A domino solitaire problem as a case study

Ph. D. course on Verification of Logic Programs

S. Contiero, A. Giani
{contiero,giani}@di.unipi.it

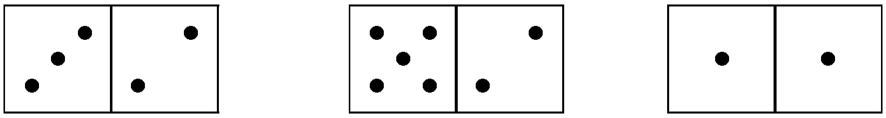
July 16, 1997

1 Description of the problem

Aim of this work is to investigate the effective applicability of methods and results introduced in [1, 2, 3, 4] in order to check several useful properties of the solution to the following problem.

1.1 Domino solitaire

You are given a set of domino stones. Each stone has 2 numbers (on the same side), one at the left and one at the right, like the examples below show :



In the usual game, each participant is given a collection of stones, some stones are put on the table, and each player in turn must either add a stone to the ones already on the table, or pass. The winner is the one who got rid of his/her stones first. One legally adds a stone, if there is a stone on the table with a free end containing number x, and he/she puts one of his/her stones with its x-side next to the free x-side on the table. Stones with two different numbers have 2 free sides initially- one for each number. Stones with two equal numbers on them, have 3 free sides. When a stone is put with a free side of another stone, both free sides become occupied. We consider here a variant, called *domino solitaire*. One has given a set of stones and he/she is to finish the game by himself/herself, i.e. choose a first stone and continue adding stones, until all of them are placed on the table. This can fail of course: In such case there is no sequence of moves able to finish all the stones.

2 A possible solution

2.1 Formal specification

Let $Stones(\alpha)$ be the set of domino stones whose symbols belong to α. The set α may be either the set $\{1,2,3,4,5,6\}$ of numeral ground arithmetic expressions, as in the usual game, or any set of ground terms. A stone $s \in Stones(\alpha)$ is represented as `stone(`x`,`y`)`, where $x, y \in \alpha$. When a stone is added to the other ones on the table, we have to specify which side is used, i.e. which side becomes occupied and which one becomes free. To represent how the stones are used, we define

- the set of oriented stones, $OStones(\alpha) = \{\langle x, y\rangle \mid x, y \in \alpha\}$,
- the function $h : Stones(\alpha) \times \{1,2\} \longrightarrow OStones(\alpha)$ that associates with a stone and with an index (i.e., the used side) the corresponding oriented stone:

$$h(stone(x,y),b) = \begin{cases} \langle x, y\rangle & \text{if } b = 1 \\ \langle y, x\rangle & \text{if } b = 2 \end{cases}$$

To represent the current free sides we use multisets of elements of α, denoted by $M(\alpha)$.

We employ the notation *ms(L)* to denote the multiset whose elements belong to the list L (regardless of the list ordering), and with $\backslash$ and $\cup$ respectively the operations of difference and union among multisets. It is worth observing that multisets can also be denoted by sets of pairs (a, i) where the first component denotes the element and the second component of the pair denotes its molteplicity (indeed, in the sequel we will always choose the list-with-repetitions notation, since it is more convenient in our context). A zero-molteplicity indicates that the element does not belong to the multiset. This compact representation helps to give a simple definition of the $\backslash$ and $\cup$ operations, as follows:

$A \backslash B = \{(a,i) \mid\ (a,j) \in A \wedge\ j > 0\ \wedge\ (a,k) \in B \wedge\ k \geq 0\ \wedge\ i = j - k\ \}.$
$A \cup B = \{(a,i) \mid\ (a,j) \in A \wedge\ (a,k) \in B \wedge\ i = j + k\ \}.$

We also introduce an operator $\oplus : M(\alpha) \times OStones(\alpha) \longrightarrow M(\alpha)$ in order to characterize how the set of current free sides changes when a stone is placed on the table. The operator is defined by means of oriented stones because the new set of free sides depends on the choice of the side (to be occupied). For $m \in M(\alpha)$ and $\langle x, y\rangle \in OStones(\alpha)$, we then have:

$$m \oplus \langle x, y\rangle = \begin{cases} \{x, y\} & \text{if } m = \emptyset\ \wedge\ x \neq y \\ \{x, x, x\} & \text{if } m = \emptyset\ \wedge\ x = y \\ m \backslash \{x\} \cup \{y\} & \text{if } m \neq \emptyset\ \wedge\ x \in m\ \wedge\ x \neq y \\ m \cup \{x\} & \text{if } m \neq \emptyset\ \wedge\ x \in m\ \wedge\ x = y \end{cases}$$

It is worth noting that $m \oplus \langle x, y\rangle$ is not defined if $m \neq \emptyset \wedge x \notin m$, (i.e. the operator specifies how the set of free sides changes only when a stone is *legally* placed on the table). In the following we will use the shorthand

$$m \oplus s_1 \oplus \cdots \oplus s_n$$

for

$$(\cdots((m \oplus s_1) \oplus s_2) \cdots) \oplus s_n.$$

Starting from the previous definitions, we can formally define a solution for the domino solitaire problem. Given a list of stones L which represents the set of stones initially given to the player, a domino solution, denoted by $dominoSol(L)$, is a list L' of stones such that:

1. $L' = [s_1, \ldots, s_n]$ is a permutation of L;

2. $\exists b_1, \ldots, b_n \in \{1, 2\}$ s.t. for $i \in [2, n]$

$$x_i \in \emptyset \oplus \langle x_1, y_1 \rangle \oplus \cdots \oplus \langle x_{i-1}, y_{i-1} \rangle$$

where $\langle x_i, y_i \rangle = h(s_i, b_i)$ (for $i \in [1, n]$).

