Scryer Prolog JIT compiler

Adrián Arroyo Calle

November 2024

Outline

$whoami

- Adrián Arroyo Calle
- aarroyoc@adrianistan.eu
- @aarroyoc@social.adrianistan.eu
- https://adrianistan.eu
- Backend developer at Telefónica (Digital Innovation)

What is a JIT compiler?

- A compiler is a program that takes code in a programming language and outputs code in another language.
    - GCC, rustc, javac, . . .
- Most of the compilers that pop in our minds when talking about the topic are AOT.
- Those compilers are run whenever we want and produce a permanent output we can use later.
- A JIT in contrast runs just before executing a piece of code, to accelerate the execution.
    - It is dictated by the flow of the program and the output they generate is not permanent.

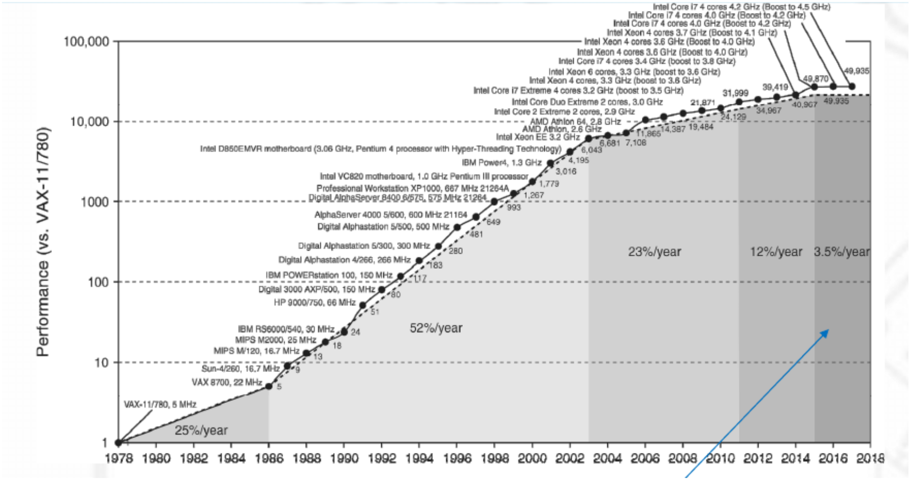
Why a JIT compiler?

- AOT compilation must deal with binary generation and ship their own runtime.
- Prolog is a very dynamic language, we need to ship a compiler too.
- A JIT compiler can live next to the interpreter, making it easier to implement.
- What about *performance*?

When a JIT compiler?

- The results of JIT compilation only live in the current session of execution.
- If a JIT compiler needs to do a compilation phase before executing it, it will only make sense if the code is going to be called several times.

- I started the JIT project as a way to increase the performance of Scryer Prolog.
- Modern computers are very fast but a traditional interpreter is not capable of exploiting all that power.



JIT compiler backend

When you implement a compiler, you can write your own backend (tons of assembly!) or you can leverage that work to a compiler backend:

- LLVM. The most popular one. Rust, Clang, Swift, Julia, ... Focus on absolute performance.
- Cranelift. Hosted by Bytecode Alliance, designed for WASM compilation in mind. Focus on fast code generation.
- QBE. Applying Pareto Principle to compiler backends. Simple but able to get more important optimizations.

Cranelift

- A JIT should focus on fast code generation.
- Cranelift is made in Rust, so no need to do complicated steps in order to integrate it into Scryer Prolog.

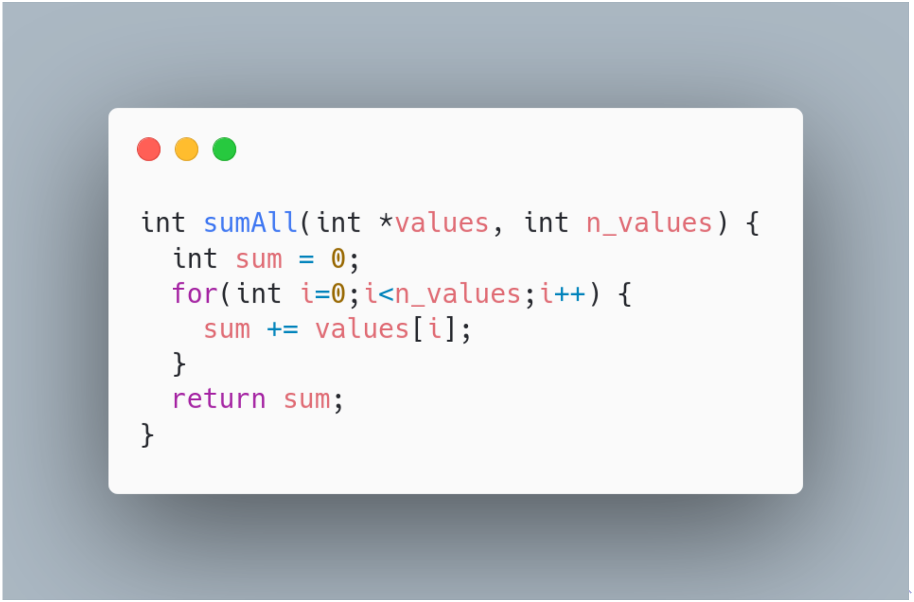
How does Cranelift work?

