Chapter 8

GiL

The first step towards creating a constraint-based tool in OpenMusic was to bring the constraint programming aspect to Lisp. This has been done by Baptiste Lapière [11] with the creation of GiL[1], an interface between Gecode[2] and Lisp using CFFI[3]. GiL will therefore be used to handle the constraints part of Melodizer, and some functionalities will be added, which are described later.
This chapter does not aim to explain how GiL works in full detail, the curious reader can read [11] for more information on that. The first section explains the basic principles of GiL, the next section details the features that were added to make Melodizer work. This is followed by an example of how to use GiL to solve CSPs in Lisp. The last section will explain briefly how to contribute to GiL. The new version of GiL can be found at `https://github.com/sprockeelsd/GiLv2.0`.

8.1 Structure

GiL is composed of two main parts : a C wrapper and a Lisp wrapper. Since CFFI provides an interface between C and Lisp, but not C++, the first step is to create a C library that executes Gecode functions, that can then be called with Lisp code in OpenMusic.

8.1.1 C Wrapper

The C Wrapper is made of two main parts: the first part, called the space wrapper, contains all the calls to Gecode functions. It includes the definition of the WSpace class, including all its methods to create variables, post constraints and specify branching strategies, as well as the search engines classes. The second part, called the gecode wrapper, is the external C library calling the methods from the space wrapper. The functions from the gecode wrapper will then be called from the Lisp Wrapper, which will be detailed later.

In Gecode, variables, constraints and branching are modeled within a subclass of the “Space” class, which represents the problem. In GiL, this translates to a “WSpace” class with methods to add variables, post constraints, and specify a branching strategy. Upon creation, the WSpace instance is casted to a void pointer to be useable in CFFI. Everytime a function from the C external library that uses a method of the space wrapper is called, the pointer is casted back to a WSpace pointer to call the method. This is illustrated in figure 8.1.

Variables Variables in Gecode are instances of a class, so another way to reference them is necessary to be able to refer to them in C. The WSpace class has a vector as attribute for each type of variables (integer, boolean) and protected methods to retrieve a variable or variable array based on their indexes. Upon creation, a variable is pushed to the corresponding vector and its index in the vector is returned so it can be accessed later.

[1]https://github.com/blapiere/GiL
[2]https://www.gecode.org
[3]The Common Foreign Function Interface, see https://common-lisp.net/project/cffi/

```
// Wraps the WSpace constructor.
void* computation_space() {
    return (void*) new WSpace();
}

//Wraps the WSpace add_intVar method.
int add_intVar(void* sp, int min, int max) {
    return static_cast<WSpace*>(sp)->add_intVar(min, max);
}
```

Figure 8.1: Code from gecode wrapper, for creating a new space and adding an integer variable to it.

variable selection strategies	value selection strategies
variable with smallest domain	smallest value of the domain
random variable	random value
variable with the highest degree	values not greater than (min + max)/2
	values greater than (min + max)/2
	biggest value not greater than the median

Figure 8.2: Currently supported branching strategies

Constraints Constraints are wrapped in a similar way. For each constraint, one or more methods are created in the space wrapper and they take the indexes of variables as arguments. The corresponding WSpace method then translates indexes into variables to post the constraints.

Branching Branching is also wrapped in a similar manner. Before this master's thesis, there was no way to specify the branching strategy in GiL, and the default strategies were choosing the variable with the smallest domain, and choosing the smallest value of the domain first.

Search Engines Search engines are wrapped separately from the space. They are modeled as a class (e.g. WdfsEngine) with an attribute that is a reference to the corresponding Gecode search engine. Before this master's thesis they provided a method to call their next function, that returns a space that is a solution of the CSP if there is any.

8.1.2 Lisp Wrapper

Now that there is an external C library, it can be wrapped in Lisp with the help of CFFI. The Lisp wrapping is done in two stages in GiL. First, foreign functions are defined using CFFI, each calling a specific function from the Gecode Wrapper. Those functions are the link between Lisp and C, calling C functions and giving the returned value to Lisp. Then, the foreign functions are wrapped by Lisp functions to provide a more readable library, making using GiL as similar to using Gecode as possible.

8.2 Contribution

Since the goal of this master's thesis was not to develop GiL but rather to produce a useful tool for composers, only necessary features were added to GiL. However, these additions were designed in a way that allows to easily develop them or add other features without creating conflicts.

Branching Branching on integer variables was improved by adding the possibility to specify branching strategies, both for variable and value selection. This was done by passing integers as arguments, representing the branching strategies, using pre-defined parameters to make it more user friendly. These integers are then translated to actual Gecode strategies in the space wrapper. Figure 8.2 shows the currently supported strategies.

Search engines The search engine constructors have been modified to support search options (more explanations below). Additionally, they now have a method allowing to detect if the search engine has been stopped. The GiL Lisp function to create search engines has been modified to make it more modular and intuitive, by allowing to specify the type of search engine to create instead of setting an additional parameter to “t” or “nil” as it was before, allowing to support more than 2 types of search engines.