As a remark, the actual value of b_1 does not affect the solution, since the sides of the first stone are initially all free. The definition above can be extended to consider an arbitrary (possibly non empty) set of free sides to start from. Given a list L of stones and a (possibly non empty) multiset of free sides $m \in M(\alpha)$ a domino solution $dominoSol(L, m)$, is a list L' of stones such that:

1. $L' = [s_1, \ldots, s_n]$ is a permutation of L;

2. $\exists b_1, \ldots, b_n \in \{1, 2\}$ s.t. for $i \in [1, n]$

$$x_i \in m \oplus \langle x_1, y_1 \rangle \oplus \cdots \oplus \langle x_{i-1}, y_{i-1} \rangle \tag{1}$$

where $\langle x_i, y_i \rangle = h(s_i, b_i)$ for $i \in [1, n]$

It is worth observing that this second definition of *dominoSol(L,m)* properly extends the previous one, since the latter can be seen as a particular case (i.e., *dominoSol(L,$\emptyset$)*) of the former.

2.2 A Prolog solution

We now present a possible solution, in term of a Prolog program, for the problem described above. Then we will analyse whether the various properties studied in [1, 2, 3, 4] hold or not for our case.
As a starting point, we briefly discuss some initial assumptions adopted. First of all, we assume that the input is represented by a list of *ground* terms. This has notable consequences on various aspects, as we will point out in the sequel. Furthermore, since *every* possible solution does not depend on the first stone chosen (i.e., every solution can be permuted so to have as a first element, a desired one), we decide to accept as a first element of a possible solution, the first element in the given input list. We finally highlight that a crucial issue, in order to solve the problem, is to keep track of how many sides (and which ones) are currently free. Before presenting the code, we clarify the notation that will be used throughout.

- We employ the shorthand $ListS$, $ListOS$ and $\alpha List$ to denote the (set of) lists of stones, the (set of) lists of oriented stones, and the (set of) lists of α-elements respectively. We outline that, since all these sets are made of *ground* elements, the elements of such sets are automatically ground (i.e., in other words, they all belong to the set $ListGt$ of lists of ground terms). In addition, we recall that each subterm of a ground term is ground, too.
- $ms(L)$ is the multiset of the elements belonging to the list L.
- ||L|| denotes the length of the list L.

Below is reported the (Prolog) code of the program `SOLITAIRE`. Each predicate is augmented with a `mode`, a couple of *(Pre,Post)-interpretations*, a `DELAY` declaration, and a level mapping.

```
% domino(L1,L2)  ←  L2 (which is a permutation of list L1) identifies
%                   the sequence in which the stones, belonging to L1,
%                   can be added, so as to put all of them on the table.

% domino(+,-)
%   Pre = {domino(L1,L2)|L1 ∈ ListS}
%   Post = {domino(L1,L2)|L1,L2 ∈ ListS ∧ L2 = dominoSol(L1,∅)}
%   | domino(L1,L2) |= 2 · ||L1|| + 1
domino([],[]).
domino([stone(X,Y)|L1],[stone(X,Y)|L2]) :-
      (X =:= Y),
      rest(L1,[X,X,X],L2).
domino([stone(X,Y)|L1],[stone(X,Y)|L2]) :-
      (X=\=Y),
      rest(L1,[X,Y],L2).
DELAY domino(L,_) UNTIL GROUND(L)

% rest(+,+,-)
%   Pre = {rest(L1,Free,L2)|L1 ∈ ListS ∧ Free ∈ αList ∧ Free ≠ [ ]}
%   Post = {rest(L1,Free,L2)|L1,L2 ∈ ListS ∧ Free ∈ αList  ∧
%              L2 = dominoSol(L1,ms(Free))}
%   | rest(L1,Free,L2) |= 2 · ||L1|| + ||Free||
rest([],F,[]).
rest([S1|L1],Free,[S2|L2]) :-
      select(S2,[S1|L1],LR),
      free_sides(S2,Free,NFree),
      rest(LR,NFree,L2).
DELAY rest(L,F,_) UNTIL GROUND(L) & GROUND(F)

% free_sides(+,+,-)
%   Pre = {free_sides(S,Free,NFree)|S ∈ Stones(α) ∧ Free ∈ αList ∧ Free ≠ [ ]}
%   Post = {free_sides(S,Free,NFree)|S ∈ Stones(α) ∧ Free,NFree ∈ αList ∧ Free ≠ [ ]
%              ∧  ∃b ∈ {1,2}   |   ms(NFree) = ms(Free) ⊕ h(S,b)}
%   | free_sides(S,Free,NFree) |= ||Free||
free_sides(stone(X,X),F,[X|F]) :-
      member(X,F).
free_sides(stone(X,Y),F,[Y|F1]) :-
      (X=\=Y),
      member(X,F),
      select(X,F,F1).
free_sides(stone(X,Y),F,[X|F1]) :-
      (X=\=Y),
      member(Y,F),
      select(Y,F,F1).
DELAY free_sides(S,F,_) UNTIL GROUND(S) & GROUND(F)

% member(+,+)
%   Pre = {member(X,Xs)|X ∈ α ∧ Xs ∈ αList}
```

% $Post = \{member(X, Xs) | X \in \alpha \wedge Xs \in \alpha List \wedge X \in ms(Xs)\}$
% $| \ member(X, Xs) \ | = ||Xs||$

```
member(X,[X|Xs]).
member(X,[Y|Xs]) :-
      member(X,Xs).
DELAY member(X,Xs) UNTIL GROUND(X) & GROUND(Xs)
```

