GNU Jitter

a low-level introduction

Luca Saiu
https://ageinghacker.net
positron@gnu.org
GNU Project

GNU's 40th anniversary celebration
Biel, Switzerland
September 27th 2023

About this talk

I have a difficult task today. I want to:

- awe you and get you curious
- without enough time to show you details
- I will link references containing more information (For example: [6])

About this talk

I have a difficult task today. I want to:

- awe you and get you curious
- without enough time to show you details
- I will link references containing more information (For example: [6])

About this talk

I have a difficult task today. I want to:

- awe you and get you curious
- without enough time to show you details
- I will link references containing more information (For example: [6])

Motivation

Interpreters are important:

- programming languages;
- shells;
- regular expressions;
- spreadsheets;
- application scripting and extensions...
- easy, fun to write and play with
- slow

We need compilation

- difficult
- not portable

Motivation

Interpreters are important:

- programming languages;
- shells;
- regular expressions;
- spreadsheets;
- application scripting and extensions...
- easy, fun to write and play with
- slow

We need compilation

- difficult
- not portable

Motivation

Interpreters are important:

- programming languages;
- shells;
- regular expressions;
- spreadsheets;
- application scripting and extensions...
- easy, fun to write and play with
- slow

We need compilation

- difficult
- not portable

Motivation

Interpreters are important:

- programming languages;
- shells;
- regular expressions;
- spreadsheets;
- application scripting and extensions...
- easy, fun to write and play with
- slow

We need compilation

- difficult
- not portable

Motivation

Interpreters are important:

- programming languages;
- shells;
- regular expressions;
- spreadsheets;
- application scripting and extensions...
- easy, fun to write and play with
- slow

We need compilation

- difficult
- not portable

Motivation, in a more personal sense

I like programming languages

- Formal languages are the best way of interacting with machines [5]
- The "software crisis" is not solved at all
- No language is good enough
  - "Increasing those designed by me" (Donald J. Trump (paraphrased))
- GNU epsilon is my attempt
  - a long term project, research, creative outlet

Motivation, in a more personal sense

I like programming languages

- Formal languages are the best way of interacting with machines [5]
- The "software crisis" is not solved at all
- No language is good enough
  - "Increasing those designed by me" [Donald J. Trump (paraphrased)]
- GNU epsilon is my attempt
  - a long term project, to maintain, develop, refine

Motivation, in a more personal sense

I like programming languages

- Formal languages are the best way of interacting with machines [5]
- The "software crisis" is not solved at all
- No language is good enough
  - . . . *"including those designed by me"*
    —Gerald J. Sussman (paraphrased)
- GNU epsilon is my attempt
  - a long term project, a research area, a hobby

Motivation, in a more personal sense

I like programming languages

- Formal languages are the best way of interacting with machines [5]
- The "software crisis" is not solved at all
- No language is good enough
  - ... *"including those designed by me"* —Gerald J. Sussman (paraphrased)
- GNU epsilon is my attempt

Motivation, in a more personal sense

I like programming languages

- Formal languages are the best way of interacting with machines [5]
- The "software crisis" is not solved at all
- No language is good enough
  - . . . *"including those designed by me"* —Gerald J. Sussman (paraphrased)
- GNU epsilon is my attempt
  - a long-term project, rewritten several times

History

GNU epsilon is meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped — Too slow

In 2016 I wrote a canonical threaded-code *language virtual machine*.

- speedup 4-6x — Too little

So I made a separate repository to experiment with language Virtual Machines.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people [GForth is a very nice GNU package])
- added ideas of my own
- it got completely out of hand

A new project, independent from epsilon.

- Jitter [1], since 2021 GNU Jitter

History

GNU epsilon is meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped — Too slow

In 2016 I wrote a canonical threaded-code *language virtual machine*.

- speedup 4-6x — Too little

So I made a separate repository to experiment with language Virtual Machines.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people [GForth is a very nice GNU package])
- added ideas of my own
- it got completely out of hand

A new project, independent from epsilon.

- Jitter [1], since 2021 GNU Jitter

History

GNU epsilon is meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped — Too slow

In 2016 I wrote a canonical threaded-code *language virtual machine*.

