* if, whenever it holds for a time interval, it holds for every subinterval of that interval. Thus, the relation `possesses` is inherited by subintervals but the relation `profitable` is not. SNARK will not assume that a relation is inherited by subintervals unless we introduce an assertion that says so. To say that `possesses` is inherited by subintervals, we may introduce the assertion

```
(assert
 '(implies
   (temporally-subsumes ?time-interval1 ?time-interval2)
   (implies
    (possesses ?person ?object ?time-interval1)
    (possesses ?person ?object ?time-interval2)))
 :name
  'possession-inherited-by-subintervals)
```

Now suppose we would like to say that George possessed the Maltese Falcon during a certain period of time, say "the good old days." Then we can give the assertion

```
(possesses george the-maltese-falcon good-old-days).
```

Because `possesses` is inherited by subintervals, this will imply that George possessed the Maltese Falcon during every subinterval of the good old days.

Now suppose we want to say that the good old days is the only time in which George possessed the Falcon. To say this in terms of the Allen logic, mentioning time intervals but not time points, we may say that George did

not possess the Falcon in any time interval other than the subintervals of the good old days.

```
(assert
 '(implies
   (not (temporally-subsumes good-old-days ?time-interval))
   (not (possesses george the-maltese-falcon ?time-interval)))
 :name 'george-does-not-possess-maltese-falcon-other-times)
```

Then if a period of hard times occurred later than the good old days, that is, if we have

```
(assert '(before good-old-days hard-times)
        :name 'good-old-days-before-hard-times)
```

then SNARK will be able to establish

```
(prove
  '(not (possesses george the-maltese-falcon hard-times)))
```

6.4 Mixed Point-Interval Relations

We have talked about temporal relations between pairs of time points and temporal relations between pairs of time intervals. In SNARK there are also *mixed* temporal relations, between time points and time intervals and between time intervals and time points. This is a useful extension of the Allen temporal logic.

The mixed relations have the same names as some of the point-point and interval-interval relations. Their meanings are analogous.

6.4.1 Point-Interval Relations

Some of these mixed relations hold only between a time point and a time interval, where the time point must be the first argument.

- `(starts ?time-point ?time-interval)`:

  Holds if `?time-point` is simultaneous with the start-point of `?time-interval`.

- `(finishes ?time-point ?time-interval)`:

  Holds if `?time-point` is simultaneous with the end-point of `?time-interval`.

- `(during ?time-point ?time-interval)`:

  Holds if `?time-point` is strictly between the start- and end-point of `?time-interval`.

6.4.2 Interval-Point Relations

The inverses of the point-interval relations hold only between an interval and a point, where the point must be the second argument.

- `(started-by ?time-interval ?time-point)`:

  Holds if `?time-point` is simultaneous with the start-point of `?time-interval`.

- `(finished-by ?time-interval ?time-point)`:

  Holds if `?time-point` is simultaneous with the end-point of `?time-interval`.

- `(contains ?time-interval ?time-point)`:

  Holds if `?time-point` is strictly between the start- and end-points of `?time-interval`.

Of the mixed relations, `before` and `after` allow points and intervals in either argument.

- `(before ?time-point ?time-interval)`:

  Holds if `?time-point` is strictly earlier than the start-point of `?time-interval`.

- `(before ?time-interval ?time-point)`:

  Holds if the end-point of `?time-interval` is strictly earlier than `?time-point`.

- `(after ?time-point ?time-interval)`:

  Holds if `?time-point` is strictly later than the end-point of `?time-interval`.

- `(after ?time-interval ?time-point)`:

  Holds if the start-point of `?time-interval` is strictly later than `?time-point`.

Some of the temporal primitives have no corresponding mixed relations. The equality relations `cotemporal` and `simultaneous-with` are not defined on mixed arguments, because a time point can never be temporally equal to a time interval. The relations `meets`, `met-by`, `overlaps`, and `overlapped-by` are not necessary for mixed arguments, because the concepts they would stand for are either nonsensical or already have other names. For instance, to say that a time point `meets` a time interval would be equivalent to saying that it `starts` the time interval. And it doesn't make sense for a time point to overlap a time interval.

6.4.3 Nonprimitive mixed relations

Some of the nonprimitive temporal relations can be used between an interval and a point, or between a point and an interval, including

- `(temporally-intersects ?time-interval ?time-point)`:
- `(temporally-intersects ?time-point ?time-interval)`:

  These relations are inverses. Both hold if `?time-point` occurs, not necessarily strictly, in `?time-interval`. That is, `?time-point` is simultaneous with the start- or end-point of `?time-interval` or is between those two time-points.

To summarize, a single Allen-style temporal reasoning system is used for all relations among points and intervals.

- There are three exhaustive, mutually exclusive relations between time points (`simultaneous-with`, `before`, and `after`).
- There are thirteen exhaustive, mutually exclusive relations between time intervals, given in Section 6.2.1.
- There are five exhaustive, mutually exclusive relations between a time point and a time interval (`before`, `starts`, `during`, `finishes`, and `after`), and their five inverses, which are relations between a time interval and a time point (`after`, `started-by`, `contains`, `finished-by`, and `before`).

Thus, all in all there are twenty-six primitive relations between temporal entities.

6.5 Temporal Functions

The temporal reasoning package defines functions that are useful for mapping time intervals and time-points into others. They include

- `(start-fn ?time-interval)`:

  The start-point of `?time-interval`.