% select(-,+,-)
% $Pre = \{select(X, Xs, Ys) | Xs \in ListS\}$
% $Post = \{select(X, Xs, Ys) | X \in Stones(\alpha) \wedge Xs, Ys \in \alpha ListS \wedge X \in ms(Xs) \ \wedge$
% $\qquad ms(Xs) \backslash \{X\} = ms(Ys)\}$
% $| \ select(X, Xs, Ys) \ | = ||Xs||$

```
select(X,[X|Xs],Xs).
select(X,[Y|Xs],[Y|Zs]) :-
      select(X,Xs,Zs).
DELAY select(_,Xs,_) UNTIL GROUND(Xs)
```

% =:=(+,+)
% $Pre = \{X =:= Y | X, Y \in \alpha\}$
% $Post = \{X =:= Y | X, Y \in \alpha \wedge X = Y\}$
% $| \ X =:= Y \ | = 0$

```
DELAY X=:=Y UNTIL GROUND(X) & GROUND(Y)
```

% =\=(+,+)
% $Pre = \{X = \backslash = Y | X, Y \in \alpha\}$
% $Post = \{X = \backslash = Y | X, Y \in \alpha \wedge X \neq Y\}$
% $| \ X = \backslash = Y \ | = 0$

```
DELAY X=\=Y UNTIL GROUND(X) & GROUND(Y)
```

Next we will take into account the topics outlined in [2, 3, 4].

3 Occur-check freedom

As a preliminary remark, let us recall some basic definitions.

Definition 3.1 [linearity]

- A family of terms is called *linear* if every variable occurs at most once in it.
- An atom is called *input* (resp. *output*) *linear* if the family of terms occurring in its input (resp. output) positions is linear.

In [2], two methods for establishing occur-check freedom are outlined. One requires the program to be *nicely moded*. This happens when for every clause of the program the nicely modedness property holds. The definition of *nicely modedness*, reported for instance in [2], requires among others that no variable appearing in the input positions of the head appears also in an output position of (the atoms) in the body. Unfortunately, this is not the case here, because some clauses occurring in the definition of predicate `free_sides`, reported in the sequel, are *not* nicely moded.

```
% free_sides(+,+,-)
free_sides(stone(X,X),F,[X|F]) :-
   member(X,F).
free_sides(stone(X,Y),F,[Y|F1]) :-
   (X=\=Y),
   member(X,F),
   select(X,F,F1).
free_sides(stone(X,Y),2,F,[X|F1]) :-
   (X=\=Y),
   member(Y,F),
   select(Y,F,F1).
DELAY free_sides(E,F,_) UNTIL GROUND(E) & GROUND(F)
```

Namely, in the second (resp. third) clause, variable X (resp. Y) plays a input role in the head and, by means of predicate `select`, an output role in the body.
A possible way out seems to rely on changing the modes of predicate `select` (from `(-,+,-)` to `(+,+,-)`) This will ensure the nice-modedness property, but, at the same time, will destroy another property, namely the input linearity of predicate `select`, necessary to apply the theorem reported in [2].
However, another possibility of establishing occur-check freedom is highlighted in [2], that basically exploits (particular) delay declarations. In our case, the chosen delay declarations, requiring the input arguments to be ground, are indeed reasonable. This derives from the problem described so far, that essentially deals with properly handling a list of *ground* elements. A delay declaration is said to *imply the moding* if every atom which satisfies its delay declaration is ground in its input positions. Hence, each delay declaration reported in our program trivially implies the moding.
Furthermore, as can be easily verified, the head of every clause of the program is *output linear*. All this allows us to exploit another theorem proved in [2] and reported below.

Theorem 3.2 *Suppose that:*

- *The head of every clause of P is output linear*
- *The delay declarations imply the moding*

Then all SLD-derivations of $P \cup \{Q\}$ which respect the delay declarations are occur-check free.

Since the two properties above are satisfied, and since the LD-derivation of Prolog is a particular *instance* of the SLD-derivation (the former is obtained by restricting the latter to employ a leftmost selection rule), all the hypotheses of the theorem hold. Hence, as far as our application is concerned, the occur-check test can be safely omitted.

4 Deadlock freedom

We start by recalling the definition of well-modedness, where we employ the notations $p(\overline{u}, \overline{v})$ to denote with $\overline{u}$ (resp. $\overline{v}$) the input (resp. the output) arguments of predicate p.

Definition 4.1 [well-modedness]

- A query $p_1(\overline{s_1}, \overline{t_1}), \ldots, p_n(\overline{s_n}, \overline{t_n})$ is called *well-moded* if for $i \in [1, n]$

$$Var(\overline{s_i}) \subseteq \bigcup_{j=1}^{i-1} Var(\overline{t_j})$$

- A clause $p_0(\overline{t_0}, \overline{s_{n+1}}) \leftarrow p_1(\overline{s_1}, \overline{t_1}), \ldots, p_n(\overline{s_n}, \overline{t_n})$ is called *well-moded* if for $i \in [1, n+1]$

$$Var(\overline{s_i}) \subseteq \bigcup_{j=0}^{i-1} Var(\overline{t_j})$$

- A program is called *well-moded* if every clause of it is.

