Bril: A Compiler Intermediate Representation for Learning

Bril, the Big Red Intermediate Language, is a programming language for learning about compilers. It's the intermediate representation we use in CS 6120, a PhD-level compilers course. Bril's design tenets include:

- Bril is an instruction-oriented language, like most good IRs.
- The core is minimal and ruthlessly regular. Extensions make it interesting.
- The tooling is language agnostic. Bril programs are just JSON.
- Bril is typed.

See the language reference for the complete language specification and the tool documentation for details on the "batteries included" monorepo.

Language Reference

This section describes the Bril language exhaustively. The Syntax chapter is about the structure of the language—the abstract syntax. Then, there are several chapters about the actual operations that fit into this general structure. The core language has the basics, including integers, Booleans, and control flow. Other extensions add optional functionality on top of this simple core language. You are cordially invented to add your own similar extensions to the Bril language!

Syntax Reference

Bril programs are JSON objects that directly represent abstract syntax. This chapter exhaustively describes the structure of that syntax. All objects are JSON values of one sort or another.

Program

```
{ "functions": [<Function>, ...] }
```

A Program is the top-level object. It has one key:

- `functions`, a list of Function objects.

Type

```
"<string>"
{"<string>": <Type>}
```

There are two kinds of types: primitive types, whose syntax is just a string, and parameterized types, which wrap a smaller type. The semantics chapters list the particular types that are available—for example, core Bril defines the basic primitive types `int` and `bool` and the memory extension defines a parameterized pointer type.

Function

```
{
  "name": "<string>",
  "args": [{"name": "<string>", "type": <Type>}, ...]?,
  "type": <Type>?,
  "instrs": [<Instruction>, ...]
}
```

A Function object represents a (first-order) procedure consisting of a sequence of instructions. There are four fields:

- `name`, a string.
- `args`, optionally, a list of arguments, which consist of a `name` and a `type`. Missing `args` is the same as an empty list.
- Optionally, `type`, a Type object: the function's return type, if any.
- `instrs`, a list of Label and Instruction objects.

When a function runs, it creates an activation record and transfers control to the first instruction in the sequence.

A Bril program is executable if it contains a function named `main`. When execution starts, this function will be invoked. The `main` function can have arguments (which implementations may supply using command-line arguments) but must not have a return `type`.

Label

```
{ "label": "<string>" }
```

A Label marks a position in an instruction sequence as a destination for control transfers. It only has one key:

- `label`, a string. This is the name that jump and branch instructions will use to transfer control to this position and proceed to execute the following instruction.

Instruction

```
{ "op": "<string>", ... }
```

An Instruction represents a unit of computational work. Every instruction must have this field:

- `op`, a string: the *opcode* that determines what the instruction does. (See the *Core Language* section and the subsequent extension sections for listings of the available opcodes.)

Depending on the opcode, the instruction might also have:

- `dest`, a string: the name of the variable where the operation's result is stored.
- `type`, a Type object: the type of the destination variable.
- `args`, a list of strings: the arguments to the operation. These are names of variables.
- `funcs`, a list of strings: any names of functions referenced by the instruction.

- `labels`, a list of strings: any label names referenced by the instruction.

There are three kinds of instructions: constants, value operations, and effect operations.

Constant

```
{ "op": "const", "dest": "<string>", "type": <Type>,
  "value": <literal> }
```

A Constant is an instruction that produces a literal value. Its `op` field must be the string `"const"`. It has the `dest` and `type` fields described above, and also:

- `value`, the literal value for the constant. This is either a JSON number or a JSON Boolean value. The `type` field must match—i.e., it must be "int" or "bool", respectively.

Value Operation

```
{ "op": "<string>", "dest": "<string>", "type": <Type>,
  "args": ["<string>", ...]?,
  "funcs": ["<string>", ...]?,
  "labels": ["<string>", ...]? }
```

A Value Operation is an instruction that takes arguments, does some computation, and produces a value. Like a Constant, it has the `dest` and `type` fields described above, and also any of these three optional fields:

- `args`, a list of strings. These are variable names defined elsewhere in the same function.
- `funcs`, a list of strings. The names of any functions that this instruction references. For example, core Bril's call instruction takes one function name.
- `labels`, a list of strings. The names of any labels within the current function that the instruction references. For example, core Bril's jump and branch instructions have target labels.

In all three cases, these keys may be missing and the semantics are identical to mapping to an empty list.

Effect Operation

```
{ "op": "<string>",
  "args": ["<string>", ...]?,
  "funcs": ["<string>", ...]?,
  "labels": ["<string>", ...]? }
```

An Effect Operation is like a Value Operation but it does not produce a value. It also has the optional `args`, `funcs`, and `labels` fields.

