Format for describing reductions between decisional problems

To describe a reduction from one problem A to another problem B, you write a program that will transform an input of A to an input of B. The program is written in a language similar to C, but specifically designed for this type of reductions, called REDNP.

An example will illustrate the use of REDNP in a particular setting. Given the following types for the variables `in` (the input) and `out` (the output)

```
in: struct {
      k: int
      numnodes: int
      edges: array of array [2] of int
    }
out: struct {
       k: int
       edges: array of array [2] of string
       nodes: array of string
     }
```

the following program

```
main {
  out.k = in.k;
  foreach (edge; in.edges) {
    u = edge[0];
    v = edge[1];
    out.edges.push = u, v;
    out.edges.push = u, "new{u}{v}";
    out.edges.push = v, "new{u}{v}";
  }
}
```

transforms a graph into another graph, converting every edge of the input graph into a little triangle in the output graph. It does this by preserving the old nodes (except the ones that are isolated), and adding a new node for every edge (u, v), which connects to both ends u and v. This is a valid reduction from the VERTEX COVER problem to the DOMINATING SET problem.

1 Variables, types and expressions

Variables in REDNP do not have to be declared, since they are created on the first assignment. Variables can only contain the simple types `int` and `string`.

Two special variables, which exist for the whole execution of the program, are the input (`in`) and the output (`out`). The `in` variable is read-only, that is, it can only accessed, not assigned. On the contrary, `out` variable is write-only, and can only be assigned, not read. The types of these variables are specified in the statement of the problem, and are checked at execution time. Therefore, even if the program must construct the `out` object, it has to do so within the bounds defined by its type. A type can be anyone of:

- `int`, an integer.
- `string`, a string.
- `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. For instance, `in.edges` is the field `edges` in the struct `in`. Access to cells in arrays is denoted using brackets (`[]`). Indices start at 0, and thus, `in.edges[0]` is the first element of the array `in.edges`. Also, the size of the array can be obtained like `in.edges.size`. To make access to deep structures easier, a variable can be assigned a reference to a value deep within the input using the `&=` assignment operator

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

REDNP implements the following operators for integers: `+` addition, `-` subtraction, `*` multiplication, `/` division, `%` modulus, `>` greater, `<` less, `<=` less or equal, `>=` greater or equal, `==` equal, and `!=` different. For strings, REDNP provides equality operators (`==` and `!=`) and concatenation (`+`). Like in C, an integer 0 can also be interpreted as the Boolean false, and any other integer as the Boolean true. The Boolean operators are: `not` for negation, `and` for conjunction, and `or` for disjunction. The latter two are evaluated with short-circuit.

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. Some related operators familiar to C programmers are *not* implemented in REDNP, most notably `+=`, `-=`, `*=`, and `/=`.

Finally, conversions from integers to strings are automatic, so an expression or assignment involving an integer where a string is expected converts the integer to a string. There are two contexts where this might be useful: (i) when naming nodes or literals it is possible to mix strings and integers and name some element `5` and another `"aux5"`, and (ii) when producing new strings by concatenation of both strings and integers (`"aux" + 5` produces `"aux5"`).

2 Control structures

REDNP provides `if-else`, `for`, and `while`, all with the familiar syntax of the C language. In addition, there is also `foreach` to easily iterate arrays. The following piece of code

```
foreach (i,clause; in.clauses) {
  <<iteration body>>
}
```

is equivalent to

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

In the case of `foreach`, the index variable (the `i` in the previous example) may be omitted if it is not needed.

With the purpose of conditionally terminating the program, a special instruction `stop` stops execution and returns the `out` object as it was at that point. This can be useful to deal with special cases in which the output is very simple and the full computation is not necessary.

3 Push and back

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. The method `push` appends a new element to an array, with a type matching the one specified beforehand. For instance, with an output type of `array of array [3] of string`, issuing the following instruction

```
out.push;
```

will append an array of 3 elements (empty strings) to the array `out`.