- We generate Cranelift IR based on our WAM code.
- Cranelift uses e-graphs and a rewriting language, *ISLE*, to find optimizations.
    - *What is an e-graph?* All optimization rewrites are kept so rules can "revert" optimizations to apply bigger ones instead.
- Generates machine code for x86-64, AArch64 (ARM), S390x and RISC-V64.

Cranelift IR

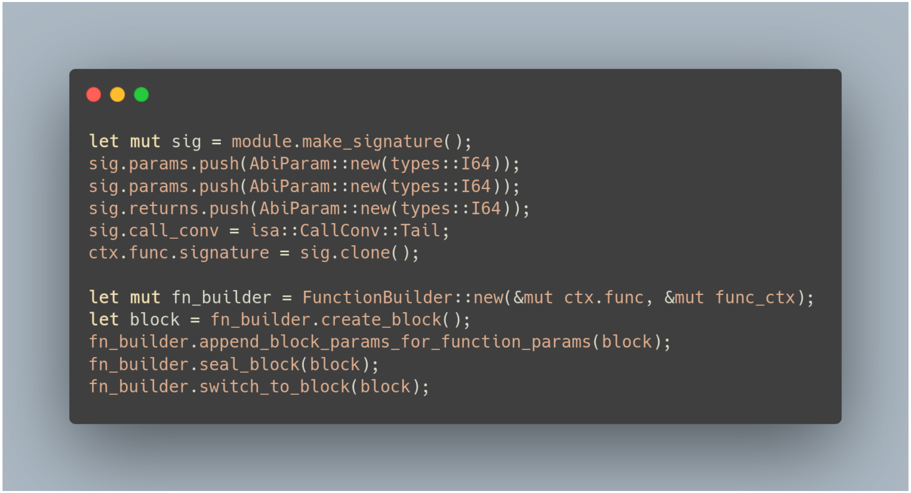
- An agnostic *assembly* language.
- We have a textual representation and a Rust DSL.
- We can define functions and basic blocks, each one with parameters
- Instead of registers we use values, which are typed and immutable (Static Single Assignment)
    - How do we do loops? We use parameters in the BB.

Cranelift IR - Function example



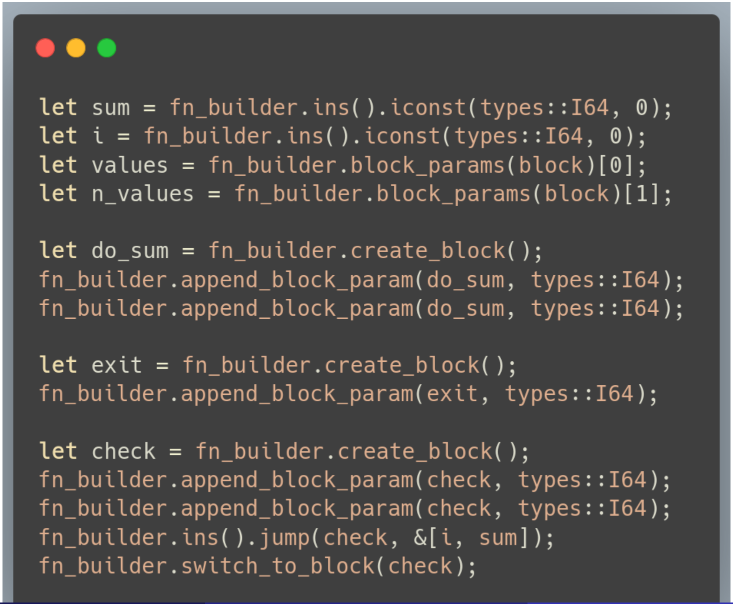
```c
int sumAll(int *values, int n_values) {
  int sum = 0;
  for(int i=0;i<n_values;i++) {
    sum += values[i];
  }
  return sum;
}
```