Intuitively speaking, the property requires that no variable appears in the body in an input position without having appeared previously in an output position (in the body). In addition, all variables in the output positions of the head must appear in an input position of the head, or in an output position of the body. It is trivial to show that the well-modedness property holds for program `SOLITAIRE` and for the intended query `domino(l,S)`. As an example let us see the definition of predicate `domino(L,s)`. The first clause contains no variable. In the second clause we have :

$Var(\overline{t_0}) = \{X, Y, L1\}$, $Var(\overline{s_3}) = \{X, Y, L2\}$
$Var(\overline{s_1}) = \{X, Y\}$, $Var(\overline{t_1}) = \emptyset$
$Var(\overline{s_2}) = \{X, L1\}$, $Var(\overline{t_2}) = \{L2\}$

Hence the relations $Var(\overline{s_1}) \subseteq Var(\overline{t_0})$, $Var(\overline{s_2}) \subseteq Var(\overline{t_0}) \cup Var(\overline{t_1})$ and $Var(\overline{s_3}) \subseteq Var(\overline{t_0}) \cup Var(\overline{t_1}) \cup Var(\overline{t_2})$ hold.

The same reasoning can be applied to the third clause. The proof that the property holds also for the other predicates is straightforward.
Furthermore, the delay declarations chosen *are implied by the moding* (i.e., they all exactly require the atoms to be ground in their input positions). These observations allow us to exploit the following theorem (taken from [2]).

Theorem 4.2 (Absence of deadlock) *Let P and Q be well-moded. Suppose that the delay declarations are implied by the moding. Then $P \cup \{Q\}$ does not deadlock.*

Since our delay declarations are implied by the moding, and since the queries we are interested in are trivially well-moded, being of kind `domino(t,L)` where `t` is a list of ground terms, we obtain that `SOLITAIRE` $\cup$ `domino(t,L)` is deadlock free.
It is worth noting that well-modedness provides a further way to prove occur-check freedom, according to the theorem proved in [1] and reported below.

Theorem 4.3 *Let P and Q be well-moded. Suppose that the head of every clause in P is output linear. Then $P \cup \{Q\}$ is occur-check free.*

The proof of this theorem exploits the following fact: if P and Q are well-moded, then all LD-derivations of $P \cup \{Q\}$ are *data driven*, i.e. all atoms selected in them are ground in their input positions. Note that Theorem 4.3 is weaker than Theorem 3.2, because the latter guarantees occur-check freedom for all SLD-derivations, while the former only deals with LD-derivations. On the other hand, Theorem 4.3 can be useful when delay declarations are not supported.

5 Absence of run-time errors

It is easy to notice that the delay declarations attached to the program clauses ensure that the comparison operators (=:= and =\=) employed in the program can not produce run-time errors.

Indeed, both the arguments will be ground before the comparison is performed, so guaranteeing a safe comparison.
Moreover, in order to check that $e_1 \neq e_2$, Prolog reduces to evaluate the *negation as failure* of the equality $e_1 = e_2$. However, ensuring the delay declarations that both arguments will be ground before the check is performed, no risk of floundering arises.

6 Parallelizability through DELAYS

The way the program is written seems to prevent a significant gain by means of a parallel execution, since the delay declarations impose quite *strong* conditions. Indeed, the *stream of information* and the algorithm chosen rather follow a sequential order.

7 Partial and Total Correctness

We exploit the relation $\vdash_t$ on triples $\{Pre\}P\{Post\}$ and $\{Pre\}Q\{Post\}$, illustrated in [4], to reason about total correctness of the program `SOLITAIRE`. For a program P, the proof that $\vdash_t \{Pre\}P\{Post\}$ holds can be given for each clause $A_0 \leftarrow A_1, \ldots, A_n$ of P, and $|\ |$, Pre, $Post$, by means of a proof outline of the form

$$
\begin{array}{llll}
\quad \{\ g_0\ \} & & & \\
A_0 & \leftarrow & & \{\ t_0\ \} \\
& & \{\ g_1\ \} & \\
& A_1, & & \{\ t_1\ \} \\
& & \{\ f_1\ \} & \\
& \vdots & & \\
& & \{\ g_{n-1}\ \} & \\
& A_{n-1}, & & \{\ t_{n-1}\ \} \\
& & \{\ f_{n-1}\ \} & \\
& & \{\ g_n\ \} & \\
& A_n. & & \{\ t_n\ \} \\
& & \{\ f_n\ \} & \\
\quad \{\ f_0\ \} & & &
\end{array}
$$

where t_i are integer expressions, and g_i, f_i are first-order formulas, $i \in [0, n]$, such that all ground instances of the following relations are satisfied:

(i) for $i \in [0, n]$: $t_i =| A_i |$

(ii) for $i \in [0, n]$: $g_i \Leftrightarrow A_i \in Pre$

(iii) for $i \in [0, n]$: $f_i \Leftrightarrow A_i \in Post$

(iv) for $i \in [1, n]$:

$$g_0 \wedge f_1 \wedge \cdots \wedge f_{i-1} \Rightarrow g_i \wedge \begin{cases} t_0 > t_i & \text{if } rel(A_0) \simeq rel(A_i) \\ t_0 \geq t_i & \text{otherwise} \end{cases}$$

(v) $g_0 \wedge f_1 \wedge \cdots \wedge f_n \Rightarrow f_0$

where $\simeq$ relates mutually recursive predicates.
For a query Q, $\vdash_t \{Pre\}Q\{Post\}$ holds iff there exists a natural number k such that for each ground instance $A_1, \ldots, A_n$ of Q, for $i \in [1, n]$

$$Post \models A_1, \ldots, A_{i-1} \Rightarrow Pre \models A_i \ \wedge \ k >\mid A_i \mid .$$

The following proof outlines show that $\vdash_t \{Pre\}$SOLITAIRE$\{Post\}$ holds with the level mapping and the couple $(Pre, Post)$ specified in Section 2.2. In the proof outlines we will omit $\{t_i\}$ either if $|t_i| = 0$ or in the case of unit clauses, since it is of no use. Furthermore, we will omit proof outlines of unit clauses when obvious.

