Basic format for solving problems through a reduction to SAT

In order to solve a problem P by reducing it to SAT, we write two programs that can handle any possible input I of P:

- **reduction**
  This program translates the received input I of P into a boolean formula F. If F is satisfiable, then the following program is executed, otherwise nothing else is done.

- **reconstruction**
  This program queries a model $\mathcal{M}$ that satisfies F (i.e., an assignment of a truth value to each free propositional variable in F that makes F evaluate to true), and reconstructs a solution S for the received input I of P.

An example will illustrate the language used to describe these programs, called REDSAT. Consider the following problem P: given a graph, paint its nodes with colors 1, 2, 3 in a way such that no two adjacent nodes have the same color. Thus, the input I of P is a graph $G = \langle V, E \rangle$, and the solution S for I is a color assignment that satisfies the desired condition. In REDSAT, I and S are stored in predefined global variables. In this example the graph G of I is represented by these two globals:

```
numnodes: int
edges: array of array [2] of int
```

and the solution S is represented by this global:

```
coloring: array of int
```

Before presenting the **reduction** and **reconstruction** programs, we formally describe the reduction using propositional variables of the form $p_{n,c}$ to denote that node n has color c. The reduction has three parts. First, it states that each node has a color, second, a node cannot have more than one color, and third, adjacent nodes cannot have the same color:

$$\bigwedge_{n \in V} \left(p_{n,1} \vee p_{n,2} \vee p_{n,3} \right)$$
$$\bigwedge_{n \in V} \left(\neg(p_{n,1} \wedge p_{n,2}) \wedge \neg(p_{n,1} \wedge p_{n,3}) \wedge \neg(p_{n,2} \wedge p_{n,3}) \right)$$
$$\bigwedge_{(n_1,n_2) \in E} \bigwedge_{c \in \{1,2,3\}} \neg(p_{n_1,c} \wedge p_{n_2,c})$$

Using the previous reduction, the REDSAT solution for this example exercise is as follows:

```
reduction {
  for (n=0 ; n<numnodes ; ++n) {
    "node{n}has{1}"  or  "node{n}has{2}"  or  "node{n}has{3}";
    not ("node{n}has{1}"  and  "node{n}has{2}");
    not ("node{n}has{1}"  and  "node{n}has{3}");
    not ("node{n}has{2}"  and  "node{n}has{3}");
  }
  foreach (edge in edges)
    for (c=1 ; c<=3 ; ++c)
      not ("node{edge[0]}has{c}"  and  "node{edge[1]}has{c}");
}
reconstruction {
  for (n=0 ; n<numnodes ; ++n)
    for (c=1 ; c<=3 ; ++c)
      if ("node{n}has{c}")
        coloring[n] = c;
}
```

Note that, in REDSAT, the boolean formula F is constructed incrementally, by individually stating certain parts of it (to state a formula we simply write it out as a statement). Also, observe that in REDSAT we write the propositional variables $p_{n,c}$ as `"node{n}has{c}"`: with this syntax we define the name of a propositional variable, where `{n}` and `{c}` are to be replaced by the value of `n` and `c` at the point of execution where `"node{n}has{c}"` appears. All such propositional variables are free in the **`reduction`** program, but in **`reconstruction`** they are bound to the value assigned to them by a model $\mathcal{M}$ that satisfies F.

1 Variables, types, and expressions

There are three distinct kinds of variables in REDSAT:

Local variables These variables do not need to be declared, since they are created on the first assignment. They can only contain values with the simple types **`int`** and **`formula`**, standing for integer and boolean formula, respectively. Integers are as in other programming languages, but boolean formulas are a rather atypical type: besides the values **`true`** and **`false`**, in REDSAT a value of type **`formula`** can also be an object that represents a boolean formula with free propositional variables.

Global variables These variables are predefined for each exercise, and are either *input* variables (accessible by **`reduction`** and **`reconstruction`**) or *output* variables (accessible only by **`reconstruction`**). As the name suggest, an input variable is read-only, that is, it can be read but not assigned, whereas an output variable is write-only, that is, it can be assigned but not read. The names and types of these variables are specified in the statement of the exercise, and are checked at execution time. Therefore, even if the program must construct the output, it has to do so within the bounds defined by its type. A type of a global variable can be anyone of:

- **`int`**, an integer.
- **`array of`** `T`, an array of variable size with cells of type `T`.
- **`array`** `[N]` **`of`** `T`, an array of size `N` with cells of type `T`.
- **`struct`** `{ f1: T1 ... fN: TN }`, a structure (inspired by the C language) of fields `f1` to `fN` with types `T1` to `TN`, respectively.