- `(end-fn ?time-interval)`:

  The end-point of `?time-interval`.

- `(time-interval ?time-point1 ?time-point2)`:

  The time interval from `?time-point1` to `?time-point2`. That is, the start-point of the interval is `?time-point1` and the end-point is `?time-point2`. Here `?time-point1` must be before `?time-point2`.

- `(time-interval-ip ?time-interval ?time-point)`:

  The time interval from the start-point of `?time-interval` to `?time-point`. It is assumed that the start-point is before `?time-point`.

- `(time-interval-pi ?time-point ?time-interval)`:

  The time interval from `?time-point` to the end-point of `?time-interval`. It is assumed that `?time-point` is before the end-point.

- `(time-interval-ii ?time-interval1 ?time-interval2)`:

  The time interval from the start-point of `?time-interval1` to the end-point of `?time-interval2`. It is assumed that the start-point comes before the end-point.

6.6 Point-Interval Temporal and Relational Reasoning

The reason to introduce the mixed point-interval temporal relations is that reasoning about a changing world is more natural if we can talk about both points and intervals. For instance, it made sense (in Section 6.3) when we asserted that George possessed the Maltese Falcon during the good old days,

but it was a bit artificial when we said that George did not possess the Falcon during any time interval not temporally subsumed by the good old days.

It might have been more natural to use mixed time intervals and time points to express the same properties. For instance, let us say that a relation is *inherited by sub-points* if, whenever that relation holds for a time interval, it also holds for every time point that temporally intersects that interval. Note that a relation may be inherited by subintervals, as we discussed in 6.3, but not inherited by sub-points, and vice versa.

We can say that the relation `possesses` is inherited by sub-points by introducing the assertion

```
(assert
 '(implies
   (temporally-intersects ?time-interval ?time-point)
   (implies
    (possesses ?person ?object ?time-interval)
    (possesses ?person ?object ?time-point)))
 :name
 'possession-inherited-by-sub-points)
```

And we can say further that George does not possess the Falcon at any time point that does not occur during the good old days:

```
(assert
 '(implies
   (not (temporally-intersects good-old-days ?time-point))
   (not (possesses george the-maltese-falcon ?time-point)))
 :name 'george-doesnt-possess-maltese-falcon-other-times
```

And finally we can say that the special time point `now` is later than the good old days:

```
(assert '(before good-old-days now)
        :name 'good-old-days-before-now)
```

Note that all these assertions use temporal relations between time intervals and time points. From these assertions SNARK can establish

```
(prove '(not (possesses george the-maltese-falcon now))).
```

6.7 Calendar Dates and Clock Times

SNARK has a built-in representation of dates on the calendar and times on the clock, and this representation is integrated into the temporal inference procedure. For example, SNARK knows that December 31, 1999 meets January 1, 2000, in Allen's sense of "meets", and that 11PM on the former date is before 1AM on the latter date.

SNARK supports the following date functions, which are based on the Cyc ontology [Cyc-UL].

- `(year-fn ?integer)`:

  The time interval corresponding to the year `?integer`. For example, `(year-fn 1999)` is the time interval with start-point 00:00:00 hours on January 1, 1999, and end-point 00:00:00 hours on January 1, 2000.

- `(month-fn ?integer ?year)`:

  The time interval corresponding to the month numbered `?integer`, between 1 and 12; e. g., `(month-fn 5 (year-fn 1999))`. The month-fn function also accepts the names of the months, e. g. `(month-fn May (year-fn 1999))`.

- `(day-fn ?integer ?month)`:

  The time interval corresponding to the calendar day `?integer` of `?month`. For example, `(day-fn 10 (month-fn 5 (year-fn 1999)))` represents the time interval corresponding to May 10, 1999. Here `integer` must be at least 1; no promises are made about what happens if `?integer` is larger than the number of days in the month.

- `(hour-fn ?integer ?day)`:

  The time interval corresponding to the hour `?integer` of `?day`. For example, `(hour-fn 11 (day-fn 10 (month-fn 5 (year-fn 1999))))` represents the interval defined by 11:00 AM, May 10, 1999. SNARK uses a 24-hour day; `?integer` should be between 0 and 23. SNARK does not know about time zones; all times should be in the same time zone, but it does not matter which.

- `(minute-fn ?integer ?hour)`:

  The time interval corresponding to the minute `?integer` of `?hour`, where `?integer` is between 0 and 59. For example, `(minute-fn 12`

`(hour-fn 11 (day-fn 10 (month-fn 5 (year-fn 1999))))` represents the interval defined by 11:12 AM, May 10, 1999. The interval starts at 12 minutes after the hour, and continues until 13 minutes after the hour.

- `(second-fn ?integer ?minute)`:

  The time interval corresponding to the second numbered `?integer` of `?minute`, where `?integer` is between 0 and 59. For example, `(second-fn 13 (minute-fn 12 (hour-fn 11 (day-fn 10 (month-fn 5 (year-fn 1999))))))` represents the interval starting 11:12:13 A.M., May 10, 1999 and continuing until 11:12:14 on the same day.