$\{ \ [stone(X,Y)|L1] \in ListS \ \}$
`domino([stone(X,Y)|L1],[stone(X,Y)|L2])` $\leftarrow$ $\{ \ 2(||L1||+1)+1 \ \}$
$\{ \ X, Y \in \alpha \ \}$
`X =:= Y,`
$\{ \ X, Y \in \alpha \ \ \wedge X = Y \ \}$
$\{ \ L1 \in ListS \wedge [X, X, X] \in \alpha List \ \}$
`rest(L1,[X,X,X],L2).` $\{ \ 2||L1||+3 \ \}$
(a) $\{ \ L1, L2 \in ListS \wedge [X, X, X] \in \alpha List \wedge L2 = dominoSol(L1, ms([X, X, X])) \ \}$
(b) $\{ \ [stone(X,Y)|L1], [stone(X,Y)|L2] \in ListS \wedge$
$[stone(X,Y)|L2] = dominoSol([stone(X,Y)|L1], \emptyset) \ \}$

Note that (b) immediately follows from (a) by noting that, when $X = Y$
$\emptyset \oplus \langle X, Y\rangle = \{X, X, X\} = ms([X, X, X])$.

$\{ \ [stone(X,Y)|L1] \in ListS \ \}$
`domino([stone(X,Y)|L1],[stone(X,Y)|L2])` $\leftarrow$ $\{ \ 2(||L1||+1)+1 \ \}$
$\{ \ X, Y \in \alpha \ \}$
`X =\=Y,`
$\{ \ X, Y \in \alpha \wedge X \neq Y \ \}$
$\{ \ L1 \in ListS \wedge [X, Y] \in \alpha List \ \}$
`rest(L1,[X,Y],L2).` $\{ \ 2||L1||+2 \ \}$
$\{ \ L1, L2 \in ListS \wedge [X, Y] \in \alpha List \wedge L2 = dominoSol(L1, ms([X, Y])) \ \}$
$\{ \ [stone(X,Y)|L1], [stone(X,Y)|L2] \in ListS \wedge$
$[stone(X,Y)|L2] = dominoSol([stone(X,Y)|L1], \emptyset) \ \}$

The proof outline for the (instances of the) third clause is omitted, because of its deep analogy with (the instances of) the second clause, the only difference being the roles of X and Y that are swapped.

$\{ \ [S1|L1] \in ListS \wedge Free \in \alpha List \ \ \wedge Free \neq []\}$
`rest([S1|L1],Free,[S2|L2])` $\leftarrow$ $\{ \ 2(||L1||+1) + ||Free|| \ \}$
$\{ \ [S1|L1] \in ListS \ \}$
`select(S2,[S1|L1],LR),` $\{ \ ||L1||+1 \ \}$
(a) $\{ \ S2 \in Stones(\alpha) \wedge [S1|L1], LR \in ListS \wedge S2 \in ms([S1|L1]) \wedge$
$ms([S1|L1]) \backslash \{S2\} = ms(LR) \ \}$
$\{ \ S2 \in Stones(\alpha) \wedge Free \in \alpha List \wedge Free \neq [\ ] \ \}$
`free_sides(S2,Free,NFree),` $\{ \ ||Free|| \ \}$
(b) $\{ \ S2 \in Stones(\alpha) \wedge Free, NFree \in \alpha List \wedge Free \neq [\ ] \wedge$

$\exists b \in \{1,2\}$ $s.t.$ $ms(NFree) = ms(Free) \oplus h(S2,b)$ }

{ $LR \in ListS \wedge NFree \in \alpha List$ }

`rest(LR,NFree,L2).` { $2||LR|| + ||NFree||$ }

(c) { $LR, L2 \in ListS \wedge NFree \in \alpha List \wedge L2 = dominoSol(LR, ms(NFree))$ }

(d) { $[S1|L1], [S2|L2] \in ListS \wedge Free \in \alpha List \wedge [S2|L2] = dominoSol([S1|L1], ms(Free))$ }

To prove (d), note that $[S2|L2]$ is a permutation of $[S1|L1]$ because $L2$ is a permutation of LR (from (c)) and $ms(LR) = ms([S1|L1]) \backslash \{S2\}$ (from (a)). Furthermore, for $[S2|L2]$ the property (1) of domino solutions directly follows from (b) and (c). Note that, if l_1, l_2 are lists of elements of α and $s \in ListOS$,

$$ms(l_1) = ms(l_2) \oplus s \;\Rightarrow\; ||l_1|| \leq ||l_2|| + 1.$$

Therefore,

$$2||LR|| + ||NFree|| = 2||L1|| + ||NFree|| \leq 2||L1|| + ||Free|| + 1 < 2||L1|| + 2 + ||Free||.$$

{ $stone(X,X) \in Stones(\alpha) \wedge F \in \alpha List \wedge F \neq [\ ]$ }

`free_sides(stone(X,X),F,[X|F])` $\leftarrow$ { $||F||$ }

{ $X \in \alpha \wedge F \in \alpha List$ }

`member(X,F).` { $||F||$ }

{ $X \in \alpha \wedge F \in \alpha List \wedge X \in ms(F)$ }

{ $stone(X,X) \in Stones(\alpha) \wedge F, [X|F] \in \alpha List \wedge F \neq [\ ]$

$\wedge \exists b \in \{1,2\}$ $s.t.$ $ms([X|F]) = ms(F) \oplus h(stone(X,X),b)$ }

The post-condition is satisfied because

1. $h(stone(X,X),b) = \langle X,X \rangle \; \forall b \in \{1,2\}$;
2. $(F \neq [\ ] \wedge X \in ms(F)) \;\Rightarrow\; ms(F) \oplus \langle X,X \rangle = ms(F) \cup \{X\} = ms([X|F])$.