Source Positions

Any syntax object may optionally have position fields to reflect a source position:

```
{ ..., "pos": {"row": <int>, "col": <int>},
       "pos_end": {"row": <int>, "col": <int>}?,
       "src": "<string>"? }
```

The `pos` and `pos_end` objects have two keys: `row` (the line number) and `col` (the column number within the line). The `src` object can optionally provide the absolute path to a file which is referenced to by the source position. If `pos_end` is provided, it must be equal to or greater than `pos`. Front-end compilers that generate Bril code may add this information to help with debugging. The text format parser, for example, can optionally add source positions. However, tools can't require positions to exist, to consistently exist or not on all syntax objects in a program, or to follow any particular rules.

Well Formedness

Not every syntactically complete Bril program is *well formed*. Here is an incomplete list of rules that well-formed Bril programs must follow:

- Instructions may name variables as arguments when they are defined elsewhere in the function. Similarly, they may only refer to labels that exist within the same function, and they can only refer to functions defined somewhere in the same file.
- Dynamically speaking, during execution, instructions may refer only to variables that have already been defined earlier in execution. (This is a dynamic property, not a static property.)
- Every variable may have only a single type within a function. It is illegal to have two assignments to the same variable with different types, even if the function's logic guarantees that it is impossible to execute both instructions in a single call.
- Many operations have constraints on the types of arguments they can take; well-formed programs always provide the right type of value.

Tools do not need to handle ill-formed Bril programs. As someone working with Bril, you never need to check for well-formedness and can do anything when fed with ill-formed code, including silently working just fine, producing ill-formed output, or crashing and burning.

To help check for well-formedness, the reference interpreter has many dynamic checks and the type inference tool can check types statically.

Core Language

This section describes the *core* Bril instructions. Any self-respecting Bril tool must support all of these operations; other extensions are more optional.

Types

Core Bril defines two primitive types:

- `int` : 64-bit, two's complement, signed integers.
- `bool` : True or false.

Arithmetic

These instructions are the obvious binary integer arithmetic operations. They all take two arguments, which must be names of variables of type `int`, and produce a result of type `int`:

- `add` : x + y.
- `mul` : x × y.
- `sub` : x - y.
- `div` : x ÷ y.

In each case, overflow follows two's complement rules. It is an error to `div` by zero.

Comparison

These instructions compare integers. They all take two arguments of type `int` and produce a result of type `bool`:

- `eq` : Equal.
- `lt` : Less than.
- `gt` : Greater than.
- `le` : Less than or equal to.
- `ge` : Greater than or equal to.

Logic

These are the basic Boolean logic operators. They take arguments of type `bool` and produce a result of type `bool`:

- `not` (1 argument)
- `and` (2 arguments)
- `or` (2 arguments)

Control

These are the control flow operations. Unlike the value operations above, they take labels and functions in addition to normal arguments.

- `jmp`: Unconditional jump. One label: the label to jump to.
- `br`: Conditional branch. One argument: a variable of type `bool`. Two labels: a true label and a false label. Transfer control to one of the two labels depending on the value of the variable.
- `call`: Function invocation. Takes the name of the function to call and, as its arguments, the function parameters. The `call` instruction can be a Value Operation or an Effect Operation, depending on whether the function returns a value.
- `ret`: Function return. Stop executing the current activation record and return to the parent (or exit the program if this is the top-level main activation record). It has one optional argument: the return value for the function.

Only `call` may (optionally) produce a result; the rest appear only as Effect Operations.

Miscellaneous

- `id`: A type-insensitive identity. Takes one argument, which is a variable of any type, and produces the same value (which must have the same type, obvi).
- `print`: Output values to the console (with a newline). Takes any number of arguments of any type and does not produce a result.
- `nop`: Do nothing. Takes no arguments and produces no result.

Static Single Assignment (SSA) Form

This language extension lets you represent Bril programs in static single assignment (SSA) form. As in the standard definition, an SSA-form Bril program contains only one assignment per variable, globally—that is, variables within a function cannot be reassigned. This extension adds ϕ-nodes to the language.

Operations

There is one new instruction:

- `phi` : Takes *n* labels and *n* arguments, for any *n*. Copies the value of the *i*th argument, where *i* is the index of the second-most-recently-executed label. (It is an error to use a `phi` instruction when two labels have not yet executed, or when the instruction does not contain an entry for the second-most-recently-executed label.)

Intuitively, a `phi` instruction takes its value according to the current basic block's predecessor.

Examples

In the text format, you can write `phi` instructions like this:

```
x: int = phi a .here b .there;
```

The text format doesn't care how you interleave arguments and labels, so this is equivalent to (but more readable than) `phi a b .here .there`. The "second-most-recent label" rule means that the labels refer to predecessor basic blocks, if you imagine blocks being "named" by their labels.

Here's a small example:

```
.top:
  a: int = const 5;
  br cond .here .there;
.here:
  b: int = const 7;
.there:
  c: int = phi a .top b .here;
  print c;
```

A `phi` instruction is sensitive to the incoming CFG edge that execution took to arrive at the current block. The `phi` instruction in this program, for example, gets its value from `a` if control came from the `.top` block and `b` if control came from the `.here` block.

