GNIRUT: a tutorial

N. Ollinger

version 1.0 – April 8th, 2008

Abstract

GNIRUT is a programming toolkit built around a language for describing Turing machines, with special features oriented towards the construction of reversible Turing machines and the use of so-called Hooper's style recursive calls.

1 Components of the Gnirut toolkit

After proper installation, the GNIRUT programming toolkit consists of the following programs plus several contributed third-party tools not described here:

`gnirut` is an interactive interpreter of the GNI language. It is the main program that will be used through this tutorial.

`gnirutc` is a compiler which transforms GNI source code into RUT Turing machine binary format. It is precisely equivalent to launch the interpreter, issue a `@use` followed by a `@save` command and quitting.

`gnipp` is a source beauty-fuller/pretty-printer: given a GNI source file as input, it produces a LaTeX source for proper inclusion into LaTeX documents. The examples in this tutorial are typeset using `gnipp`.

`rutdump` is a program for printing in a human-readable form the content of a RUT Turing machine binary format. It is precisely equivalent to launch the interpreter, issue a `@load` followed by a `@dump` command and quitting.

`rutrev` is a program for reverting a (reversible) RUT Turing machine binary format and save it into a new RUT file. It is precisely equivalent to launch the interpreter, issue a `@load` followed by a `@revert`, followed by a `@save` command and quitting.

`rutrun` is a program for running a RUT Turing machine binary format on a specific input starting from a given state. It is precisely equivalent to launch the interpreter, issue a `@load` followed by a `@run` command and quitting.

2 Programming Turing machines

In this part, we will get familiar with the basic usage of the GNI language: how to define a machine, use macros to avoid writing the same things again and again, load and save RUT files, store programs in files and include them.

➥ Launch the `gnirut` interpreter in a terminal. After announcing itself, it displays a `>>>` prompt, waiting for your commands. To quit the interpreter at any time, just hit `Ctrl+C` ou `Ctrl+D`.

2.1 Formal Rut machines

Gni programs specify Rut machines. Let us first describe Rut machines in an abstract formal way.

A Rut Turing machine is triple (S, Σ, I) where S is a finite set of states, Σ is a finite set of symbols, and I is a finite set of instructions, a subset of $S \times \{\leftarrow, \rightarrow\} \times S$ (move instructions) union $S \times \Sigma \times \Sigma \times S$ (matching instructions). All machines considered are supposed to be deterministic: if a state s appears in a move instruction $(s, \ldots)$, then it does not appear in any other instruction; if a state s appears in a matching instruction $(s, a, \ldots)$, then the other instructions it appears in are only matching instructions $(s, b, \ldots)$ where $a \neq b$.

A configuration of a Rut machine (S, Σ, I) is a triple (s, k, t) where $s \in S$ is the current state of the machine, $k \in \mathbb{Z}$ is the position of the head of the machine and $t \in \Sigma^{\mathbb{Z}}$ is a coloring of a bi-infinite tape by symbols.

Starting from a configuration (s, k, t), in one step of computation, the machine (S, Σ, I) looks up at its instruction set I for an instruction associated to state s:

- if there exists a move instruction $(s, \leftarrow, s')$, the machine enters configuration $(s', k-1, t)$;
- if there exists a move instruction $(s, \rightarrow, s')$, the machine enters configuration $(s', k+1, t)$;
- if there exists a matching instruction (s, a, b, s') and if $t(k) = a$, the machine enters configuration (s', k, t') where t' is equal to t everywhere but in k where $t'(k) = b$;
- if no such instruction exists, the machine halts.

A Rut machine is complete if it cannot halt. A Rut machine is reversible if there exists a second Rut machine such that for any configuration c, if the first machine transforms the c into c' in one step, then the second machine transforms c' into c in one step.