{ $stone(X,Y) \in Stones(\alpha) \wedge F \in \alpha List \wedge F \neq [\ ]$ }

`free_sides(stone(X,Y),F,[Y|F1])` $\leftarrow$ { $||F||$ }

{ $X, Y \in \alpha$ }

`X =\=Y,`

{ $X, Y \in \alpha \wedge X \neq Y$ }

{ $X \in \alpha \wedge F \in \alpha List$ }

`member(X,F),` { $||F||$ }

{ $X \in \alpha \wedge F \in \alpha List \wedge X \in ms(F)$ }

{ $F \in \alpha List$ }

`select(X,F,F1).` { $||F||$ }

{ $X \in \alpha \wedge F, F1 \in \alpha List \wedge X \in ms(F) \wedge ms(F1) = ms(F) \backslash \{X\}$ }

{ $stone(X,Y) \in Stones(\alpha) \wedge F, [Y|F1] \in \alpha List \wedge F \neq [\ ]$

$\wedge \exists b \in \{1,2\}$ $s.t.$ $ms([Y|F1]) = ms(F) \oplus h(stone(X,Y),b)$ }

The post-condition is satisfied for $b = 1$, because, if $F \neq [\,]$ and $X \in ms(F)$, then $ms(F)\backslash\{X\} \cup \{Y\} = ms(F1) \cup \{Y\} = ms([Y|F1])$.
The proof outline for the last clause of `free_sides` is analogous to the one above, and hence omitted. Indeed, the unique differences are the roles of X and Y, that are swapped, and the value of b.

$\{\ X \in \alpha \wedge [X|Xs] \in \alpha List\ \}$
`member(X,[X|Xs]).`
$\{\ X \in Gt \wedge [X|Xs] \in \alpha List \wedge X \in ms([X|Xs])\ \}$

$\{\ X \in \alpha \wedge [Y|Xs] \in \alpha List\ \}$
`member(X,[Y|Xs])` $\leftarrow$ $\{\ ||Xs|| + 1\ \}$
$\{\ X \in \alpha \wedge Xs \in \alpha List\ \}$
`member(X,Xs).` $\{\ ||Xs||\ \}$
$\{\ X \in \alpha \wedge Xs \in \alpha List \wedge X \in ms(Xs)\ \}$
$\{\ X \in \alpha \wedge [Y|Xs] \in \alpha List \wedge X \in ms([Y|Xs])\ \}$

$\{\ [X|Xs] \in \alpha List\ \}$
`select(X,[X|Xs],Xs).`
$\{\ X \in \alpha \wedge [X|Xs], Xs \in \alpha List \wedge X \in ms([X|Xs]) \wedge ms(Xs) = ms([X|Xs])\backslash\{X\}\ \}$

$\{[Y|Xs] \in \alpha List\ \}$
`select(X,[Y|Xs],[Y|Zs])` $\leftarrow$ $\{\ ||Xs|| + 1\ \}$
$\{\ Xs \in \alpha List\ \}$
`select(X,Xs,Zs).` $\{\ ||Xs||\ \}$
$\{\ X \in \alpha \wedge Xs, Zs \in \alpha List \wedge X \in ms(Xs) \wedge ms(Zs) = ms(Xs)\backslash\{X\}\ \}$
$\{\ X \in \alpha \wedge [Y|Xs], [Y|Zs] \in \alpha List \wedge X \in ms([Y|Xs]) \wedge ms([Y|Zs]) = ms([Y|Xs])\backslash\{X\}\ \}$

The previous proof outlines show that the following relation holds:

$$\vdash_t \{Pre\}\,\texttt{SOLITAIRE}\,\{Post\}\,.$$

where Pre (resp. $Post$) includes the Pre- (resp. $Post$-) conditions of all predicates of the program. Therefore, according to [4], `SOLITAIRE` is *weak totally correct* w.r.t. $(Pre, Post)$, i.e.

(*i*) $M_{\texttt{SOLITAIRE}} \cap Pre \subseteq Post$

(*ii*) for each query Q such that $\vdash_t \{Pre\}Q\{Post\}$, every LD-tree of P and Q is finite.

It is easy to verify that the property $\vdash_t \{Pre\}Q\{Post\}$ is satisfied by every query $Q =$`domino(t,L)` where `t`$\in ListS$, because $Q \in Pre$ and $|Q| = 2||\texttt{t}|| + 1$ is bounded, `t` being ground.
The relation $\vdash_t$ allows us to reason about correctness and termination under a single framework. However, it is also possible to directly prove left-termination of the program `SOLITAIRE` by exploiting the notion of *semi-acceptability*, introduced in [3].

8 Characterization of computed instances

In order to reason about the relation between correct and computed instances of our program, we first recall the property of well-supportedness [4] and then we prove that this property holds for our program.