- speedup 4-6x — Too little

So I made a separate repository to experiment with language Virtual Machines.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people [GForth is a very nice GNU package])
- added ideas of my own
- it got completely out of hand

A new project, independent from epsilon.

- Jitter [1], since 2021 GNU Jitter

History

GNU epsilon is meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped — Too slow

In 2016 I wrote a canonical threaded-code *language virtual machine*.

- speedup 4-6x — Too little

So I made a separate repository to experiment with language Virtual Machines.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people [GForth is a very nice GNU package])
- added ideas of my own
- it got completely out of hand

A new project, independent from epsilon.

- Jitter [1], since 2021 GNU Jitter

History

GNU epsilon is meant to be efficient, but:

- very "dynamic" in certain execution phases
- written in itself, bootstrapped — Too slow

In 2016 I wrote a canonical threaded-code *language virtual machine*.

- speedup 4-6x — Too little

So I made a separate repository to experiment with language Virtual Machines.

- tried techniques from scientific papers (many by Anton Ertl and the other GForth people [GForth is a very nice GNU package])
- added ideas of my own
- it got completely out of hand

A new project, independent from epsilon.

- Jitter [1], since 2021 GNU Jitter

Language Virtual Machines

The way of out the interpreters vs compilers dilemma: Language Virtual Machine

Jitter is a *generator* of language virtual machine

- C code generator (like Bison) from a human-written specification

Describe programs as composition of (high-level) operations:

- VM "instructions" defined in C by a human
  - easy, flexible
- simple compiler
  - fun

Language Virtual Machines

The way of out the interpreters vs compilers dilemma: Language Virtual Machine

Jitter is a *generator* of language virtual machine

- C code generator (like Bison) from a human-written specification

Describe programs as composition of (high-level) operations:

- VM "instructions" defined in C by a human
  - easy, flexible
- simple compiler
  - fun

Language Virtual Machines

The way of out the interpreters vs compilers dilemma: Language Virtual Machine

Jitter is a *generator* of language virtual machine

- C code generator (like Bison) from a human-written specification

Describe programs as composition of (high-level) operations:

- VM "instructions" defined in C by a human
  - easy, flexible
- simple compiler
  - fun

Language Virtual Machines

The way of out the interpreters vs compilers dilemma: Language Virtual Machine

Jitter is a *generator* of language virtual machine

- C code generator (like Bison) from a human-written specification

Describe programs as composition of (high-level) operations:

- VM "instructions" defined in C by a human
  - easy, flexible
- simple compiler
  - fun

Language Virtual Machines

The way of out the interpreters vs compilers dilemma: Language Virtual Machine

Jitter is a *generator* of language virtual machine

- C code generator (like Bison) from a human-written specification

Describe programs as composition of (high-level) operations:

- VM "instructions" defined in C by a human
  - easy, flexible
- simple compiler
  - fun

Runtime data

VM code "feels" assembly-like, with abstract data:

- (an unlimited number of) registers
- stack(s)

An example program written in an extension language (register VM code)

Example from [6]

The program to run...

```
var a = 1333333333,
    b = 1;

while a <> b do
  if a < b then
    b := b - a;
  else
    a := a - b;
  end;
end;

print a;
```

...Translated into a register-VM routine

```
        mov     1333333333, %r0
        mov     1, %r1
        be      %r0, %r1, $L9
$L3:
        bge     %r0, %r1, $L6
        minus   %r1, %r0, %r1
        b       $L7
$L6:
        minus   %r0, %r1, %r0
$L7:
        bne     %r0, %r1, $L3
        b       $L9
$L9:
        print   %r0
        exitvm
```

An example program written in an extension language (register VM code)

Example from [6]

The program to run...

```
var a = 1333333333,
    b = 1;

while a <> b do
  if a < b then
    b := b - a;
  else
    a := a - b;
  end;
end;

print a;
```

...Translated into a register-VM routine

```
        mov     1333333333, %r0
        mov     1, %r1
        be      %r0, %r1, $L9
$L3:
        bge     %r0, %r1, $L6
        minus   %r1, %r0, %r1
        b       $L7
$L6:
        minus   %r0, %r1, %r0
$L7:
        bne     %r0, %r1, $L3
        b       $L9
$L9:
        print   %r0
        exitvm
```