Cranelift IR - Rust DSL example



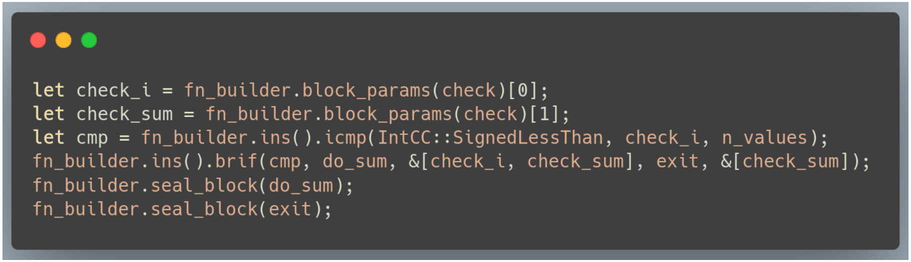
```
let mut sig = module.make_signature();
sig.params.push(AbiParam::new(types::I64));
sig.params.push(AbiParam::new(types::I64));
sig.returns.push(AbiParam::new(types::I64));
sig.call_conv = isa::CallConv::Tail;
ctx.func.signature = sig.clone();

let mut fn_builder = FunctionBuilder::new(&mut ctx.func, &mut func_ctx);
let block = fn_builder.create_block();
fn_builder.append_block_params_for_function_params(block);
fn_builder.seal_block(block);
fn_builder.switch_to_block(block);
```

Cranelift IR - Rust DSL example



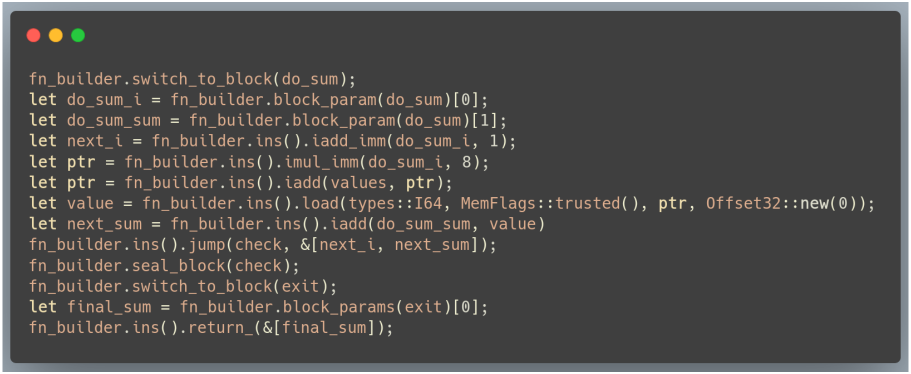
```
let sum = fn_builder.ins().iconst(types::I64, 0);
let i = fn_builder.ins().iconst(types::I64, 0);
let values = fn_builder.block_params(block)[0];
let n_values = fn_builder.block_params(block)[1];

let do_sum = fn_builder.create_block();
fn_builder.append_block_param(do_sum, types::I64);
fn_builder.append_block_param(do_sum, types::I64);

let exit = fn_builder.create_block();
fn_builder.append_block_param(exit, types::I64);

let check = fn_builder.create_block();
fn_builder.append_block_param(check, types::I64);
fn_builder.append_block_param(check, types::I64);
fn_builder.ins().jump(check, &[i, sum]);
fn_builder.switch_to_block(check);
```

Cranelift IR - Rust DSL example



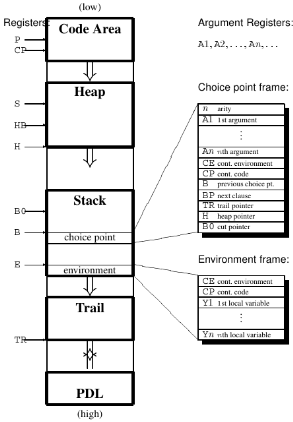
```
let check_i = fn_builder.block_params(check)[0];
let check_sum = fn_builder.block_params(check)[1];
let cmp = fn_builder.ins().icmp(IntCC::SignedLessThan, check_i, n_values);
fn_builder.ins().brif(cmp, do_sum, &[check_i, check_sum], exit, &[check_sum]);
fn_builder.seal_block(do_sum);
fn_builder.seal_block(exit);
```