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 type of `array of string`, issuing the instruction

```
out.push = "x";
```

will append a new element to the array `out` and set it to `"x"`.

In the case that `out` is an array of variable size which contains as elements arrays of variable size (e.g., `out` has type `array of array of int`), one could write a statement like

```
out.push.push = 2;
```

to indicate that `out` should be grown by one element, and this new element (an empty array of integers) should also be grown by one element, the integer 2.

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

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

will produce the array `[[0, 1, 2], [3, 4], []]`.

Finally, for those cases where arrays are of a fixed, small size, a special notation describes array literals, which construct arrays of a fixed size by separating several expressions with commas at the right-hand side of an assignment. For instance, given that the type of `out` is `array of array [3] of string`, it is possible to write

```
out.push = "x", "y", "z";
```

which will append an array of 3 strings `["x", "y", "z"]` to `out`, instead of the more verbose

```
out.push[0] = "x";
out.back[1] = "y";
out.back[2] = "z";
```

4 String interpolation

To create new nodes, literals or different elements in the output, it is often needed to concatenate strings and values to produce new names. For instance, the name `"new_1_2"` could be produced by the following code (saved into variable `newname`)

```
i = 1;
j = 1;
newname = "new_" + i + "_" + (j+1);
```

However, generating new names this way can quickly lead to code that is difficult to read and understand. To aid in these situations, REDNP provides a simple form of string interpolation, in which variables or general expressions can be substituted into a string simply by surrounding them with two curly braces `{` and `}`. The last example could be written more concisely in the following way

```
newname = "new_{i}_{j+1}";
```

To avoid confusion, the braces are preserved in the substitution, that is, supposing that `x` is 1, and `y` is 2, the expression `"a{x}{y}"` evaluates to the string `"a{1}{2}"` and not to `"a12"`. The use of braces within strings is tied to string interpolation and therefore string literals such as the following will produce an error: `"aux{"`, `"xyz{}"`, or `"tmp}"`.

5 Functions and insertsat

The following functions are defined for integers in REDNP:

- `min(a,b)`, minimum of two integers `a` and `b`.
- `max(a,b)`, maximum of two integers `a` and `b`.
- `abs(x)`, absolute value of the integer `x`.

In addition, a procedure `insertsat` is provided to deal with the construction of clauses in those problems whose output produces a formula for SAT. The procedure `insertsat` has two parameters: the destination array and a string (if the destination array is `out`, it can be omitted). The destination array must be of type `array of array of string` and it is assumed to contain a list of clauses with literals specified by strings (i.e., a CNF formula). The second parameter to `insertsat` is a string with a logic expression. This string is compiled into a list of clauses, which are appended to the array specified as the first parameter. These logic expressions are written in a mini-language that makes it easy to describe logic formulas which are often used in reductions to SAT. Some examples will clarify its use:

```
"a => -b and -c"
"a implies not b & not c"
"s11 <= (s01 & -c1) | (s00 & c1)"
```

More precisely, the `insertsat` mini-language accepts the following operators:

- conjunction: `and`, `&`
- disjunction: `or`, `|`
- negation: `not`, `-`
- implication: `implies`, `=>`
- reversed implication: `if`, `<=`
- biconditional: `iff`, `<=>`

And it also allows cardinality constraints with strings of the form:

- `"atleast(k f1 f2 ... fN)"`: satisfied if and only if at least k of the subformulas `f1`, `f2`, ..., `fN` are true.

- **`"atmost(k f1 f2 ... fN)"`**: satisfied if and only if no more than k of the subformulas `f1`, `f2`, ..., `fN` are true.
- **`"exactly(k f1 f2 ... fN)"`**: satisfied if and only if exactly k of the subformulas `f1`, `f2`, ..., `fN` are true.

The parentheses are optional: if they are not present, the cardinality constraint encompasses all the remaining subformulas in the string.