Search options In Gecode, it is possible to specify options for the search of solutions. It is done by using Options objects. In GiL, they are wrapped in a similar way to search engines, in a WSearchOptions class with one attribute that is an Options object from Gecode. The class has methods allowing to set the different options. Currently, only the “threads” and “stop”[4] options are supported, but others can easily be added in a similar way.

Stop Option The “stop” option in Gecode works by creating a stop object and specifying a limit depending on its type. GiL currently only supports TimeStop objects, and therefore only allows to specify a time limit and not for example a limit in terms of number of nodes visited. TimeStop objects are wrapped the same way search engines and options objects are, and they have a method that allows to reset the timer.

Value precedence constraint The Gecode “precede” constraint has been added to GiL. Given a variable array x and two integers s and t, the constraint ensures that there will always be a “s” before a “t”. In other words,

$$x_0 \neq t$$

$$x_j = t \Rightarrow \exists i \in [0, j[\mid x_i = s$$

Count constraint One variation of the “count” constraint has been added to GiL. Given an array of variables x, a set of integers c, a relation rel_type and an integer z, the constraint ensures that the number of variables in x that take a value in c has relation rel_type to z. In other words,

$$\#\{i \in [0, ...|x| - 1] \mid x_i \in c\} \; rel\_type \; z$$

8.3 Example of use

This section will show how the different steps of creating a simple CSP compare in Gecode and in GiL. The problem chosen here is a very simple problem with 3 variables with a domain ranging from 1 to 4, where the only constraint posted is that all variables must be different. The branching strategies are selecting the variable with the smallest domain first and selecting the smallest value of the domain first. A depth first search engine is used. Figure 8.3a show the code for this simple problem in Gecode, and figure 8.3b shows the exact same problem in Lisp using GiL.

8.4 How to add to GiL

Adding a feature to GiL can be done in four steps. First, it must be wrapped in the space wrapper. If you wish to add a constraint, a simple function calling the Gecode function is enough. If you want to add something more complex like a search engine type, you will probably need to wrap it into a class. Then, one or more functions must be created in the gecode wrapper (external C library) to call the functions created in the space wrapper. Note that pointer arguments for functions in the gecode wrapper must be of type void*, and casted into the appropriate type when calling space wrapper functions. The third step is to create Lisp functions that call the C functions using CFFI. Finally, the last step is to add more user-friendly functions to call the foreign functions created with CFFI, using for example the “int-var” class.

[4]for a comprehensive list of options, see [14], Chapter 9.3

```
class Simple_problem : public Space {
  protected:
    IntVarArray vars;
  public:
    Simple_problem(): vars(*this,3, 1, 4){ // initialize the variables
      // post the constraints
      distinct(*this, vars);
      //specify the branching
      branch(*this, vars, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
    }
    ... //code that is not relevant to show how Gecode and GiL compare
};

int main(int argc, char* argv[]) {
  Simple_problem* m = new Simple_problem(); //create the space
  Gecode::Search::Options opts;// create and initialize the search options
  Gecode::Search::TimeStop maxTime(500);//create and initialize the time stop object
  opts.stop = &maxTime; //specify the time limit in the search options
  DFS<Simple_problem> e(m);// create the search engine
  delete m;

  // search and print all solutions
  while (Simple_problem* s = e.next()) {
    s->print(); delete s; maxTime.reset();// reset timer
  }
  return 0;
}

```

(a) Simple CSP in Gecode

```
(defun dummy-problem ()
    (let ((sp (gil::new-space))     ; create the space
    vars se tstop sopts max id-list)
        (setq vars (gil::add-int-var-array sp 3 1 4)); initialize the variables
        ; post constraints
        (gil::g-distinct sp vars)
        ; specify branching
        (gil::g-branch sp vars gil::INT_VAR_SIZE_MIN gil::INT_VAL_MIN)

        ; search options
        (setq sopts (gil::search-opts)); create the search options object
        (gil::init-search-opts sopts); initialize it
        ;time stop
        (setq tstop (gil::t-stop)); create the time stop object
        (gil::time-stop-init tstop 500); initialize it (in ms)
        (gil::set-time-stop sopts tstop); specify the time limit in the search options

        ; create the search engine
        (setq se (gil::search-engine sp (gil::opts sopts) gil::DFS))
        ; return for the ‘‘next" function
        (list se vars tstop sopts)))

(defun search-next (l)
    (let ((se (first l)) (pitch* (second l)) (tstop (third l)) (sopts (fourth l)) sol
    pitches)
        (gil::time-stop-reset tstop);reset the timer before launching the search
        (setq sol (gil::search-next se)); search for the next solution
        (if (null sol) (error ‘‘No more solutions"))
        (setq pitches (gil::g-values sol pitch*)); store the values of the solution
        (print pitches)))

```

(b) Simple CSP in lisp using GiL

Figure 8.3: Simple CSP in Gecode and in GiL