6.8 Dates in Other Time Intervals

It is quite common to specify dates in a non-calendar time interval; for example, "The President made a statement on the third day of the war." SNARK supports such reasoning by an extension of the function `day-fn`, described as follows:

- `(day-fn ?integer ?constant)`:

  The fifth day of the time interval corresponding to `?constant`. For example, `(day-fn 5 scenario)` represents the fifth day in `scenario`. It is understood that `scenario` is supposed to correspond to a time interval, but SNARK does not enforce that. The date reasoning procedure is unable to compare dates between two different non-calendar time intervals, even if temporal relations are known between the two intervals. For instance, even if we assert `(before the-civil-war the-age-of-aquarius)`, SNARK will not know that `(before (day-fn 1 the-civil-war) (day-fn 2 the-age-of-aquarius))`.

6.9 Temporal Reasoner Interface

The interface between SNARK and the Allen temporal-reasoning procedure uses constraint resolution [Burckert]. Each row is split between ;

If a pure temporal relation between two ground (i. e., variable-less) terms is asserted or deduced, it is introduced into a graph representation of all known temporal relations. If the relation already follows from the relations in the graph, the graph is not changed. If the relation contradicts what is

already known in the graph, a contradiction has been deduced and the proof is complete.

If a more complex formula is asserted of deduced, it is split between a logical part and a temporal constraint, which are kept in the same row. The constrained row means that if some instance of the temporal constraint is satisfied, the corresponding instance of the logical part is true.

For example, suppose a proof contains the row

```
(Row 137
   (possesses george the-maltese-falcon ?time-interval)
   Temporal-Constraint
    (ii%temporally-subsumes good-old-days ?time-interval)).
```

Here `ii%temporally-subsumes` is the graph representation of the relation `temporally-subsumes`. This row means that if the good-old-days interval temporally subsumes `?time-interval`, then George possesses the Maltese Falcon during `?time-interval`.

Deduction rules applied to constrained rows generate a new row, which contains both a logical part and a temporal constraint. When the temporal constraint is ground, the graph representation of all known temporal constraints is used to see if the constraint is satisfied; if so, the constraint is removed. If the constraint contradicts the known temporal relationships, the entire row is discarded.

The link between the Allen relations that appear in formulas and their corresponding graph representations is achieved by a number of assertions that are added automatically by SNARK. For example:

```
(Row ~ii%temporally-subsumes
   (not (temporally-subsumes ?time-interval ?time-interval1))
   assertion
   Temporal-Constraint
    (not
     (ii%temporally-subsumes ?time-interval ?time-interval1))).
```

This asserts that if the graph representation of the relation `temporally-subsumes` is satisfied, the relation itself holds. Applying the resolution rule to this assertion and a formula that mentions the relation will have the effect of removing the relation from the formula and adding it to the temporal constraint.

The proof is complete when SNARK has discovered a contradiction in the logical parts of the rows, and when the corresponding temporal constraints are also satisfied. Should SNARK deduce a contradictory row `false` whose corresponding temporal constraints cannot be satisfied, the row is discarded and the search continues.

7 Procedural Attachment

We have seen that, for some areas such as temporal reasoning, it is advantageous to use special-purpose inference procedures rather than to rely on only SNARK's general-purpose inference rules. It is impossible, however, for SNARK to include every special-purpose procedure that may be useful for some application. Instead, SNARK includes a procedural attachment facility that allows a user to invoke external procedures as part of the standard resolution.

There are two principal ways to introduce procedural attachments, by intervening in either the rewriting mechanism or the resolution mechanism. We treat each separately. (There is also a way to use procedural attachment via the paramodulation rule.)

7.1 Rewrite Code

The rewrite-code mechanism allows the user to provide external code to rewrite expressions, much as a rewrite rule does. This code is associated with particular function or predicate symbols.

7.1.1 Built-in Rewrite Code

SNARK already has built-in procedural attachments, in the form of rewrite code, for important arithmetical, symbolic, and list-processing functions. For instance, if there were no procedural attachment mechanism, the only way we would be able to add two numbers would be to reason from the axioms for addition, a rather ponderous business. However, if we select the option `use-code-for-numbers`, arithmetic operations will be carried out by the corresponding LISP code. In particular, a term `(+ 2 2)` will be immediately rewritten as `4`, without invoking any axioms or inference rules. SNARK has procedural attachments for the principal arithmetic function and predicate

symbols in the ANSI KIF COMMON LISP library. Similarly, selecting the options `use-code-for-lists` and `use-code-for-characters` will give SNARK access to ANSI KIF COMMON LISP's list and character libraries.

The SNARK equality function is also rewritten by a built-in procedural attachment. For instance, a formula of form `(= <term> <term>)`, where both arguments are alike, will be automatically rewritten to `true`.

A procedure can be attached to a user-defined SNARK function or predicate symbol by means of its declaration. Some built-in procedures can be attached to user-defined symbols. For example, suppose we wish to declare that the relation `near` is reflexive, that is, that a place is to be regarded as near to itself. Then we may include the declaration

```
(declare-predicate-symbol
  'near 2 :rewrite-code 'reflexivity-rewriter)
```

in addition to whatever other declarations are given for the predicate symbol `near`. The program `reflexivity-rewriter` is LISP code built into SNARK that performs a rewriting analogous to the one we have described for equality. In using this rewriting in the course of a proof, SNARK will report `rewrite ... :code-for-near` in the explanation.

