# Appendix B

# GiL

## B.1 A simple example: SEND+MORE=MONEY

This example is a basic CSP generally used to introduce constraint programming. The real-world problem to solve is to assign a value (between 0 and 9) to the letters S, E, N, D, M, O, R and Y such that the following addition holds:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

Each letter must have a unique value; S and M must be different than 0 (as they are the first digit of a number).

### B.1.1 GiL program

```
(let ((sp (new-space))
      l c x dfs)
    (setq l (add-int-var-array sp 0 9))
    ;l[0] = s l[1] = e l[2] = n l[3] = d
    ;l[4] = m l[5] = o l[6] = r l[7] = y

    ; no leading 0
    (g-rel sp (nth 0 l) IRT_NQ 0)
    (g-rel sp (nth 4 l) IRT_NQ 0)
```

```
    ; all letters distinct
    (g-distinct sp l)
    ; linear equation
    (setq c '(1000 100 10 1 1000 100 10 1 -10000 -1000 -100 -10 -1))
    (setq x (list (nth 0 l) (nth 1 l) (nth 2 l) (nth 3 l)
                  (nth 4 l) (nth 5 l) (nth 6 l) (nth 1 l)
                  (nth 4 l) (nth 5 l) (nth 2 l) (nth 1 l) (nth 7 l)))
    (g-linear sp c x IRT_EQ 0)
    ; post branching
    (g-branch sp l 0 0)

    (setq dfs (search-engine sp nil))
    (do ((sol 0 (search-next dfs)))
        ((null sol) nil)
        (loop for v in (g-values sol l) (write i))
        (terpri) ;new line
    )
)
```

### B.1.2 Gecode program

```
class SendMoreMoney : public Space {
protected:
  IntVarArray l;
public:
  SendMoreMoney(void) : l(*this, 8, 0, 9) {
    //l[0] = s l[1] = e l[2] = n l[3] = d
    //l[4] = m l[5] = o l[6] = r l[7] = y

    // no leading zeros
    rel(*this, l[0], IRT_NQ, 0);
    rel(*this, l[4], IRT_NQ, 0);
    // all letters distinct
    distinct(*this, l);
    // linear equation
    IntArgs c(4+4+5); IntVarArgs x(4+4+5);
    c[0]=1000;    c[1]=100;    c[2]=10;    c[3]=1;
    x[0]=l[0];    x[1]=l[1];   x[2]=l[2];  x[3]=l[3];
    c[4]=1000;    c[5]=100;    c[6]=10;    c[7]=1;
    x[4]=l[4];    x[5]=l[5];   x[6]=l[6];  x[7]=l[1];
```

```
    c[8]=-10000; c[9]=-1000; c[10]=-100; c[11]=-10;  c[12]=-1;
    x[8]=l[4];   x[9]=l[5];   x[10]=l[2]; x[11]=l[1]; x[12]=l[7];
    linear(*this, c, x, IRT_EQ, 0);
    // post branching
    branch(*this, l, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
  }
  // search support
  SendMoreMoney(SendMoreMoney& s) : Space(s) {
    l.update(*this, s.l);
  }
  virtual Space* copy(void) {
    return new SendMoreMoney(*this);
  }
  // print solution
  void print(void) const {
    std::cout << l << std::endl;
  }
};

// main function
int main(int argc, char* argv[]) {
  // create model and search engine
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  // search and print all solutions
  while (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

## B.2 Improvements

### B.2.1 Tutorial: adding a constraint

This tutorial show an example of constraint wrapping from gecode to GiL, using the $abs(x, y)$ that expresses the $y = |x|$. Adding a use case of a constraint is a four-step process. The first step is to add a method that post this constraint in the class *WSpace*:

```
//space_wrapper.hpp
class WSpace : public IntMinimizeSpace
        ...
        void abs(int x, int y);
        ...

//space_wrapper.cpp
//Call the Gecode function abs() on this space and the integer
//variables at indices <x> and <y> of the IntVar vector.
void WSpace::abs(int x, int y) {
        Gecode::abs(*this, get_int_var(x), get_int_vars(y));
}
```

Then, a function must be created in the external C library (i.e. the Gecode Wrapper) with a void pointer parameter that will be cast to a *WSpace* pointer, that calls the *abs* method:

```
//gecode_wrapper.hpp
extern "C"
        ...
        void abs(void* sp, int x, int y);
        ...

//gecode_wrapper.cpp
//Call the above-defined abs method of the WSpace referenced
//by <sp>.
void abs(void* sp, int x, int y) {
        return static_cast<WSpace*>(sp)->abs(x, y);
}
```

The third step is to to create a CFFI function in the lisp part of the interface to call the C function. The CFFI only specifies the name of the C function called, the name of the new Lisp function, the return type, the documentation and the type of the arguments:

```
;ll-gil.lisp
(cffi::defcfun ("abs" abs) :void
```

```
    "Post the constraint that |vid1| = vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)
```

Finally, the last step is to create a Lisp method that uses the *int-var* class (in this case), and get the *vid* field of the varibales (i.e. their index in the integer variables vector of the *WSpace*) to call the foreign function via CFFI:

```
;ui-gil.lisp
(defmethod g-abs (sp (v1 int-var) (v2 int-var))
    (abs sp (vid v1) (vid v2))
)
```

The next steps are to add other use cases, for example wher $x$ is a fixed integer and $y$ is an integer variable. The user should pay attention to the names of the function: some C functions names are not allowed in the CFFI foreign functions definition; For example, it is highly probable that the *abs* name is not authorized, and a call to the lisp function *abs* will provoke a memory error.

The user should also remember that any modification of the C++ files requires a new compilation in order the changes to take effect.

### B.2.2 Branching strategies

In Gecode, branching strategies are set as in the following example:

```
//Post branching on the integer variables in x, beginning by the
//firt variable with the smallest domain (INT_VAR_SIZE_MIN) and
//assigning its trying its smallest value first (INT_VAL_MIN)
branch(*this, x, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
```

The third and fourth arguments are functions that return instances of strategy classes that extend the classes *VarBranch* and *ValBranch* respectively Some of the strategy functions require arguments to work properly. To wrap the strategies, the branching methods of the *WSpace* should convert their strategy selectors arguments to call to those functions. The challenge is to find a way to represent all the different types of arguments these functions can have in a way that can be carried from C to Lisp through CFFI.

### B.2.3 Expressions

In order to include support for expressions in GiL, the wrapper should implement its own "minimodel" (see chapter 7: *Modeling convenience: MiniModel* in [Schulte et al., 2019]) that would allow to perform operations on variables. In practice, it would decompose a complex operation into atomic operation corresponding to temporary variables and post the constraint afterward.