2.2 Basics statements

A Gni program is a sequence of statements. Each statement is written on a separate line. As we will see when discussing macros, the indentation matters in a Gni program. For this subsection, just be sure not to put spaces or tabulations in front of you statements. Comments can be added to statements: starting by a `#` symbol, the comment run until the end of the line. Comments are just ignored by the interpreter.

In a Gni program, states and symbols are represented by identifiers: non-empty sequences of letters (lowercase or uppercase, case-sensitive), digits, underscore, dot and prime (more precisely, any sequence matching the regular expression `[.a-zA-Z0-9_']*[a-zA-Z0-9_']`). The set of states and symbols of the machine defined by a program is the set of states and symbols appearing in its statements.

The syntax of basic statements is the following:

`s. <-, s'` move instruction $(s, \leftarrow, s')$;

`s. ->, s'` move instruction $(s, \rightarrow, s')$;

`s. a:b, s'` matching instruction (s, a, b, s').

When not describing precise syntax, we will give programs in a stylized form like in the following example. Notice that the line numbers are just given to help reading and should not be typed.

1 *begin.* `x:x`, *search*
2 *search.* →, *loop*
3 *loop.* `_:_`, *search*
4 *loop.* `x:x`, *end*

➥ Enter these statements, one after the other, into your interpreter.

To see the current defined machine inside the interpreter, type the special command `@dump`.

➥ Type `@dump` and see what your interpreter currently knows. Notice that the matching instructions appear in some compound form.

To test the machine behavior, it is possible to run it on partial configurations, using the `@show` command. The syntax is the following:

`@show s +n "abababababab"` run the machine starting from the configuration obtained by the tape content `abababababab` (only 1 letter identifiers allowed), in state `s` at position `n` from the left side of the tape, stops on halt or when the head exits the defined tape region;

`@show +mx s +n "abababababab"` same as above but execute during at most `mx` time steps. Convenient to avoid infinite loops during debugging.

➥ Test your machine with `@show begin +1 "_x___x"`

Exercise 1. Program in the interpreter a small machine that starting in state s on the first letter of a word $x\_^n x\_$, modifies the word into $x\_^{n+1} x$ and halts in state t at the same position. Run it!

2.3 Compound statements

To simplify the writing of matching statements, several matching with the same initial state can be merged into a compound statement. With the same philosophy, default behavior can be defined to facilitate the writing of compound matchings of the kind *replace a by b and enter state t but if the letter is not a neither b or c then replace it by x and enter state u.* The syntax of compound statements is the following:

`s. a:b, t | b:c, u | d:e, v` sequence of matching instructions (s, a, b, t), (s, b, c, u), (s, d, e, v);

`s. a:b, s' else t` same as before but with a default behavior: on any letter not defined in the compound matching instructions, do not modify the letter but jump to state t: it is syntactically equivalent to replace `else t` by $|w : w, t$ for each letter w of the program (already defined or appearing in latter statements) not already appearing in the statement;

`s. a:b, s' else t but b,c` same as an `else` modifier but the modifier does not apply for letters appearing after the `but` keyword;

`s. a:b, s' else t write a` same as an `else` modifier but instead of not modifying the letter, any letter is replace by a: syntactically like $|w : a, t$;

`s. a:b, s' else t write a but b,c` is a combination of all the modifiers.

➥ Use the `@clear` command to remove all instructions from the interpreter and then reenter your answer to exercise 1 using compound statements.

Exercise 2. Program in the interpreter a small machine that starting in state s on the first letter of a word $x\_^n x\_$, halts at the same position, in state z if $n = 0$ or in state p if $n > 0$. Run it!

2.4 Dealing with files

The interpreter is a nice tool to design parts of machines but there is no way to save the statements. Thus, a better approach to design big machines is to write the program in some source file (`.gni` extension is good practice) and to use the `@use` command to load it into the interpreter.

The produced Rut machines can also be saved in order to be reloaded into the interpreter or used with third-party tools in a format independent of the Gni language.