7.1.2 User-Supplied Rewrite Code

Let us look at the program `reflexivity-rewriter` described in the previous section; then we can describe the constructs necessary to understand it and to build analogous rewrite code for our own theories.

```
(defun REFLEXIVITY-REWRITER (atom subst)
  (let ((args (args atom)))
    (if (equal-p (first args) (second args) subst) true none)))
```

This program tests if, after applying the substitution `subst`, the two arguments of the formula `atom` are equal; if so, it returns `true`; otherwise, it returns the special symbol `:none`, which indicates that the formula is not to be rewritten. (The LISP variable `none` has value `:none`.)

Every piece of rewrite code has two arguments, an expression and a substitution, here called `atom` and `subst` respectively. The actual expression being simplified is the result of applying `subst` to `atom`. For example, if `atom` is `(near ?place ohio)` and `subst` is a substitution that replaces `?place` with

`ohio`, the actual expression being substituted is `(near ohio ohio)`. For reasons of efficiency, SNARK sometimes carries around the substitution rather than applying it.

The following LISP functions are defined in SNARK and are useful for writing procedural attachments.

`:none`:

As mentioned above, a special symbol that can be returned by a rewrite rule to indicate that the expression is not to be rewritten by that rule. The LISP variable `none` is assigned the value `:none`.

`(head exp)`:

The principal function or predicate symbol of the expression `exp`.

`(args exp)`:

The argument list of the expression `exp`.

`(equal-p exp1 exp2 subst)`:

A test which is true if applying the substitution `subst` to the expressions `exp1` and `exp2` yields identical expressions, false otherwise.

A more complex construct is

```
(dereference exp subst
  :if-constant const-code
  :if-variable var-code
  :if-compound comp-code)
```

The construction applies the substitution `subst` to the expression `exp` and evaluates `exp`. Then

If the result is a constant, it evaluates the LISP expression `const-code`.

If the result is a variable, it evaluates the LISP expression `var-code`.

If the result is a compound expression, such as the application of a function symbol to arguments, it evaluates the LISP expression `comp-code`.

In each case, the value of the entire `dereference` expression is the value of the evaluated subexpression. Any of the keyword cases can be omitted, and their order is inconsequential. In case none of the given keywords is applicable, the value of the `dereference` expression is `nil`.

7.1.3 Example: Rewrite Code for `mother`

Let us use some of these constructs in the family theory to introduce rewrite code for the function `mother`. Suppose there is a LISP function `mother-fun` that can compute the mother of any constant of sort `person`; for instance, `mother-fun` might consult external geneological tables for this purpose. We would like to attach this program to the function symbol `mother` in our family theory.

First we indicate that the symbol `mother` is to be given a procedural attachment in the form of rewrite code.

```
(declare-function-symbol 'mother 1
  :rewrite-code 'mother-rewriter)
```

Then we define the function `mother-rewriter` to extract the argument `arg` from terms of form `(mother arg)`, where `arg` is a constant, and to invoke `mother-fun` on that constant:

```
(defun MOTHER-REWRITER (term subst)
  (let ((child (first (args term))))
    (dereference child subst
                 :if-constant
                 (mother-fun child)
                 :if-variable none
                 :if-compound none)))
```

Note that this will have no effect on terms of form `(mother arg)`, where `arg` is a variable or a compound term. Also, we are assuming that all the constant symbols returned by `mother-fun` have already been declared to be of sort `woman` (and hence `person`); if new names are introduced by `mother-fun`, they must be declared by `mother-rewriter`.

Given this procedural attachment, SNARK can answer queries such as

```
(prove
 '(and (= ?woman (mother ?person))(parent ?person carol))
 :answer '(ans ?woman)
```

where all the information it has about the mothers of individual constants is given by the function `mother-fun`. SNARK will behave as if all this information was stored as rewrite rules. For instance, suppose

```
(mother-fun carol) = betty
```

and

```
(mother-fun betty) = alice.
```

Then SNARK will include `alice` among its answers for the above query. Use of `mother-fun` in the proof will be justified by the annotation `rewrite ... :code-for-mother` in the explanation.

7.2 Satisfy and Falsify Code

The same limitations that apply when information is represented by rewrite rules also applies when information is stored in rewrite code. For instance, we cannot use the rewrite code for mother to answer the query

```
(prove
 '(= betty (mother ?person))
 :answer '(ans ?person)
 :name 'whose-mother-is-betty?).
```

Although the procedural attachment can rewrite `(mother carol)` to `betty`, it has no effect on the term `(mother ?person)`. In fact, because of the one-way nature of a function, we cannot use a procedural attachment to a function symbol to answer this kind of question. We can, however, use the *satisfy-* and *falsify-code* mechanisms, which allow a procedural attachment to intervene in the resolution mechanism. We introduce these with an example.

7.2.1 Satisfy Code for mother

Suppose that, in addition to the `mother` function symbol, we also introduce a two-place `mother` predicate symbol. Imagine that we have a table of mother-child pairs, e.g.,

```
(defvar MOTHER-TABLE
  '((alice betty)
    (alice barbara)
    (betty carol)
    (betty claudia)
    ))
```

In other words, Alice is the mother of Betty, Alice is the mother of Barbara, and so forth. Then we can use the satisfy-code mechanism to make SNARK behave as if it had been given the corresponding atomic assertions

```
(assert '(mother alice betty))

(assert '(mother alice barbara))

(assert '(mother betty carol))

(assert '(mother betty claudia))
```

We do this by attaching to the predicate symbol `mother` a procedure that, whenever SNARK is trying to establish the truth of a formula of form `(mother person1 person2)`, will cycle through `mother-table` and attempt to unify the pair `(person1 person2)` with successive pairs of the table.