8.1 Well supportedness

Definition 8.1 Assume that $\vdash \{Pre\}\ P\ \{Post\}$ holds.
Post is a *well-supported interpretation* (w.r.t. P and Pre) iff there exist

- a well-founded poset $(W, <)$, and
- a function $|\ | : B_L \rightarrow W$

such that for any $A \in Post \cap Pre$ there exists $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ such that

$$\forall i \in [1, n] : Post \models B_i \wedge |A| > |B_i|.$$

□

Roughly, the idea underlying this definition is that each atom belonging to the intersection of Pre and $Post$ has a successful ground derivation.

As highlighted in [4], proof outlines for well-supportedness in the case of total correctness (i.e., under the assumption that $\vdash_t \{Pre\}\ P\ \{Post\}$ already holds) are simpler than those in the case of partial correctness .

Definition 8.2 A proof outline $\mathcal{PO}$ for a clause $A \leftarrow B_1, \ldots, B_n$ and Pre, $Post$, is a construct of the form

$$\begin{array}{lll} & \{g\} & \\ A & & \leftarrow \\ & & B_1, \\ & & \quad \{f_1\} \\ & & \vdots \\ & & B_{n-1}, \\ & & \quad \{f_{n-1}\} \\ & & B_n. \\ & & \quad \{f_n\} \end{array}$$

where f_i, g, for $i \in [0, n]$ are assertions (in some formal logic), such that every ground instance of the following *proof obligations* holds:

(i) for $i \in [1, n]$: $g \wedge f_i \Rightarrow B_i \in Post$,

(ii) for $i \in [1, n]$: $g \Rightarrow f_i$.

□

Let us see how we can apply the above schema to our program.

$\{\ true\ \}$

```
domino([],[]).
```

$\{\ stone(X,Y) \in Stones(\alpha)\ \wedge\ X = Y\ \wedge L1, L2 \in ListS \wedge$
$[stone(X,Y)|L2] = dominoSol([stone(X,Y)|L1], \emptyset)\}$

```
domino([stone(X,Y)|L1],[stone(X,Y)|L2])    ←
      X =:= Y,
```

$\{\ X = Y\ \}$

```
      rest(L1,[X,X,X],L2).
```

$\{\ [X, X, X] \in \alpha List\ \wedge\ L2 = dominoSol(L1, ms([X, X, X]))\ \}$

$\{\ stone(X,Y) \in S \ \wedge \ X \neq Y \ \wedge \ L1, L2 \in ListS$
$\wedge\ [stone(X,Y)|L2] = dominoSol([stone(X,Y)|L1], \emptyset)\}$

```
domino([stone(X,Y)|L1],[stone(X,Y)|L2])   ←
    X =\=Y,
```

$\{\ X, Y \in \alpha\ \}$

```
    rest(L1,[X,Y],L2).
```

$\{\ [X,Y] \in \alpha List \ \wedge \ L2 = dominoSol(L1, ms([X,Y]))\ \}$

The proof outline for the (instances of the) third clause is omitted, because is analogous to the second one (the only difference being the roles of X and Y that are swapped).
Moreover, it is worth observing that all the possible heads belonging to $Post \cap Pre$ are taken into account. Indeed, the initial list can be empty (case handled by the first unit clause), or may begin with an element with two equal sides (i.e., case handled by the second clause) or with two different sides (case handled by the third clause).

$\{\ F \in \alpha List\ \}$

```
rest([],F,[]).
```

$\{\ [S1|L1] \in ListS \wedge Free, NFree \in \alpha List \ \wedge \ [S2|L2] = dominoSol([S1|L1], ms(Free)) \ \wedge$
$\exists b \in \{1,2\} \mid \ ms(Free) \oplus h(S2,b) = ms(NFree) \ \wedge \ ms([S1|L1]) \backslash \{S2\} = ms(LR)\ \}$

```
rest([S1|L1],Free,[S2|L2])   ←
    select(S2,[S1|L1],LR),
```

$\{\ S2 \in Stones(\alpha) \ \wedge \ S2 \in ms([S1|L1])$

```
    free_sides(S2,Free,NFree),
```

$\{\ S2 \in Stones(\alpha) \ \wedge \ \exists b \in \{1,2\} \ \mid \ ms(NFree) = ms(Free) \oplus h(S2,b)\ \}$
$\{\ LR \in ListS \wedge NFree \in \alpha List\ \}$

```
    rest(LR,NFree,L2).
```

$\{\ L2 = dominoSol(LR, ms(NFree))\ \}$

Here we observe that $(ms(Free) \oplus s_2) \oplus s_3 \oplus ... \oplus s_n = ms(NFree) \oplus s_3 \oplus ... \oplus s_n$
and that $[s_3, \ldots, s_n]$ is a permutation of the list LR. Moreover, also the other conditions required to be a *dominoSol* are satisfied. Hence $L2 = dominoSol(LR, ms(NFree))$.
Again, it is easy to show that all the possible heads belonging to $Post \cap Pre$ are justified, since the first clause deals with the empty list, and the second (resp. third) deals with a list beginning with a $stone(X,X)$ (resp. $stone(X,Y)$) element.

$\{\ stone(X,X) \in Stones(\alpha) \wedge F \in \alpha List \ \wedge X \in ms(F)\ \}$

```
free_sides(stone(X,X),F,[X|F])   ←
    member(X,F).
```

$\{\ X \in \alpha\ \}$

$\{\ stone(X,Y) \in Stones(\alpha) \wedge X \neq Y \wedge F \in \alpha List \wedge X \in ms(F) \wedge ms(F) \backslash \{X\} = ms(F1)\ \}$

```
free_sides(stone(X,Y),F,[Y|F1])   ←
    X =\=Y,
```

$\{\ X, Y \in \alpha\ \}$

```
    member(X,F),
    select(X,F,F1).
```

$\{\ F1 \in \alpha List\ \}$

The proof outline for the last clause of `free_sides` is analogous to the one reported above, and hence omitted.
It is easy to notice that the three clauses cover all the cases. Indeed, for each element of the kind $stone(X, Y)$ either:
(a) X equals Y (case captured by the first clause)
(b) X differs from Y (case captured by the second and third clauses).