An example program written in an extension language (stack VM code)

```
        mov             1333333333, %r0
        mov             1, %r1
        push-stack      %r0
        push-stack      %r1
        different-stack
        bf-stack        $L24
$L6:
        push-stack      %r0
        push-stack      %r1
        less-stack
        bf-stack        $L15
        push-stack      %r1
        push-stack      %r0
        minus-stack
        pop-stack       %r1
        b               $L19
```

```
$L15:
        push-stack      %r0
        push-stack      %r1
        minus-stack
        pop-stack       %r0
$L19:
        push-stack      %r0
        push-stack      %r1
        different-stack
        bt-stack        $L6
        b               $L24
$L24:
        push-stack      %r0
        print-stack
        exitvm
```

A VM instruction example

add VM instruction: Jitter specification, human-written

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGN0 + JITTER_ARGN1;
  end
end
```

Instantiate into every possible instantiation of register and immediate. One example:

add specialisation r4/n1/r4: Generated C, macroexpanded, simplified

```
add_r4_n1_r4_begin:
  _local_state.r4 = _local_state.r4 + 1;
add_r4_n1_r4_end:
```

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
  addq $1, %rdx   # Here %rdx is both input and output
add_r4_n1_r4_end:
```

A VM instruction example

add VM instruction: Jitter specification, human-written

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGN0 + JITTER_ARGN1;
  end
end
```

Instantiate into every possible instantiation of register and immediate. One example:

add specialisation r4/n1/r4: Generated C, macroexpanded, simplified

```
add_r4_n1_r4_begin:
  _local_state.r4 = _local_state.r4 + 1;
add_r4_n1_r4_end:
```

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
  addq $1, %rdx   # Here %rdx is both input and output
add_r4_n1_r4_end:
```

A VM instruction example

add VM instruction: Jitter specification, human-written

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGN0 + JITTER_ARGN1;
  end
end
```

Instantiate into every possible instantiation of register and immediate. One example:

add specialisation r4/n1/r4: Generated C, macroexpanded, simplified

```
add_r4_n1_r4_begin:
  _local_state.r4 = _local_state.r4 + 1;
add_r4_n1_r4_end:
```

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
  addq $1, %rdx   # Here %rdx is both input and output
add_r4_n1_r4_end:
```

A VM instruction example

add VM instruction: Jitter specification, human-written

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGN0 + JITTER_ARGN1;
  end
end
```

Instantiate into every possible instantiation of register and immediate. One example:

add specialisation r4/n1/r4: Generated C, macroexpanded, simplified

```
add_r4_n1_r4_begin:
  _local_state.r4 = _local_state.r4 + 1;
add_r4_n1_r4_end:
```

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
  addq $1, %rdx    # Here %rdx is both input and output
add_r4_n1_r4_end:
```

What Jitter does

(In its most efficient mode [1; 4]) Jitter:

- generates a huge C function containing all of the VM instructions specialisations

Then at runtime, the VM:

- copies compiled code for each VM instruction into executable memory, with `mmap`...
- ...the concatenation of the copies is a correct executable routine
- ...jump to the beginning of it

What Jitter does

(In its most efficient mode [1; 4]) Jitter:

- generates a huge C function containing all of the VM instructions specialisations

Then at runtime, the VM:

- copies compiled code for each VM instruction into executable memory, with mmap...
- ...the concatenation of the copies is a correct executable routine
- ...jump to the beginning of it

What Jitter does

(In its most efficient mode [1; 4]) Jitter:

- generates a huge C function containing all of the VM instructions specialisations

Then at runtime, the VM:

- copies compiled code for each VM instruction into executable memory, with `mmap`...
- ...the concatenation of the copies is a correct executable routine
- ...jump to the beginning of it

What Jitter does

(In its most efficient mode [1; 4]) Jitter:

- generates a huge C function containing all of the VM instructions specialisations

Then at runtime, the VM:

- copies compiled code for each VM instruction into executable memory, with `mmap`... [patching the copy as needed]
- ...the concatenation of the copies is a correct executable routine
- ...jump to the beginning of it

What Jitter does

(In its most efficient mode [1; 4]) Jitter:

- generates a huge C function containing all of the VM instructions specialisations

Then at runtime, the VM:

- copies compiled code for each VM instruction into executable memory, with `mmap`... [patching the copy as needed]
- ...the concatenation of the copies is a correct executable routine
- ...jump to the beginning of it

What Jitter does

(In its most efficient mode [1; 4]) Jitter:

- generates a huge C function containing all of the VM instructions specialisations

Then at runtime, the VM:

- copies compiled code for each VM instruction into executable memory, with `mmap`. . . [patching the copy as needed]
- . . . the concatenation of the copies is a correct executable routine
- . . . jump to the beginning of it

What Jitter does

(In its most efficient mode [1; 4]) Jitter:

- generates a huge C function containing all of the VM instructions specialisations

Then at runtime, the VM:

- copies compiled code for each VM instruction into executable memory, with `mmap`... [patching the copy as needed]
- ...the concatenation of the copies is a correct executable routine
- ...jump to the beginning of it

Doing this correctly is complicated

Generated code very portable, but complicated [1]

- If compiled by GCC it is very fast, thanks to non-portable GNU extensions and (dangerous) tricks [4] ["The fun of playing with fire"]
  - The dangerous tricks are hidden in the generated code: human-written specification is clean
  - You do not need to know the details to *use* Jitter
- The most efficient mode requires GCC...
- ...with other compilers slower modes: same behaviour, slower

Doing this correctly is complicated

Generated code very portable, but complicated [1]

- If compiled by GCC it is very fast, thanks to non-portable GNU extensions and (dangerous) tricks [4] ["The fun of playing with fire"]
  - The dangerous tricks are hidden in the generated code: human-written specification is clean
  - You do not need to know the details to *use* Jitter
- The most efficient mode requires GCC...
- ...with other compilers slower modes: same behaviour, slower

Doing this correctly is complicated

Generated code very portable, but complicated [1]

- If compiled by GCC it is very fast, thanks to non-portable GNU extensions and (dangerous) tricks [4] ["The fun of playing with fire"]
  - The dangerous tricks are hidden in the generated code: human-written specification is clean
  - You do not need to know the details to *use* Jitter

- The most efficient mode requires GCC...
- ...with other compilers slower modes: same behaviour, slower

Doing this correctly is complicated

Generated code very portable, but complicated [1]

- If compiled by GCC it is very fast, thanks to non-portable GNU extensions and (dangerous) tricks [4] ["The fun of playing with fire"]
  - The dangerous tricks are hidden in the generated code: human-written specification is clean
  - You do not need to know the details to *use* Jitter
- The most efficient mode requires GCC...
- ...with other compilers slower modes: same behaviour, slower

Doing this correctly is complicated

Generated code very portable, but complicated [1]

- If compiled by GCC it is very fast, thanks to non-portable GNU extensions and (dangerous) tricks [4] ["The fun of playing with fire"]
  - The dangerous tricks are hidden in the generated code: human-written specification is clean
  - You do not need to know the details to *use* Jitter
- The most efficient mode requires GCC...
- ...with other compilers slower modes: same behaviour, **slower**

(For technical reasons; still, **lovely, isn't it?**)

Doing this correctly is complicated

Generated code very portable, but complicated [1]

- If compiled by GCC it is very fast, thanks to non-portable GNU extensions and (dangerous) tricks [4] ["The fun of playing with fire"]
  - The dangerous tricks are hidden in the generated code: human-written specification is clean
  - You do not need to know the details to *use* Jitter
- The most efficient mode requires GCC...
- ...with other compilers slower modes: same behaviour, **slower**
  (For technical reasons; still, **lovely, isn't it?**)

Overflow arithmetic

add VM instruction: Jitter specification

```
instruction addo (?R, ?Rn 1 -1, !R, ?f)
  code
    JITTER_PLUS_BRANCH_FAST_IF_OVERFLOW (JITTER_ARGN2,
                                         JITTER_ARGN0, JITTER_ARGN1,
                                         JITTER_ARGF3);
  end
end
```