First we indicate that the predicate symbol `mother` is given a procedural attachment in the form of satisfy code:

```
(declare-predicate-symbol 'mother 2
  :satisfy-code 'mother-satisfier)
```

Then we provide a LISP function `mother-satisfier` that will attempt to unify the arguments of the formula against the successive pairs of the table:

```
(defun MOTHER-SATISFIER (cc atom subst)
  (let ((args (args atom)))
    (mapc
     (lambda (pair) (unify cc args pair subst))
     mother-table)))
```

Here each pair in `mother-table` is unified with the pair of arguments of the formula under consideration. Note that `mother-satisfier`, like all satisfy code, has a continuation `cc` as its first argument. The continuation is a function that, when called, will attempt to complete the rest of the proof. If `unify` succeeds in unifying `args` with `pair`, it will then invoke the continuation `cc`, passing on whatever substitutions `unify` has discovered.

Once we have provided this procedural attachment, SNARK can use the table to provide answers to the query

```
(prove
 '(mother betty ?person)
 :answer '(ans ?person)
 :name 'who-is-bettys-child?).
```

Invoking this query once will provide one answer, Carol. Then executing `(closure)` will send SNARK back into the table to find another answer, Claudia.

While introducing a procedural attachment for such a small table gives no benefit, it is reasonable to introduce procedural attachments for large tables and it is unavoidable if the table can only be accessed via an external function call or a web access.

7.2.2 Falsify code for `mother`

Satisfy code allows us to establish that a relation is true as the result of executing a procedure. Sometimes, however, a procedure can tell us that the relation is false. To invoke such a procedure, we use *falsify code*.

For instance, suppose we want to introduce code to embody the idea that the relation `mother` is irreflexive, i. e., that a person cannot be her own mother. One could do this by introducing an assertion

```
(assert
 '(not (mother ?person ?person))
 :name 'mother-is-irreflexive)
```

Alternatively, we can introduce falsify code to have the same effect as resolution against the above assertion.

For this purpose, we declare the predicate symbol `mother` to have falsify code `mother-falsifier`:

```
(declare-predicate-symbol 'mother 2
  :falsify-code 'mother-falsifier)
```

Note that the same symbol can have satisfy code, falsify code, and rewriting code.

We define the LISP function `falsify-code` as

```
(defun MOTHER-FALSIFIER (cc atom subst)
  (let ((args (args atom)))
    (unify cc (first args) (second args) subst)))
```

Here, again, `cc` is continuation code that attempts to complete the proof. This function will be invoked when a formula `(mother person1 person2)` occurs in a context in which we are trying to prove that it is false, e. g., in a query of form

```
(prove
 '(not (mother person1 person2))),
```

where `person1` and `person2` are terms that may have variables. It will attempt to unify `person1` and `person2`. If it succeeds, `unify` will call the continuation code `cc`, passing on whatever substitutions it has discovered, in an attempt to complete the proof. If `unify` fails, `mother-falsify` returns and other avenues to complete the proof are sought.

Actually, SNARK has general-purpose code for declaring a relation to be irreflexive and introducing the appropriate falsify code. So, to declare the predicate `mother` to be irreflexive, we could obtain the same effect simply by introducing the declaration

```
(declare-predicate-symbol 'mother 2
  :falsify-code 'irreflexivity-falsifier).
```

8 Support for KIF/OKBC Users

Knowledge Interchange Format (KIF) is a language designed for use in the interchange of knowledge amongst disparate computer systems [Genesereth]. KIF was a result of a community effort and a draft of the KIF specification is under consideration as an ANSI standard. Open Knowledge Base Connectivity (OKBC) is an application programming interface for accessing knowledge representation systems. OKBC interfaces to many popular knowledge representation systems exist.

Given such a broad base of KIF and OKBC users, SNARK supports input of axioms in a language called KIF+C("KIF plus Classes"). KIF+C uses the ANSI draft KIF syntax for writing axioms, and recognizes some standard relation names from the OKBC knowledge model.

The KIF+C not only makes it easier for SNARK to do knowledge sharing with other systems, but also implements a connection between three ways of representing classes of entities in SNARK: through sorts, through sets, and through predicate symbols. In this section we introduce this way of presenting information to SNARK.

8.1 Introduction to KIF

KIF has declarative semantics, is logically comprehensive with its support for arbitrary logical sentences, and supports representation of knowledge about knowledge.

KIF accepts sentences built up of constants, function and predicate symbols, logical connectives, and quantifiers. KIF accepts the equality symbol `=`, the connectives `and`, `or`, `not`, and the quantifiers `forall` and `exists`.

Function and relation symbol may occur with different arities. KIF assumes that if the same symbol occurs with varying arity, all those occurrences stand for the same function, and the function itself has variable arity.

KIF has three directions of implication connective, `=>`, `<=` and `<=>`.

Free variables in assertions have tacit universal quantification, while free variables in queries have tacit existential quantification.

KIF has four constructs for defining new symbols:

defobject Introduces a new constant, standing for a thing or entity.

defrelation Introduces a new predicate symbol, standing for a relation.

deffunction Introduces a new function symbol, standing for a function.

deflogical Introduces a new propositional symbol, standing for a truth-value.