Access to fields in structures is denoted using a dot (`.`), just like in the C language. Access to cells in arrays is denoted using brackets (`[]`). Indices start at 0, and thus, `edges[0]` is the first element of the array `edges`. Also, the size of the array can be obtained like `edges.size`.

Aliases To ease the access to deep structures in the input/output global variables, an alias can be created with the **`&=`** operator. For instance, the expression

```
e &= edges[0];
```

defines `e` as an alias for the first edge in the graph, and thus, from that statement onwards, any read or write performed on `e` is translated into a read or write on `edges[0]`. An alias may also be implicitly created by the **`foreach`** iteration, like `edge` in the previous example.

REDSAT implements the typical arithmetic operators for integers (**`+`** addition, **`-`** subtraction, **`*`** multiplication, **`/`** division, **`%`** modulus) with the usual precedence. It also implements the comparison operators for integers (**`>`** greater, **`<`** less, **`<=`** less or equal, **`>=`** greater or equal, **`==`** equal, **`!=`** different), which evaluate either to the boolean formula **`true`** or to **`false`**. Increment (**`++`**) and decrement (**`--`**) are also implemented, both in prefix and postfix form.

Thus, it is possible to write `i++` or `++i`. However, increments and decrements are instructions (not expressions), so assignments like `a = i++` will result in an error. The language also provides three functions on integers: `min(a,b)` returns the minimum among the `a` and `b` integers, `max(a,b)` returns their maximum, and `abs(x)` returns the absolute value of the integer `x`.

REDSAT has a rich set of boolean operators, as the language is specifically designed to ease the construction of boolean formulas. From higher to lower precedence, these operators are: `not` for negation, `and` for conjunction, `or` for disjunction, `implies`/`if` for implication and inverse implication, respectively, `iff` for biconditional. They can be equivalently written as `!`, `&`, `|`, `->`, `<-`, `<->`, respectively. Moreover, it is important to remark that *none* of these operators is evaluated with short-circuit, and that `implies`, `if`, `iff` are not associative. Another important remark is that, in contrast with most programming languages, in REDSAT the boolean operators need not evaluate to `true` or `false`: instead, when the operators receive free propositional variables, they are evaluated to a boolean formula with such free propositional variables. For instance, the expression `"p" implies "q"` evaluates to an object that represents the propositional formula $p \to q$, where p, q are two distinct free propositional variables.

2 Control structures

REDSAT provides `if-else`, `for`, and `while` with the syntax and semantics of the C language. There is also `foreach` to easily iterate arrays. The following codes are equivalent:

```
foreach (i,edge in edges) {
  <<iteration body>>
}
```

```
for (i = 0; i < edges.size; ++i) {
  edge &= edges[i];
  <<iteration body>>
}
```

The index variable of a `foreach` (the `i` in the previous example) may be omitted if it is not needed. To ease the iteration over a range of numbers from `n` (inclusive) to `m` (exclusive), REDSAT provides the following syntax:

```
foreach (i in n...m) {
  <<iteration body>>
}
```

When iterating from `n` to `m`, both inclusive, it is written as:

```
foreach (i in n..m) {
  <<iteration body>>
}
```

Note that the `...` range is empty when $n = m$, whereas the `..` range always contains at least one item, `n`. The range might be increasing ($n < m$) or decreasing ($n > m$), and the iteration will act accordingly.

With the purpose of conditionally terminating the program, a special instruction `stop` halts the execution.

3 Array manipulation

Although the type of the output is specified in advance, arrays within the output which do not have a fixed size have to be grown during the execution of the program `reconstruction`. The method `push` appends a new element to an array, with a type matching the one specified beforehand. For instance, with an output variable `x` of type `array of array [3] of int`, issuing the following instruction

```
x.push;
```

will append an array of 3 elements (all `0`) to the array `x`.

The method **`push`** will in fact return a reference to the newly inserted element, so that in the same instruction the value of this element can be assigned. For instance, with an output variable `x` of type **`array of int`**, issuing the instruction

```
x.push = 42;
```

will append a new element to the array `x` and set it to `42`. As seen in the initial example of coloring a graph, an array can also be grown by simply accessing a still-non-existing position `n` and assigning a value to it:

```
coloring[n] = c;
```

Once an element is pushed to the end of the array, it can be accessed with **`back`**, which returns a reference to the last element of the array. This is useful for constructing nested arrays. For instance, suppose that an output variable `x` has type **`array of array of int`** and it is empty. The following code

```
x.push.push = 0;
x.back.push = 1;
x.back.push = 2;
x.push;
x.back.push = 3;
x.back.push = 4;
x.push;
```

will set `x` to be the array `[[0, 1, 2], [3, 4], []]`.