The reference interpreter can supports programs in SSA form because it can faithfully execute the `phi` instruction.

Manually Managed Memory

While core Bril only has simple scalar stack values, the memory extension adds a manually managed heap of array-like allocations. You can create regions, like with `malloc` in C, and it is the program's responsibility to delete them, like with `free`. Programs can manipulate pointers within these regions; a pointer indicates a particular offset within a particular allocated region.

You can read more about the memory extension from its creators, Drew Zagieboylo and Ryan Doenges.

Types

The memory extension adds a parameterized `ptr` type to Bril:

```
{"ptr": <Type>}
```

A pointer value represents a reference to a specific offset within a uniformly-typed region of values.

Operations

These are the operations that manipulate memory allocations:

- `alloc`: Create a new memory region. One argument: the number of values to allocate (an integer). The result type is a pointer; the type of the instruction decides the type of the memory region to allocate. For example, this instruction allocates a region of integers:

  ```
  {
      "op": "alloc",
      "args": ["size"],
      "dest": "myptr",
      "type": {"ptr": "int"}
  }
  ```

- `free`: Delete an allocation. One argument: a pointer produced by `alloc`. No return value.

- `store` : Write into a memory region. Two arguments: a pointer and a value. The pointer type must agree with the value type (e.g., if the second argument is an `int` , the first argument must be a `ptr<int>`). No return value.
- `load` : Read from memory. One argument: a pointer. The return type is the pointed-to type for that pointer.
- `ptradd` : Adjust the offset for a pointer, producing a new pointer to a different location in the same memory region. Two arguments: a pointer and an offset (an integer, which may be negative). The return type is the same as the original pointer type.

It is an error to access or free a region that has already been freed. It is also an error to access (`load` or `store`) a pointer that is out of bounds, i.e., outside the range of valid indices for a given allocation. (Doing a `ptradd` to produce an out-of-bounds pointer is not an error; subsequently accessing that pointer is.)

Printing

It is not an error to use the core `print` operation on pointers, but the output is not specified. Implementations can choose to print any representation of the pointer that they deem helpful.

Floating Point

Bril has an extension for computing on floating-point numbers.

You can read more about the extension, which is originally by Dietrich Geisler and originally included two FP precision levels.

Types

The floating point extension adds one new base type:

```
"float"
```

Floating point numbers are 64-bit, double-precision IEEE 754 values. (There is no single-precision type.)

Operations

There are the standard arithmetic operations, which take two `float` values and produce a new `float` value:

- `fadd`
- `fmul`
- `fsub`
- `fdiv`

It is not an error to `fdiv` by zero; as in IEEE 754, the result is infinity.

There are also comparison operators, which take two `float` values and produce a `bool`:

- `feq`
- `flt`
- `fle`
- `fgt`
- `fge`

Printing

The core `print` operation prints `float` values with 17 decimal digits of precision, including trailing zeros. (This is like using the `%.17lf` format specifier in C's `printf`.) Positive and negative zero, while they are equal according to `feq`, look different when printed. Not-a-number values are printed as `NaN`; infinite values are printed as the strings `Infinity` or `-Infinity`.

Speculative Execution

This extension lets Bril programs use a form of explicit speculative execution with rollback.

In general, *speculation* is when programs perform work that might not actually be necessary or even correct, under the assumption that it is *likely* to be right and useful. If this assumption turns out to be wrong, speculation typically needs some *rollback* mechanism to undo incorrect side effects and recover to a correct state.

In this Bril extension, programs can explicitly enter a *speculative mode*, where variable assignments are temporary. Then, they can either abort or commit those assignments, discarding them or making them permanent.

Operations

- `speculate` : Enter a speculative execution context. No arguments.
- `commit` : End the current speculative context, committing the current speculative state as the "real" state. No arguments.
- `guard` : Check a condition and possibly abort the current speculative context. One argument, the Boolean condition, and one label, to which control is transferred on abort. If the condition is true, this is a no-op. If the condition is false, speculation aborts: the program state rolls back to the state at the corresponding `speculate` instruction, execution jumps to the specified label.

Speculation can be nested, in which case aborting or committing a child context returns execution to the parent context. Aborting speculation rolls back normal variable assignments, but it does not affect the memory extension's heap—any changes there remain. It is an error to commit or abort outside of speculation. It is not an error to perform side effects like `print` during speculation, but it is probably a bad idea.

Examples

Committing a speculative update makes it behave like normal:

```
v: int = const 4;
speculate;
v: int = const 2;
commit;
print v;
```

So this example prints `2`. However, when a guard fails, it rolls back any modifications that happened since the last `speculate` instruction:

```
  b: bool = const false;

  v: int = const 4;
  speculate;
  v: int = const 2;
  guard b .failed;
  commit;

.failed:
  print v;
```

The guard here fails because `b` is false, then `v` gets restored to its pre-speculation value, and then control transfers to the `.failed` label. So this example prints `4`. You can think of the code at `.failed` as the "recovery routine" that handles exceptional conditions.