[Demo]

Overflow arithmetic

add VM instruction: Jitter specification

```
instruction addo (?R, ?Rn 1 -1, !R, ?f)
  code
    JITTER_PLUS_BRANCH_FAST_IF_OVERFLOW (JITTER_ARGN2,
                                         JITTER_ARGN0, JITTER_ARGN1,
                                         JITTER_ARGF3);
  end
end
```

[Demo]

Sub-package mode

Just like Gnulib, trivial to build for users:

- A copy of the Jitter sources distributed as part of the sources of the project using it
  - A sub-directory with its own `configure`, `Makefile.in`...
  - works automatically from `configure` or `make` in the super-package, following GNU conventions
    - out-of-tree builds;
    - `configure` options;
    - ...
    - even `make dist` works *automatically*!
- Idea suggested by José Marchesi
- See [2] about how this works

Sub-package mode

Just like Gnulib, trivial to build for users:

- A copy of the Jitter sources distributed as part of the sources of the project using it
  - A sub-directory with its own `configure`, `Makefile.in`...
  - works automatically from `configure` or `make` in the super-package, following GNU conventions
    - out-of-tree builds;
    - `configure` options;
    - ...
    - even `make dist` works automatically!
- Idea suggested by José Marchesi
- See [2] about how this works

Sub-package mode

Just like Gnulib, trivial to build for users:

- A copy of the Jitter sources distributed as part of the sources of the project using it
  - A sub-directory with its own `configure`, `Makefile.in`...
  - works automatically from `configure` or `make` in the super-package, following GNU conventions
    - out-of-tree builds;
    - `configure` options;
    - ...
    - even `make dist` works *automatically*!
- Idea suggested by José Marchesi
- See [2] about how this works

Sub-package mode

Just like Gnulib, trivial to build for users:

- A copy of the Jitter sources distributed as part of the sources of the project using it
  - A sub-directory with its own `configure`, `Makefile.in`...
  - works automatically from `configure` or `make` in the super-package, following GNU conventions
    - out-of-tree builds;
    - `configure` options;
    - ...
    - even `make dist` works *automatically*!
- Idea suggested by José Marchesi
- See [2] about how this works

Sub-package mode

Just like Gnulib, trivial to build for users:

- A copy of the Jitter sources distributed as part of the sources of the project using it
  - A sub-directory with its own `configure`, `Makefile.in`...
  - works automatically from `configure` or `make` in the super-package, following GNU conventions
    - out-of-tree builds;
    - `configure` options;
    - ...
    - even `make dist` works *automatically*!
- Idea suggested by José Marchesi
- See [2] about how this works

Sub-package mode

Just like Gnulib, trivial to build for users:

- A copy of the Jitter sources distributed as part of the sources of the project using it
  - A sub-directory with its own `configure`, `Makefile.in`...
  - works automatically from `configure` or `make` in the super-package, following GNU conventions
    - out-of-tree builds;
    - `configure` options;
    - ...
    - even `make dist` works *automatically*!
- Idea suggested by José Marchesi
- See [2] about how this works

Sub-package mode

Just like Gnulib, trivial to build for users:

- A copy of the Jitter sources distributed as part of the sources of the project using it
  - A sub-directory with its own `configure`, `Makefile.in`...
  - works automatically from `configure` or `make` in the super-package, following GNU conventions
    - out-of-tree builds;
    - `configure` options;
    - ...
    - even `make dist` works *automatically*!
- Idea suggested by José Marchesi
- See [2] about how this works

Sub-package mode

Just like Gnulib, trivial to build for users:

- A copy of the Jitter sources distributed as part of the sources of the project using it
  - A sub-directory with its own `configure`, `Makefile.in`...
  - works automatically from `configure` or `make` in the super-package, following GNU conventions
    - out-of-tree builds;
    - `configure` options;
    - ...
    - even `make dist` works *automatically*!
- Idea suggested by José Marchesi
- See [2] about how this works

Jitter comes with extensive examples

Jitter comes with extensive examples:

- The Structured language (an Algol- or Pascal-like language), with two different backends [6]
  - stack VM
  - register VM
- JitterLisp (it has its own manual [the least incomplete part of Jitter's documentation])
- The Uninspired VM (easy to experiment with: the program is the VM, with nothing more)

Jitter comes with extensive examples

Jitter comes with extensive examples:

- The Structured language (an Algol- or Pascal-like language), with two different backends [6]
  - stack VM
  - register VM
- JitterLisp (it has its own manual [the least incomplete part of Jitter's documentation])
- The Uninspired VM (easy to experiment with: the program is the VM, with nothing more)

Jitter comes with extensive examples

Jitter comes with extensive examples:

- The Structured language (an Algol- or Pascal-like language), with two different backends [6]
  - stack VM
  - register VM
- JitterLisp (it has its own manual [the least incomplete part of Jitter's documentation])
- The Uninspired VM (easy to experiment with: the program is the VM, with nothing more)

Jitter comes with extensive examples

Jitter comes with extensive examples:

- The Structured language (an Algol- or Pascal-like language), with two different backends [6]
  - stack VM
  - register VM
- JitterLisp (it has its own manual [the least incomplete part of Jitter's documentation])
- The Uninspired VM (easy to experiment with: the program is the VM, with nothing more)

Status

A Jittery VM powers:

- GNU epsilon (soon)
- GNU poke [José Marchesi was the first Jitter user] (now)
- . . . I would like to propose this to other projects . . . any GNU Smalltalk hackers here?

Status

A Jittery VM powers:

- GNU epsilon (soon)
- GNU poke [José Marchesi was the first Jitter user] (now)
- . . . I would like to propose this to other projects . . . any GNU Smalltalk hackers here?

Status

A Jittery VM powers:

- GNU epsilon (soon)
- GNU poke [José Marchesi was the first Jitter user] (now)

- . . . I would like to propose this to other projects
  - any GNU Smalltalk hackers here?

Status

A Jittery VM powers:

- GNU epsilon (soon)
- GNU poke [José Marchesi was the first Jitter user] (now)

- . . . I would like to propose this to other projects
  - any GNU Smalltalk hackers here?

Any questions?

https://gnu.org/s/jitter

You are welcome to subscribe to the mailing list

Thanks!

Any questions?

https://gnu.org/s/jitter

You are welcome to subscribe to the mailing list

Thanks!

Bibliography

[1] Saiu, L. (2017). The art of the language VM *or* Machine-generating virtual machine code *or* Almost zero overhead with almost zero assembly *or* My virtual machine is faster than yours. GNU Hackers' Meeting 2017, Knüllwald-Niederbeisheim, Germany, August 2017. The first public presentation about Jitter, still useful as an introduction. Slides and video recording available from https://ageinghacker.net/talks/.

[2] Saiu, L. (2019). Sub-packages, dependencies and information flow. GNU Hackers' Meeting 2019, Madrid, Spain, August 2019. Slides available from https://ageinghacker.net/talks/.

[3] Saiu, L. (2021). Informal Jitter talk. Informal live presentation, March 2021. A friendly talk including a live demo, mostly improvised and not particularly well prepared, with friends from the GNU poke project. Video recording available from https://archive.org/details/jitter-presentation--2021-03-25.

Bibliography II

[4] Saiu, L. (2022a). GNU Jitter and the illusion of simplicity *or* Copying, patching and combining compiler-generated code in executable memory *or* The Anarchist's guide to GCC *or* The fun of playing with fire. Binary T00ls Summit, online event, 2022. A technical talk about code generation and how it interacts with GCC optimisations. Slides available from http://ageinghacker.net/talks, video recording available at https://binary-tools.net/summit.html.

[5] Saiu, L. (2022b). In defence of language as an interface — a statement of the obvious. GNU Hackers' Meeting 2022, İzmir, Turkey. Slides available at https://ageinghacker.net/talks/language-slides--saiu--ghm-2022--2022-10-01.pdf; Video recording available at https://audio-video.gnu.org/video/ghm2022/2022-10--language--saiu--ghm.webm.

[6] Saiu, L. (2023). Jitter (unfinished tutorial). Available at https://ageinghacker.net/projects/jitter-tutorial/.