8.2 Description of KIF+C

The domain of discourse for the KIF+C consists of individuals, relations, functions, and assertions. Unary relations are identified with classes. We consider each of these, beginning with classes, which are given special treatment in KIF+C because they are tied to SNARK's sort mechanism.

To enable the KIF+C interface, we place the following statement at the top of our source files:

```
(in-language :hpkb-with-ansi-kif)
```

or

```
(in-language :hpkb-with-kif-3.0)
```

All the examples in this section will work with either version of the KIF interface.

8.2.1 Declaring Classes

A class in KIF+C corresponds to a set, and a unary predicate symbol (i. e., a predicate symbol of arity 1). It is declared with the construct `defrelation`.

For instance, to declare a class, `person`, we execute

```
(defrelation person
  (class person))
```

The relation name `class` is a standard name derived from the OKBC knowledge model.

The declaration constructs each allow an optional string as their first argument, which can be used for documentation:

```
(defrelation object
    "Collection of all objects."
  (class object))
```

Specifying documentation string as an optional first argument is allowed in the ANSI version of KIF, but not in KIF 3.0.

Within the `defrelation` construct we can provide many declarations and axioms related to the class being declared. For instance, we can declare that one class is a subclass of another:

```
(defrelation man
    "Collection of all men."
  (class man)
  (subclass-of man person))
```

This construct introduces a new class `man` and declares that `man` is a subclass of `person`, i. e., that every man is also a person. The relation name `subclass-of` is a standard relation name based on the OKBC knowledge model.

It is a convention (which SNARK does not enforce) that the statements included in a declaration be relevant to the entity being declared; in particular, it is recommended that a subclass declaration `(subclass-of man ...)` should appear in the declaration for the class `man`, rather than elsewhere.

8.2.2 Declaring Individuals

If we want to introduce an individual that is not itself a set, we use the construct `defobject`. For instance,

```
(defobject george
  (instance-of george man))
```

introduces a constant `george` that is of sort `man` and an element of the set `man`. Here `instance-of` corresponds to the set membership relation, and is a standard relation name derived from the OKBC knowledge model.

The predicate symbols `subclass-of` and `instance-of` are given special treatment in SNARK. In particular, when the class `person` is declared, an assertion

```
(instance-of ?person person)
```

is automatically introduced; in other words, any term of sort `person` stands for an element of the set `person`. Also, any formula of form

```
(person <term>)
```

where `<term>` is a term, is automatically rewritten as

```
(instance-of <term> person).
```

Thus, any use of `person` as a predicate symbol is automatically translated into a use of `person` as a set.

Special procedures are built into SNARK to take into account the class, subclass, and object declarations during a proof. For instance, with the above declarations, that `george` is a `man` and that `man` is a subclass of `person`, SNARK will be able to prove immediately that `george` is a person, i. e., that

```
(person george).
```

This is rewritten as

```
(instance-of george person).
```

The proof is carried out simply by examining the declared sorts and objects, without invoking any axioms. Because `man` is a subset of `person`, any member of `man` is also a member of `person`.

It is possible to use the relation `instance-of` to assert that a class, rather than an individual, is a member of another class. For instance, here is a declaration of the class `woman`.

```
(defrelation woman
    "Collection of all women."
  (class woman)
  (instance-of woman biological-classification-type))
```

The statement

```
(instance-of woman biological-classification-type)
```

says that the class of women is a biological classification type, a kind of class. This is quite different from saying that the class of women is a subclass of the biological classification type—that would imply that every woman is herself a class.

It is natural to make assertions about the properties of a class as follows.

```
(defrelation person
  (class person)
  (average-age person 70))
```

The `average-age` statement says something about the entire class of people, not about individual members of the class. If we really want to say something about each element of the class, we can use the construct `template-slot-value`, another standard relation name from the OKBC knowledge model:

```
(defrelation person
  (class person)
  (template-slot-value ancestor person adam))
```

Here

```
(template-slot-value ancestor person adam)
```

says that an ancestor of every person is Adam. This is equivalent to the following SNARK assertion:

```
(ancestor ?person adam).
```

8.2.3 Declaring Relations

We have seen that when we declare a class we are simultaneously declaring a unary predicate symbol, which stands for a unary relation. Let us consider the declaration of n-ary relations.

For example, suppose we want to declare a relation `possesses`, which takes two arguments, a `person` and an `object`. Then we may say

```
(defrelation possesses
    "a ?person possesses an ?object if he or she owns it."
  (relation-arity possesses 2)
  (nth-domain possesses 1 person)
  (nth-domain possesses 2 object))
```

An alternative, and equivalent, way to make the same declaration would be

```
(defrelation possesses
    "a ?person possesses an ?object if he or she owns it."
  (relation-arity possesses 2)
  (domain possesses person)
  (slot-value-type possesses  object))
```

This last formulation is acceptable only for binary relations, of arity 2. The relation names `domain`, and `slot-value-type` are derived from the OKBC knowledge model.