Interpreter

The reference interpreter supports speculative execution. However, it does not support function calls during speculation, so you will get an error if you try to use a `call` or `ret` instruction while speculating.

Import

Typically, Bril programs are self-contained: they only use functions defined elsewhere in the same program. This *import* extension lets Bril code use functions defined in other files.

A Bril import refers to a file and lists the functions to import from it, like this:

```
{
    "path": "my_library.json",
    "functions": [{"name": "libfunc"}]
}
```

This import assumes that there's a Bril file called `my_library.json`, and that it declares a function `@libfunc`. The current Bril file may now invoke `@libfunc` as if it were defined locally.

Syntax

The top-level Bril program is extended with an `imports` field:

```
{ "functions": [<Function>, ...], "imports": [<Import>, ...] }
```

Each import object has this syntax:

```
{
    "path": "<string>",
    "functions": [
        { "name": "<string>", "alias": "<string>"? },
        ...
    ]
}
```

The path is a relative reference to a Bril JSON file containing the functions to import. In the objects in the `functions` list, the `name` is the *original* name of the function, and the optional `alias` is the *local* name that the program will use to refer to the function. A missing `alias` makes the local name equal to the original name.

It is an error to refer to functions that do not exist, or to create naming conflicts between imports and local functions (or between different imports). Import cycles are allowed.

Text Format

In Bril's text format, the `import` syntax looks like this:

```
from "something.json" import @libfunc, @otherfunc as @myfunc;
```

Search Paths

We do not define the exact mechanism for using the `path` string to find the file to import. Reasonable options include:

- Resolve the path relative to the file the `import` appears in.
- Use a pre-defined set of library search paths.

We only specify what it means to import JSON files; implementations can choose to allow importing other kinds of files too (e.g., text-format source code).

Character

Types

The character extension adds one new base type:

```
"char"
```

Characters are a singular Unicode character.

Operations

Comparison operators, which take two `char` values and produce a `bool`:

- `ceq`
- `clt`
- `cle`
- `cgt`
- `cge`

Conversion operators:

- `char2int`: One argument: a variable of type `char`. Returns an integer representing the Unicode code point of the given value.
- `int2char`: One argument: a variable of type `int`. Returns the corresponding Unicode character. Throws if the value does not correspond to a valid Unicode code point.

Printing

The core `print` operation prints `char` values.

Bril Tools

These sections describe tools for dealing with Bril programs.

Interpreter

`brili` is the reference interpreter for Bril. It is written in TypeScript. You can find `brili` in the `bril-ts` directory in the Bril repository.

The interpreter supports core Bril along with the memory, floating point, SSA, and speculation extensions.

Install

To use the interpreter, you will need Deno. Just run:

```
$ deno install brili.ts
```

As Deno tells you, you will then need to add `$HOME/.deno/bin` to your `$PATH`.

Run

The `brili` program takes a Bril program as a JSON file on standard input:

```
$ brili < my_program.json
```

It emits any `print` outputs to standard output. To provide inputs to the main function, you can write them as command-line arguments:

```
$ brili 37 5 < add.json
42
```

Profiling

The interpreter has a rudimentary profiling mode. Add a `-p` flag to print out a total number of dynamic instructions executed to stderr:

```
$ brili -p 37 5 < add.json
42
total_dyn_inst: 9
```

Bril Text Format

While Bril's canonical representation is a JSON AST, humans don't like to read and write JSON. To accommodate our human foibles, we also have a simple textual representation. There is a parser and pretty printer tool that can convert the text representation to and from JSON.

For example, this Bril program in JSON:

```
{
  "functions": [{
    "name": "main",
    "instrs": [
      { "op": "const", "type": "int", "dest": "v0", "value": 1 },
      { "op": "const", "type": "int", "dest": "v1", "value": 2 },
      { "op": "add", "type": "int", "dest": "v2", "args": ["v0", "v1"] },
      { "op": "print", "args": ["v2"] },
      { "op": "alloc", "type": { "ptr" : "int" }, "dest": "v3", "args": ["v0"] },
      { "op": "free", "args": ["v3"] },
    ]
  }]
}
```

Gets represented in text like this:

```
@main {
  v0: int = const 1;
  v1: int = const 2;
  v2: int = add v0 v1;
  print v2;
  v3: ptr<int> = alloc v0;
  free v3;
}
```

Tools

The `bril-txt` parser & pretty printer are written in Python. You can install them with Flit by doing something like:

```
$ pip install --user flit
$ cd bril-txt
$ flit install --symlink --user
```

You'll now have tools called `bril2json` and `bril2txt`. Both read from standard input and write to standard output. You can try a "round trip" like this, for example:

```
$ bril2json < test/parse/add.bril | bril2txt
```

The `bril2json` parser also supports a `-p` flag to include source positions.