The syntax of the file commands is the following:

`@use "toto.gni"` open the file `toto.gni` and execute its statements;

`@load "toto.rut"` load the `toto.rut` Rut machine;

`@save "toto.rut"` save the current machine into the file `toto.rut`.

➥ Write the source code of exercise 2 into a file. Clear the interpreter, then use your source file. Save the compiled machine to a file. Quit the interpreter. Relaunch it. Load your saved machine. Run it!

Exercise 3. Now that you can modify source with a text editor and save the machine, building big machines is easier. Design and test a machine that starting in state s on the first letter of an input word $u \in \{a, b\}$, written on a blank tape filled with $\_$, halts on the last letter of the word, in state y if the word is a palindrome or in state n otherwise. Feel free to add more symbols to fit your needs but be sure that the word u is written on the tape at the end of the computation.

2.5 Using macros

As such, the language is clearly sufficient to encode any machine but it can be boring to repeat over and over the same patterns, copies of the same sub-machine performing a given task like *go to the first x to the right*. To help on this, the language provides a macro definition system.

Defining a macro is really the same as defining the main machine. The same instructions are used. In both cases what you obtain is a Rut machine. But for macro definitions all states loose their names after definition but the ones you select as relevant to the outside world. Moreover, the machine associated to the macro is not added to the current main machine but stored for future use. Once the macro defined, to use it, you ask the interpreter to add a fresh copy of the machine states in the current environment.

Indentation plays a big role in macro definition. After the `def` starting the macro definition, all the lines corresponding to that macro should BE indented by the same amount of spaces and/or tabulations, which should be strictly more than the indentation of the `def` line. When the indentation goes back to the original level, the macro definition ends.

The syntax for macro definition and usage is the following:

`def [i1,...,im|toto|o1,...,on>:` begins the definition of a macro called `toto`; the only states that will be visible from outside are $i_1, \ldots, i_m$ and $o_1, \ldots, o_n$. There is no technical distinction between i and o but it is good practice to consider i as input states and o as output states, to facilitate reading of source code.

`[a,b|titi|c,d,e>` inserts a copy of the macro machine `titi` in the current machine definition: the fresh copy of the machine `titi` will use a, b as input states for its i_1, i_2 and c, d, e as output states for its o_1, o_2, o_3.

The following code defines a macro and uses it:

```
1  def [s|search|t⟩ :
2      s. x:x, u
3      u. →, r
4      r. _:_, u | x:x, t
5
6   [s|search|a⟩
7   [a|search|b⟩
8   b. →, c
9   c. a:b, d | b:a, d
```

➥ Observe the behavior of the given example starting from state s on the first letter of a word of the kind $x\_^n x\_^m xa$.

Exercise 4. Reusing your answers to previous exercises as a basis to define macros, construct three macros $[s|\text{test}|z, p\rangle$, $[s|\text{inc}|t\rangle$ and $[s|\text{dec}|t\rangle$, the three of them aimed to be started from a partial configuration of the kind $x\_^n x\_$ and respectively: testing whether $n = 0$ or $n > 0$, incrementing n, decrementing n. Explain how to use these macros to simulate a one counter machine. Take a sample machine of that kind, encode it using the macros. Run it!

3 Advanced topics

3.1 Applying transforms

Sometimes, one would like to reuse a machine with slight syntactical transformations. Currently, two such transforms are defined in Gni. The syntax and usage is the following:

`[a,b|lr titi|c,d,e>` inserts a copy of the macro machine `titi` where the move instruction have inverted directions: ← is replaced with → and reciprocally.

`<o1,...,on|toto|i1,...,im]` inserts a copy of the inverse of the machine `toto`, provided that `toto` is reversible.

➥ insert a `lr` copy of the previous `search` macro in the interpreter and dump it. Then, clear the interpreter (this does not remove the defined macros, just the main machine) and insert an inverse copy of `search`. Dump it, verify that both dumped machines are indeed the same! Restart with a search that is not symmetrical (for example, replace the right x by y).