The assertions involving the relations can be included in a `defobject` construct.

```
(defobject the-maltese-falcon
    "the-maltese-falcon is an object possessed by George."
  (instance-of the-maltese-falcon object)
  (possesses george the-maltese-falcon))
```

There are many relations, for example, `average-age`, that apply to classes. The arguments of such relations are constant symbols that represent classes. While declaring such relations, the relevant arguments should be restricted to be classes. This can be accomplished as follows.

```
(defrelation average-age
    "Average age of the members of a collection of objects''
  (relation-arity average-age 2)
  (nth-domain-subclass-of average-age 1 physical-object)
  (nth-domain average-age 2 integer))
```

The relation name `nth-domain-subclass-of` restricts the first argument to only those constant symbols that represent classes that are subclasses of `physical-object`. The relation names `range-subclass-of`, `slot-value-type-subclass-of` and `domain-subclass-of` may also be used.

To support the implementation of the type restriction when the arguments of a relation are restricted to the constant symbols representing classes, for every KIF+C class, we automatically declare a class of subclasses of that class. For example, because we have declared a class `person`, we declare `subclass-of-person` as a class as well. Every `subclass-of person` will be an `instance-of subclass-of-person`. Internally, SNARK reduces the `nth-domain-subclass-of` restriction to an `nth-domain` restriction on class of subclasses. For example,

```
(nth-domain-subclass-of average-age 1 physical-object)
```

is internally represented as

```
(nth-domain average-age 1 subclass-of-physical-object)
```

The objective of declaring the class of all subclasses was to take advantage of the sort system in SNARK to deal with meta-classes. The sort names such as `subclass-of-person` are not meant to be visible to the user, and are outside the scope of KIF+C. It is, however, possible to use them while writing axioms. For example, SNARK will recognize `?subclass-of-person` as a variable of sort `subclass-of-person`. Quantification over classes is not a well explored area for KIF+C and is open for future research.

Any of the keyword arguments accepted by the SNARK relation declarations can be supplied through KIF+C. For example, the following SNARK declaration

```
(declare-function-symbol 'mother 1
  :rewrite-code 'mother-rewriter)
```

can be written in KIF+C as follows:

```
  (deffunction mother
(function-arity mother 1)
(rewrite-code mother mother-rewriter))
```

The SNARK keyword arguments are recognized as KIF+C relation names.

8.2.4 Declaring Functions

KIF function declarations are similar to the relation declarations. For example, suppose we want to declare the function `mother`, which takes a `person` as its argument and yields a `woman` as its value. Then we may use the KIF construct `deffunction`, which is analogous to `defrelation`:

```
(deffunction mother
    "the mother of a person."
  (function-arity mother 1)
  (nth-domain mother 1 person)
  (range mother woman)
  (parent (mother ?person) ?person))
```

8.2.5 Declaring Assertions

The assertion declaration is an extension to KIF, included to allow one to make SNARK assertions with KIF syntax, outside of declarations. Assertions may be given a name and a documentation string. For example, to say that everyone has at most one spouse (at a given time), we can make the assertion

```
(assertion
 (forall ((?x1 person)
          (?x2 person)
          (?x person))
         (=>
          (and (spouse ?x ?x1)
               (spouse ?x ?x2))
          (= ?x1 ?x2)))
 :name uniqueness-of-spouse
 :documentation
  "A person may have only one spouse at a time.")
```

Note that SNARK accepts KIF syntax for logical symbols, such as `=>` instead of `implies`. Also note that KIF does not have the convention that `?person1` is a variable of sort `person`, say. Therefore we have spelled out explicitly the sort of each variable:

```
(forall ((?x1 person)
         (?x2 person)
         (?x person)) ... )
```

SNARK allows automatic coercion of variable types. For example, if the above assertion were written as

```
(assertion
  (=>
   (and (spouse ?x ?x1)
        (spouse ?x ?x2))
   (= ?x1 ?x2))
 :name uniqueness-of-spouse
 :documentation
  "A person may have only one spouse at a time.")
```

and the arguments of `spouse` were declared to be of type `person`, SNARK could automatically coerce `?x`, `?x1`, and `?x2` to be of types `person`. To enable such automatic sort coercion, one select the option `use-well-sorting`.

The `uniqueness-of-spouse` axiom can also be written as

```
(assertion
  (=>
   (and (person ?x)
        (person ?x1)
(person ?x2)
        (spouse ?x ?x1)
        (spouse ?x ?x2))
   (equal ?x1 ?x2))
 :name uniqueness-of-spouse
 :documentation
  "A person may have only one spouse at a time.")
```

Exercise: Uniqueness of Mothers-in-law. Within the KIF extension of SNARK introduce a relation `mother-in-law`. Using your definition and the KIF family theory introduced in this section, use SNARK to prove that everyone has at most one mother-in-law.

Solution. The definition of a mother-in-law is given within the KIF declaration

```
(defrelation mother-in-law
    "?x1 is the mother-in-law of ?x2
     if ?x1 is the mother of the spouse of ?x2."
  (relation-arity mother-in-law 2)
  (domains mother-in-law person person)
  (forall ((?x1 person)
           (?x2 person)
           (?x person))
    (<=>
     (mother-in-law ?x1 ?x2)
     (exists ((?x person))
     (and
(spouse ?x1 ?x)
(equal ?x2 (mother ?x)))))))
```

To prove the uniqueness of the mother-in-law, we give SNARK the task

```
(prove '(forall ((?x1 person)
                 (?x2 person)
                 (?x person))
          (=>
           (and
            (mother-in-law ?x ?x1)
            (mother-in-law ?x ?x2))
           (equal ?x1 ?x2))))
```

8.3 Relationship of KIF+C with SNARK

It is helpful to consider the equivalence between the KIF+C constructs and the native SNARK constructs.