TypeScript-to-Bril Compiler

Bril comes with a compiler from a very small subset of TypeScript to Bril called `ts2bril`.

It is not supposed to make it easy to port existing JavaScript code to Bril; it is a convenient way to write larger, more interesting programs without manually fiddling with Bril directly. It also emits somewhat obviously inefficient code to keep the compiler simple; some obvious optimizations can go a long way.

Install

The TypeScript compiler uses Deno. Type this:

```
$ deno install --allow-env --allow-read ts2bril.ts
```

If you haven't already, you will then need to add `$HOME/.deno/bin` to [your `$PATH`][path].

Use

Compile a TypeScript program to Bril by giving a filename on the command line:

```
$ ts2bril mycode.ts
```

The compiler supports both integers (from core Bril) and floating point numbers. Perhaps somewhat surprisingly, plain JavaScript numbers and the TypeScript `number` type map to `float` in Bril. For integers, use JavaScript big integers whenever you need an integer, like this:

```
var x: bigint = 5n;
printInt(x);

function printInt(x: bigint) {
    console.log(x);
}
```

The `n` suffix on literals distinguishes integer literals, and the `bigint` type in TypeScript reflects them.

Fast Interpreter in Rust

The `brilirs` directory contains a fast Bril interpreter written in Rust. It is a drop-in replacement for the reference interpreter that prioritizes speed over completeness and hackability. It implements core Bril along with the SSA, memory, char, and floating point extensions.

Read more about the implementation, which is originally by Wil Thomason and Daniel Glus.

Install

To use `brilirs` you will need to install Rust. Use `echo $PATH` to check that `$HOME/.cargo/bin` is on your path.

In the `brilirs` directory, install the interpreter with:

```
$ cargo install --path .
```

During installation, `brilirs` will attempt to create a tab completions file for current shell. If this of interest, follow the instructions provided as a warning to finish enabling this.

Run a program by piping a JSON Bril program into it:

```
$ bril2json < myprogram.bril | brilirs
```

or

```
$ brilirs --text --file myprogram.bril
```

Similar to brilck, `brilirs` can be used to typecheck and validate your Bril JSON program by passing the `--check` flag (similar to `cargo --check`).

To see all of the supported flags, run:

```
$ brilirs --help
```

Syntax Plugin for Text Editors

There is a Vim syntax highlighting plugin for Bril's text format available in `bril-vim`. You can use it with a Vim plugin manager. For example, if you use vim-plug, you can add this to your `.vimrc`:

```
Plug 'sampsyo/bril', { 'for': 'bril', 'rtp': 'bril-vim' }
```

You can read more about the plugin, which is originally by Edwin Peguero.

Type Inference

Bril requires exhaustive type annotations on every instruction, which can quickly get tedious. The `type-infer` directory contains a simple global type inference tool that fills in missing type annotations. For example, it can turn this easier-to-write program:

```
@main(arg: int) {
  five = const 5;
  ten = const 10;
  res = add arg five;
  cond = le res ten;
  br cond .then .else;
.then:
  print res;
.else:
}
```

Into this actually executable program:

```
@main(arg: int) {
  five: int = const 5;
  ten: int = const 10;
  res: int = add arg five;
  cond: bool = le res ten;
  br cond .then .else;
.then:
  print res;
.else:
}
```

The tool is a simple Python program, `infer.py`, that takes JSON programs that are missing types and adds types to them. It is also useful even on fully-typed programs as a type *checker* to rule out common run-time errors. The included text format tools support missing types for both parsing and printing, so here's a shell pipeline that adds types to your text-format Bril program:

```
cat myprog.bril | bril2json | python type-infer/infer.py | bril2txt
```

You can read more about the inference tool, which is originally by Christopher Roman.

Type Checker

Bril comes with a simple type checker to catch errors statically. It checks the types of instructions in the core language and the floating point, SSA, memory, and speculation extensions. It also checks calls and return values and the labels used in control flow.

Install

The `brilck` tool uses Deno. Type this:

```
$ deno install brilck.ts
```

If you haven't already, you will then need to add `$HOME/.deno/bin` to [your `$PATH`][path].

Check

Just pipe a Bril program into `brilck`:

```
bril2json < benchmarks/fizz-buzz.bril | brilck
```

It will print any problems it finds to standard error. (If it doesn't find any problems, it doesn't print anything at all.)

You can optionally provide a filename as a (sole) command-line argument. This filename will appear in any error messages for easier parsing when many files are involved.

Consider supplying the `-p` flag to the `bril2json` parser to get source positions in the error messages.

Benchmarks

The `bench` directory in the Bril repository contains a fledgling suite of microbenchmarks that you can use to measure the impact of your optimizations. (Benchmarks are different from tests because they are meant to actually calculate something instead of just exercising a language feature.)