4 Advanced boolean expressions

In many exercises, the reduction to SAT might require to express a cardinality constraint, that is, express that a specific number of formulas are true. In the initial example of coloring a graph, there is implicitly one such constraint: each node must have *exactly one* color. We have encoded such constraint in a rather simple way: with $p_{n,1} \vee p_{n,2} \vee p_{n,3}$ we guarantee that node n has *at least one* color, and with $\neg(p_{n,1} \wedge p_{n,2}) \wedge \neg(p_{n,1} \wedge p_{n,3}) \wedge \neg(p_{n,2} \wedge p_{n,3})$ we guarantee that it has *at most one* color. REDSAT provides a simpler way to express such constraints by means of special functions:

- **`atleast`**`(k,f1,f2,...,fN)`: returns a formula that is satisfiable if and only if at least `k` of the formulas `f1`, `f2`, ..., `fN` are true.
- **`atmost`**`(k,f1,f2,...,fN)`: returns a formula that is satisfiable if and only if no more than `k` of the formulas `f1`, `f2`, ..., `fN` are true.
- **`exactly`**`(k,f1,f2,...,fN)`: returns a formula that is satisfiable if and only if exactly `k` of the formulas `f1`, `f2`, ..., `fN` are true.

Hence, in the example we could have equivalently written:

```
exactly(1, "node{n}has{1}", "node{n}has{2}", "node{n}has{3}");
```

Two additional functions that return a boolean formula are provided as syntactic sugar:

- **`and`**`(f1,f2,...,fN)`: returns a formula that is satisfiable if and only if all the formulas `f1`, `f2`, ..., `fN` are true (or there is no formula, i.e., `N` is 0).
- **`or`**`(f1,f2,...,fN)`: returns a formula that is satisfiable if and only if any of the formulas `f1`, `f2`, ..., `fN` are true.

In most cases, the previous forms of the **atleast**, **atmost**, **exactly**, **and**, **or** functions will not be very useful, since the number N of formulas will need to vary depending on the input received. For instance, if in the coloring example we had been allowed to choose among K colors instead of just 3 (with K given as an input value), it would not be possible to write an expression **exactly**(1, ...) that receives K parameters, as K is not known beforehand. To solve this limitation, REDSAT admits a very powerful syntax to pass a variable number of parameters: instead of writting formulas as parameters, we can write *arbitrary code* as parameter. For instance, in the coloring example with K colors, we can write:

```
exactly(1,
  foreach (c in 1..K)
    "node{n}has{c}";
);
```

The effect of this statement is that the **exactly** function receives K parameters of the form **"node{n}has{c}"**, with c ranging from 1 to K (both inclusive). Intuitively, the functions **atleast**, **atmost**, **exactly**, **and**, **or** start a scope that captures any formula that is written out as a statement. Inside a scope, any code is allowed: control structures **if**, **for**, **while**, **foreach** are admitted, and even defining/modifying variables or nesting scopes is possible.

As a final example, assume that we are given a global input variable cnf with type **array of array of int** representing a formula in conjunctive normal form (i.e., it is an array of clauses, where each clause is an array of literals, and each literal is either a positive number p representing the propositional variable p or a negative number $-p$ representing the negation of the propositional variable p). Using REDSAT we can easily transform cnf into an actual value of type **formula**, and save it in a local variable f:

```
f = and(
  foreach (clause in cnf)
    or(
      foreach (lit in clause)
        if (lit > 0)
          "var{lit}";
        else
          not "var{abs(lit)}";
    );
);
```

Note that the propositional variables are given names of the form **"var{i}"**, that the literals are stated inside an **or** and thus they are captured in the **or**'s scope, and that each of such **or** is in turn captured into the **and**'s scope. To further illustrate the fact that inside a scope it is possible to even define new local variables, the previous code could also be written as:

```
f = and(
  i = 0;
  while (i < cnf.size) {
    or(
      j = 0;
      while (j < cnf[i].size) {
        lit &= cnf[i][j];
        "var{abs(lit)}" iff (lit > 0);
        j = j + 1;
      }
    );
    i = i + 1;
  }
);
```

Local variables and aliases defined inside a scope are accessible outside the scope.