In both SNARK and KIF the same function and relation symbol may occur with different arities. However, in SNARK there is no assumption that the meaning of a symbol with arity 2, say, has any relation at all to the meaning of the same symbol with arity 3. In KIF, on the other hand, the assumption is that if the same symbol occurs with varying arity, all those occurrences stand for the same function, and the function itself has variable arity.

For example, in SNARK we could introduce a unary function `minus` and a binary function `minus`, without the idea that both of these are instances of the same function. In KIF, we would need to represent these functions by different symbols. If we talk about `plus` with two arguments in one place and three arguments in another in KIF, it is understood that these are the same function.

KIF has two directions of implication connective, `=>` and `<=`, analogous to the SNARK `implies` and `implied-by`. However, while the SNARK connectives accept exactly two arguments, the KIF connectives accept two or more. The KIF

```
(=> <Form1> ... <Formn> <Form>)
```

is equivalent to the SNARK

```
(implies (and <Form1> ... <Formn>) <Form>)
```

and the KIF

```
(<= <Form> <Form1> ... <Formn>)
```

is equivalent to the SNARK

```
(implied-by <Form> (and <Form1> ... <Formn>))
```

The declaration

```
(defrelation person
  (class person))
```

makes the following declarations in SNARK

- the sort `person`.
- a constant `person`, which stands for the set of all people.

- a unary predicate symbol `person`, which stands for the relation that is true for people and false for other entities.

Thus the KIF `class` declaration has more effects than the simple SNARK declaration

```
(declare-sort 'person).
```

The following declaration

```
(defrelation possesses
    "a ?person possesses an ?object if he or she owns it."
  (relation-arity possesses 2)
  (nth-domain possesses 1 person)
  (nth-domain possesses 2 object)
```

is equivalent to the SNARK declaration

```
(declare-predicate-symbol 'possesses 2
  :sort '(boolean person object))
```

8.4 Built-in Number Sorts

SNARK has some built-in sorts of numbers. The number sorts in SNARK are based on the ANSI KIF specification. SNARK recognizes the following number sorts: `number`, `complex`, `real`, `rational`, `integer`, `natural`, `zero`, `positive`, `negative`, `odd`, and `even`. The sort `number` is declared to be a subsort of `complex`, `real` a subsort of `complex`, `rational` a subsort of `real`, and `integer` a subsort of `rational`. The sort `real` is partitioned into three disjoint subsorts: `negative`, `zero`, and `positive`. The sort `integer`is partitioned into `even` and `odd`, with `zero` included as a subsort of `even`. The sort `natural` comprises the nonnegative integers. To enable the automatic declaration of these number sorts, one must select the option `use-number-sorts`.

SNARK's number sorts are based on those of KIF, which are based in turn on COMMON LISP's number types.

Acknowledgment of Support

The research reported here has been partly supported by DARPA under Contracts N66001-97-C-8550 (HPKB) and N66001-97-8551-00-SC-01, Subcontract PSRW-97-8551-00-SC-01 (Genoa).

References

[Allen] J. F. Allen, Time and Time Again: The Many Ways to Represent Time, *International Journal of Intelligent Systems*, Vol. 6, No. 4 (July 1991), pp. 341–355.

[Burckert] H.-J. Bürckert, *A Resolution Principle for a Logic with Restricted Quantifiers*, Lecture Notes in Artificial Intelligence No. 568, Springer Verlag, Berlin (1991).

[Chang] C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, NY (1973).

[Chaudhri] V. K. Chaudhri, A. Farquhar, et al., OKBC: A Programmatic Foundation for Knowledge Base Interoperability, *Proceedings of the AAAI-98*, Madison, WI (1998).

[Dershowitz] N. Dershowitz and J.-P. Jouannaud, "Rewrite Systems," in J. van Leeuwen (editor), *Handbook of Theoretic Computer Science*, Elsevier, Amsterdam, The Netherlands (1989), pp. 241–320.

[Genesereth] M. R. Genesereth and R. E. Fikes, *Knowledge Interchange Format, Version 3.0 Reference Manual*, (Logic-92-1) (1992).

[Graham] P. Graham, ANSI *Common Lisp*, Prentice Hall, Englewood Cliffs, NJ (1996).

[Lenat] D. Lenat and R. V. Guha, *Building Large Knowledge Based Systems*, Addison-Wesley, Reading, MA (1990). See also `http://www.cyc.com`.

`[Cyc-UL]` `D. Lenat, *Cyc Upper Ontology*, See http://www.cyc.com/cyc-2-1/index.html`

`[McCune]` `W. McCune, *Otter 3.0 User's Guide*, Technical Report ANL-94/6, Argonne National Laboratory, Argonne, IL (1994).`

[Manna] Z. Manna and R. Waldinger, *Deductive Foundations of Computer Programming*, Addison-Wesley, Reading, MA (1993).

[Waldinger] Z. Manna and R. Waldinger, ‘‘Fundamentals of Deductive Program Synthesis,’’ *IEEE Transactions on Software Engineering,* Vol. 18, No. 8 (August 1992), pp. 674--704.

[Pitman] K. Pitman, *Common Lisp HyperSpec*, Harlequin Group, Cambridge, UK (1996).