The current benchmarks are:

- `ackermann` : Print the value of `Ack(m, n)` , the two-argument Ackermann–Péter function.
- `adj2csr` : Convert a graph in adjacency matrix format (dense representation) to Compressed Sparse Row (CSR) format (sparse representation). The random graph is generated using the same linear congruential generator.
- `adler32` : Computes the Adler-32 Checksum of an integer array.
- `armstrong` : Determines if the input is an Armstrong number, a number that is the sum of its own digits each raised to the power of the number of digits.
- `binary-fmt` : Print the binary format for the given positive integer.
- `binary-search` : Search a target integer within an integer array, outputs the index of target.
- `birthday` : Simulation of the birthday paradox with an input of `n` people in a given room.
- `bitwise-ops` : Computes the OR, AND, or XOR between two 64-bit integers. (Three modes: 0 = AND, 1 = OR, 2 = XOR)
- `bitshift` : Computes the LEFTSHIFT and RIGHTSHIFT for any integer, also implements an efficient pow function for integers
- `bubblesort` : Sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- `catalan` : Print the *n*th term in the Catalan sequence, compute using recursive function calls.
- `check-primes` : Check the first *n* natural numbers for primality, printing out a 1 if the number is prime and a 0 if it's not.
- `cholesky` : Perform Cholesky decomposition of a Hermitian and positive definite matrix. The result is validated by comparing with Python's `scipy.linalg.cholesky` .
- `collatz` : Print the Collatz sequence starting at *n*. Note: it is not known whether this will terminate for all *n*.
- `conjugate-gradient` : Uses conjugate gradients to solve `Ax=b` for any arbitrary positive semidefinite `A` .
- `cordic` : Print an approximation of sine(radians) using 8 iterations of the CORDIC algorithm.
- `csrmv` : Multiply a sparse matrix in the Compressed Sparse Row (CSR) format with a dense vector. The matrix and input vector are generated using a Linear Feedback Shift Register

random number generator.

- `digial-root` : Computes the digital root of the input number.
- `dead-branch` : Repeatedly call a br instruction whose condition always evaluates to false. The dead branch should be pruned by a smart compiler.
- `dot-product` : Computes the dot product of two vectors.
- `eight-queens` : Counts the number of solutions for *n* queens problem, a generalization of Eight queens puzzle.
- `euclid` : Calculates the greatest common divisor between two large numbers using the Euclidean Algorithm with a helper function for the modulo operator.
- `euler` : Approximates Euler's number using the Taylor series.
- `fact` : Prints the factorial of *n*, computing it recursively.
- `factors` : Print the factors of the *n* using the trial division method.
- `fib` : Calculate the *n*th Fibonacci number by allocating and filling an array of numbers up to that point.
- `fizz-buzz` : The infamous programming test.
- `function_call` : For benchmarking the overhead of simple function calls.
- `gcd` : Calculate Greatest Common Divisor (GCD) of two input positive integer using Euclidean algorithm.
- `hanoi` : Print the solution to the *n*-disk Tower of Hanoi puzzle.
- `is-decreasing` : Print if a number contains strictly decreasing digits.
- `lcm` : Compute LCM for two numbers using a very inefficient loop.
- `leibniz` : Approximates Pi using Leibniz formula.
- `loopfact` : Compute *n!* imperatively using a loop.
- `major-elm` : Find the majority element in an array using a linear time voting algorithm.
- `mandelbrot` : Generates a really low resolution, ascii, mandelbrot set.
- `mat-inv` : Calculates the inverse of a 3x3 matrix and prints it out.
- `mat-mul` : Multiplies two `nxn` matrices using the naive matrix multiplication algorithm. The matrices are randomly generated using a linear congruential generator.
- `max-subarray` : solution to the classic Maximum Subarray problem.
- `mod_inv` : Calculates the modular inverse of `n` under to a prime modulus p.
- `newton` : Calculate the square root of 99,999 using the newton method
- `norm` : Calculate the euclidean norm of a vector
- `n_root` : Calculate nth root of a float using newton's method.
- `orders` : Compute the order ord(u) for each u in a cyclic group <Zn,+> of integers modulo *n* under the group operation + (modulo *n*). Set the second argument *is_lcm* to true if you would like to compute the orders using the lowest common multiple and otherwise the program will use the greatest common divisor.
- `pascals-row` : Computes a row in Pascal's Triangle.
- `palindrome` : Outputs a 0-1 value indicating whether the input is a palindrome number.
- `perfect` : Check if input argument is a perfect number. Returns output as Unix style return code.