Exercise 5. Rewrite the macros of exercise 4 so that the three macros are reversible, taking advantage of the transforms (for example using both `[|search|>` and `<|search|]`). To check that the macros are reversible, just try to insert their inverse in a cleared main machine.

3.2 Hooper's style recursive calls

Macros are fine but co-recursive machines would lead to infinite state machine, which is forbidden. One way to avoid this is to use the tape of the Turing machine as a place to push entry point information before calling a machine and popping from that same place to know where to go back. Gni provides that kind of feature, with the following assumptions. It is the programmers responsibility to ensure that the called machine will never modify the tape cell from which it starts and will come back to that particular position at the end of the computation. The syntax and usage is the following:

`fun [i1,...,im|toto|o1,...,on>`: defines a callable function: the syntax and principle are the same as for defining macros but the obtained machine is not usable as a macro, it is used through calls and just one copy of it will be put into the main machine if it is called from at least one place.

`call [i1,...,im|toto|o1,...,on> from a` insert a function call: when entering states i_1 to i_m, the machine will push entry point information on the tape, replacing the letter a that is written on it; on return from the call it will pop the entry point information, replace it by a and change state to o_1,...,o_n depending on the return state. The function does not need to exist at the time the call is inserted, so co-recursive calls are allowed.

`call [i1,...,im|lr toto|o1,...,on> from a` calls the `lr` version of `toto`.

`call <o1,...,on|toto|i1,...,im] from a` calls the inverse version of `toto`.

`call <o1,...,on|lr toto|i1,...,im] from a` calls the inverse of the `lr` version of `toto`.

`@link` issuing a `call` instruction in the interpreter does not effectively modify the machine but stores calling informations. The `@link` command is taking care of wiring all calls, putting copies of used functions and `lr` and/or inverse of them as appropriate, recursively inserting all needed functions that might be called by inserted functions.

`@link [a,b|toto|c,d,e>` is a special form of linking that ensure that at least the function `toto` will be linked, with input and output states named a, b and c, d, e.

The following code defines a function and uses it to recursively compute a recursive function on positive integers represented as blocks of x:

```
1  def [s|test|z, p⟩ :
2      s. →, r
3      r. _:_, za | x:x, pa
4      za. ←, z
5      pa. ←, p
6
7  def [s|dec|t⟩ :
8      s. →, a
9      a. _:_, b | x:x, s
10     b. ←, c
11     c. x:_, r
12     r. ←, u
13     u. _:_, t | x:x, r
14
15 def [s|copy|t⟩ :
16     s. →, a
17     a. _:_, b | x:o, s
18     b. ←, c
19     c. o:o, b else d
20     d. →, e
21     e. _:_, t | o:x, f
22     f. →, g
23     g. _:_, h | o:o, f
24     h. →, i
25     i. _:x, j | x:x, h
26     j. ←, k
27     k. _:_, b | x:x, j
28
29 fun [s|f|t⟩ :
30     [s|test|t, p⟩
31     [p|copy|a⟩
32     [a|dec|b⟩
33     call [b|f|c⟩ from _
34     c. _:x, l
35     l. ←, o
36     o. x:x, l else t
37
38 @link [s|f|t⟩
```

➥ Type this example, understand it, run it from state s starting from the left of words of the kind $\_x^n\_^\omega$ for small values of n.

Exercise 6. Construct a RUT machine that starting on state s from the left of words of the kind $\_x^n\_^\omega$, halt on state t on the left of a word of the kind $\_x^m\_^\omega$ such that $m = f(n)$ where f is the fibonacci function defined by $f(0) = f(1) = 1$ and $f(n+2) = f(n) + f(n+1)$.

Exercise 7. Modify your answer from exercise 6 to construct a reversible machine computing the same function on valid images. More generally, explain how any injective function computed by a machine starting from an ultimately constant tape can be computed by a reversible machine with the same input and output tape (*i.e.* no garbage).