$\{\ [X|Xs] \in \alpha List\ \}$
```
member(X,[X|Xs]).
```

$\{\ [Y|Xs] \in \alpha List \wedge X \in Xs\ \}$
```
member(X,[Y|Xs])    ←
      member(X,Xs).
```

$\{\ [X|Xs] \in \alpha List\ \}$
```
select(X,[X|Xs],Xs).
```

$\{\ X, Y \in \alpha \wedge Xs, Zs \in \alpha List\ \ \wedge X \in Xs\}$
```
select(X,[Y|Xs],[Y|Zs])    ←
      select(X,Xs,Zs).
```

The well-supportedness property can be exploited to characterise the strongest postcondition of program `SOLITAIRE`:

Theorem 8.3 *Assume that $\vdash_t \{Pre\}\ P\ \{Post\}$ holds. Then the following statements are equivalent:*

(i) $Post \cap Pre = sp(P, Pre)$,

(ii) $T_P(Post) \supseteq Post \cap Pre$,

(iii) $Post$ *is well-supported (w.r.t.* P *and* Pre. *)*

In our case, we thus directly obtain (the hypothesis is satisfied, and case (iii) holds) that $Post \cap Pre = sp(P, Pre)$ where $sp(P, Pre)$ denotes the strongest postcondition.
This is sufficient to prove *total correctness*, by means of the following result:

Theorem 8.4 *If $\vdash_t \{Pre\}\ P\ \{Post\}$ holds, then P is totally correct w.r.t. the specification $(Pre, sp(P, Pre))$.*

Since, in our case, $Post \cap Pre = Post$, we can state that P is totally correct w.r.t. $(Pre, Post)$, i.e. $Post$ coincides with the correct instances of the intended queries ($M_{\texttt{SOLITAIRE}} \cap Pre = Post$).

Moreover, well-supportedness also helps us to characterize the relation between correct and computed instances, as the following theorem highlights.

Theorem 8.5 *Assume that Σ_L contains infinitely many symbols, and $\vdash \{Pre\}\ P\ \{Post\}$, $\vdash \{Pre\}\ Q\ \{Post\}$ hold with $Post$ well-supported. Consider the set $\mathcal{Q} = \{Q\theta \mid Post \models Q\theta\}$:*

(i) $\mathcal{Q}$ is the set of correct instances of P and Q,

(ii) each of following conditions is sufficient to state that $\mathcal{Q}$ is the set of computed instances of P and Q:

(a) $Min(\mathcal{Q}) = \mathcal{Q}$,

(b) $\mathcal{Q}$ is a set of ground queries,

(c) $\mathcal{Q}$ is a finite set.

In our case, the hypotheses of the theorem are all satisfied. Indeed, we can assume the underlying language Σ_L to contain infinitely many symbols. Moreover, we have shown that $\vdash_t \{Pre\}\ P\ \{Post\}$ holds, and this obviously implies that $\vdash \{Pre\}\ P\ \{Post\}$ holds, too. In addition, also $\vdash \{Pre\}\ Q\ \{Post\}$ trivially holds in our case.
Now, it is easy to notice that $\mathcal{Q}$ is a finite set, since it is a subset of the permutations of a ground list of finite length. Hence we obtain that in our case the sets of correct and computed instances coincide.

9 Conclusions

The problem tackled in these pages, although quite easy, can indeed be part of a set of benchmark examples useful to test the results studied, among others, in [1, 2, 3, 4]. The effective application of the suggested techniques guarantees a series of crucial properties, such as correctness, termination, absence of errors, all of whom are really desirable in any computational environment. The constraints imposed on the program to satisfy the various properties seem reasonable in many, if not all, cases. From this point of view, our problem does not represent an exception. Moreover, the constraint imposed in order to satisfy such properties do not affect, in general, the natural writing of the program. As a unique exception to this, we can point out the use of unification to (implicitly) test the syntactical equality of two terms. Using a term like $stone(X, X)$ is a typical way programmers exploit unification. This can provoke some troubles when considering input and output linearity. However, many extensions and refinements of logic programming (e.g., constraint logic programming) require the variables in the head of each clause to be all different. So, it is now more common to accept these kind of restrictions. On the other side, it is worth observing that the various results that exploit these forms of linearity resort on necessary conditions. It could then be interesting to look for less constraining conditions or at least to investigate the possibility of weakening the requested conditions for a subset of the program clauses (e.g., for particular, "safe" cases).

References

[1] K. Apt. *From Logic Programming to Prolog.* CAR Hoare Series Editor. Prentice Hall, 1996.

[2] K. Apt and I. Luitjes. Verification of Logic Programs with Delay Declaration. In V.S. Alagar and M. Nivat, editors, *Proc. of AMAST'95*, number 936 in LNCS, pages 66–90. Springer-Verlag, 1995.

[3] K. Apt and D. Pedreschi. Reasoning about termination of pure prolog programs. *Journal of Logic and Computation*, 106(1):109–157, 1993.

[4] S. Ruggieri and D. Pedreschi. Verification of Logic Programs. *Submitted for publication*, 1997.