- `pow` : Computes the n^th power of a given (float) number.
- `primes-between` : Print the primes in the interval `[a, b]` .
- `primitive-root` : Computes a primitive root modulo a prime number input.
- `pythagorean_triple` : Prints all Pythagorean triples with the given c, if such triples exist. An intentionally very naive implementation.
- `quadratic` : The quadratic formula, including a hand-rolled implementation of square root.
- `quickselect` : Find the kth smallest element in an array using the quickselect algorithm.
- `quicksort` : Quicksort using the Lomuto partition scheme.
- `quicksort-hoare` : Quicksort using Hoare partioning and median of three pivot selection.
- `recfact` : Compute *n!* using recursive function calls.
- `rectangles-area-difference` : Output the difference between the areas of rectangles (as a positive value) given their respective side lengths.
- `fitsinside` : Output whether or not a rectangle fits inside of another rectangle given the width and height lengths.
- `relative-primes` : Print all numbers relatively prime to *n* using Euclidean algorithm.
- `riemann` : Prints the left, midpoint, and right Riemann Sums for a specified function, which is the square function in this benchmark.
- `sieve` : Print all prime numbers up to *n* using the Sieve of Eratosthenes.
- `sqrt` : Implements the Newton–Raphson Method of approximating the square root of a number to arbitrary precision
- `sum-bit` : Print the number of 1-bits in the binary representation of the input integer.
- `sum-check` : Compute the sum of [1, n] by both loop and formula, and check if the result is the same.
- `sum-divisors` : Prints the positive integer divisors of the input integer, followed by the sum of the divisors.
- `sum-sq-diff` : Output the difference between the sum of the squares of the first *n* natural numbers and the square of their sum.
- `totient` : Computes Euler's totient function on an input integer *n*.
- `two-sum` : Print the indices of two distinct elements in the list [2, 7, 11, 13] whose sum equals the input.
- `up-arrow` : Computes Knuth's up arrow notation, with the first argument being the number, the second argument being the number of Knuth's up arrows, and the third argument being the number of repeats.
- `vsmul` : Multiplies a constant scalar to each element of a large array. Tests the performance of vectorization optimizations.
- `reverse` : Compute number with reversed digits (e.g. 123 -> 321).

Credit for several of these benchmarks goes to Alexa VanHattum and Gregory Yauney, who implemented them for their global value numbering project.

TypeScript Library

`bril-ts` is a TypeScript library for interacting with Bril programs. It is the basis for the reference interpreter and the included type checker, but it is also useful on its own.

The library includes:

- `bril.ts` : Type definitions for the Bril language. Parsing a JSON file produces a value of type `Program` from this module.
- `builder.ts` : A builder class that makes it more convenient to generate Bril programs from front-end compilers.
- `types.ts` : A description of the type signatures for Bril operations, including the core language and all currently known extensions.

OCaml Library

The OCaml `bril` library, which lives in the `bril-ocaml` directory, provides an OCaml interface and parser for Bril's JSON files.

Install

To build the library, you first need to install OCaml. Then, install the dependencies with `opam install core yojson`.

To install the bril-ocaml library:

```
git clone https://github.com/sampsyo/bril path/to/my/bril
opam pin add -k path bril path/to/brill/bril-ocaml
opam install bril
```

That's it! You can include it in your Dune files as `bril`, like any other OCaml library.

Use

The interface for the library can be found in `bril.mli` —good starting points are `from_string`, `from_file`, and `to_string`. A small code example for the library lives in the `count` subdirectory.

If you wish to make changes to the bril OCaml library, simply hack on the git clone.

When you are done, simply reinstall the package with `opam reinstall bril`. Restart the build of your local project to pick up changes made to bril-ocaml.

For Development

ocamlformat is recommended for style consistency. The dune documentation on Automatic Formatting has information about using ocamlformat with dune.

Rust Library

This is a no-frills interface between Bril's JSON and your Rust code. It supports the Bril core along with the SSA, memory, floating point, speculative execution, char, and source positions extensions.

Use

Include this by adding the following to your `Cargo.toml`:

```
[dependencies.bril-rs]
version = "0.1.0"
path = "../bril-rs"
features = ["ssa", "memory", "float", "speculate", "position"]
```

Each of the extensions to Bril core is feature gated. To ignore an extension, remove its corresponding string from the `features` list.

There are two helper functions: `load_program` will read a valid Bril program from stdin, and `output_program` will write your Bril program to stdout. Otherwise, this library can be treated like any other serde JSON representation.

Tools

This library supports fully compatible Rust implementations of `bril2txt` and `bril2json`. This library also implements the import extension with a static linker called `brild`.

This library is used in a Rust compiler called `rs2bril` which supports generating core, float, and memory Bril from a subset of valid Rust.

This library is used in a Bril-to-LLVM IR compiler called `brillvm` which supports core, float, memory, and ssa.

For ease of use, these tools can be installed and added to your path by running the following in `bril-rs/`:

```
$ make install
```

Make sure that `~/.cargo/bin` is on your path. Each of these tools supports the `--help` flag which specifies some helpful flags.

Development

To maintain consistency and cleanliness, run:

```
cargo fmt
cargo clippy
cargo doc
make test
make features
```

Brench

Brench is a simple benchmark runner to help you measure the impact of optimizations. It can run the same set of benchmarks under multiple treatments, check that they still produce the correct answer, and report their performance under every condition.

Set Up

Brench is a Python tool. There is a `brench/` subdirectory in the Bril repository. Get Flit and then type:

```
$ flit install --symlink --user
```

Configure

Write a configuration file using TOML. Start with something like this:

```
extract = 'total_dyn_inst: (\d+)'
benchmarks = '../benchmarks/*.bril'

[runs.baseline]
pipeline = [
    "bril2json",
    "brili -p {args}",
]

[runs.myopt]
pipeline = [
    "bril2json",
    "myopt",
    "brili -p {args}",
]
```

The global options are:

- `extract`: A regular expression to extract the figure of merit from a given run of a given benchmark. The example above gets the simple profiling output from the Bril interpreter in `-p` mode.

- `benchmarks` (optional): A shell glob matching the benchmark files to run. You can also specify the files on the command line (see below).
- `timeout` (optional): The timeout of each benchmark run in seconds. Default of 5 seconds.

Then, define an map of *runs*, which are the different treatments you want to give to each benchmark. Each one needs a `pipeline`, which is a list of shell commands to run in a pipelined fashion on the benchmark file, which Brench will send to the first command's standard input. The first run constitutes the "golden" output; subsequent runs will need to match this output.

Run

Just give Brench your config file and it will give you results as a CSV:

```
$ brench example.toml > results.csv
```

You can also specify a list of files after the configuration file to run a specified list of benchmarks, ignoring the pre-configured glob in the configuration file.

The command has only one command-line option:

- `--jobs` or `-j`: The number of parallel jobs to run. Set to 1 to run everything sequentially. By default, Brench tries to guess an adequate number of threads to fill up your machine.

The output CSV has three columns: `benchmark`, `run`, and `result`. The latter is the value extracted from the run's standard output and standard error using the `extract` regular expression or one of these three status indicators:

- `incorrect`: The output did not match the "golden" output (from the first run).
- `timeout`: Execution took too long.
- `missing`: The `extract` regex did not match in the final pipeline stage's standard output or standard error.

To check that a run's output is "correct," Brench compares its standard output to that of the first run (`baseline` in the above example, but it's whichever run configuration comes first). The comparison is an exact string match.

Cranelift Compiler

Brilift is a ahead-of-time or just-in-time compiler from Bril to native code using the Cranelift code generator. It supports core Bril, floating point, and the memory extension.

In AOT mode, Brilift emits `.o` files and also provides a simple run-time library. By linking these together, you get a complete native executable. In JIT mode, Brilift mimics an interpreter.

Build

Brilift is a Rust project using the bril-rs library. You can build it using Cargo:

```
$ cd brilift
$ cargo run -- --help
$ cargo install --path .  # If you want the executable on your $PATH.
```

Ahead-of-Time Compilation

Provide the `brilift` executable with a Bril JSON program:

```
$ bril2json < something.bril | brilift
```

By default, Brilift produces a file `bril.o`. (You can pick your own output filename with `-o something.o`; see the full list of options below.)

A complete executable will also need our runtime library, which is in `rt.c`. There is a convenient Makefile rule to produce `rt.o`:

```
$ make rt.o
```

Then, you will want to link `rt.o` and `bril.o` to produce an executable:

```
$ cc bril.o rt.o -o myprog
```

If your Bril `@main` function takes arguments, those are now command-line arguments to the `myprog` executable.

Just-in-Time Compilation

Use the `-j` flag to compile and run the program immediately:

```
$ bril2json < something.bril | brilift -j
```

Pass any arguments to the Bril `@main` function as command-line arguments to Brilift. For example, if you have a function `@main(foo: int, bar: bool)`, you can type `brilift -j 42 true`.

Options

Type `brilift --help` to see the full list of options:

- `-j`: JIT-compile the code and run it immediately, instead of AOT-compiling an object file (the default).
- `-O [none|speed|speed_and_size]`: An optimization level, according to Cranelift. The default is `none`.
- `-v`: Enable lots of logging from the Cranelift library.
- `-d`: Dump the Cranelift IR text for debugging.

These options are only relevant in AOT mode:

- `-o <FILE>`: Place the output object file in `<FILE>` instead of `bril.o` (the default).
- `-t <TARGET>`: Specify the target triple, as interpreted by Cranelift. These triples resemble the target triples that LLVM also understands, for example. For instance, `x86_64-unknown-darwin-macho` is the triple for macOS on Intel processors.

Web Playground

Web playground is available for Bril.

Features:

- Code evaluation using the reference interpreter
- CFG visualization
- Dominator visualization
- SSA transformation

Source code

https://github.com/agentcooper/bril-playground