Cranelift IR - Rust DSL example



```
fn_builder.switch_to_block(do_sum);
let do_sum_i = fn_builder.block_param(do_sum)[0];
let do_sum_sum = fn_builder.block_param(do_sum)[1];
let next_i = fn_builder.ins().iadd_imm(do_sum_i, 1);
let ptr = fn_builder.ins().imul_imm(do_sum_i, 8);
let ptr = fn_builder.ins().iadd(values, ptr);
let value = fn_builder.ins().load(types::I64, MemFlags::trusted(), ptr, Offset32::new(0));
let next_sum = fn_builder.ins().iadd(do_sum_sum, value)
fn_builder.ins().jump(check, &[next_i, next_sum]);
fn_builder.seal_block(check);
fn_builder.switch_to_block(exit);
let final_sum = fn_builder.block_params(exit)[0];
fn_builder.ins().return_(&[final_sum]);
```

Data structures



- WAM uses several data structures in its design.
- Two approaches:
  - Share as much as possible the data structures between the compiled code and the interpreter
  - Develop separate data structures for the compiled code

First approach

Pros

- Easier to get started and get some results.
- We do not need to copy data back and forth.

Cons

- Code becomes too complex, we need to interact with Rust code that it's not FFI safe.
- Memory can be reallocated, to solve that we need to pass some addresses via registers, and work with offsets.
- We can't use all the optimizations of Cranelift.

Second approach

Pros

- We can leverage Cranelift to do more optimizations.
- Less complex code.

Cons

- We need to implement lots of things from scratch.
- We need to copy data before and after entering the compiled section.

How to start

- I decided to start implementing the JIT by following the WAM Book by *Hassan Aït-Kaci*.
- At first I tried the first approach (sharing everything we can)
- Later, I started from scratch with the second approach
- We'll talk about the second approach.

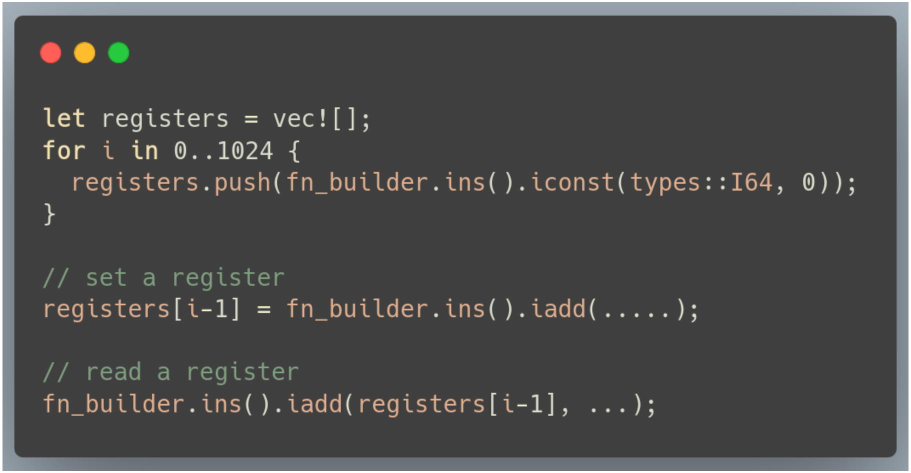
Code

- We take the WAM instructions for a predicate in order as a list.
- For each instruction, we emit code that implements the instruction.
- Every instruction implementation is independent of other instructions, but we have some compilation-time data structures to implement some behaviors.

The registers

- How to we map the WAM registers to Cranelift?
- We use an array of SSA values!
- Instructions modify the contents of the array, adding or removing new SSA variables
    - In WAM we just refer to the registers value with their index number
    - Remember SSA values are immutable, but we can use another one in the same index position of the array.

Registers example



```
let registers = vec![];
for i in 0..1024 {
  registers.push(fn_builder.ins().iconst(types::I64, 0));
}

// set a register
registers[i-1] = fn_builder.ins().iadd(.....);

// read a register
fn_builder.ins().iadd(registers[i-1], ...);
```

Heap

- Heap in Scryer Prolog in v0.9.4 is just a Rust vector
- We need to be able to push it, consult the length and obtain a pointer to the underlying array.
- We can use external functions in our assembly to do that.
- We share the heap between interpreted and native code.

store, deref and bind

- Basic operations, they're used in multiple WAM instructions.
- They're implemented in assembly, inlined (multiple store invocations result in repeated code).
- They're very common, so we try to reduce the overhead of calling them.

Unification instructions

- Unification instructions set a MODE and a READ pointer (to be used in READ mode).
- We use Cranelift *vars* for it
  - It's an abstraction over SSA, that works like a mutable variable.
  - We define them at the beginning of a predicate
  - Each instruction can read and set them
- And unification is also inlined!

Calls

- Predicate calls are mapped to function calls, argument registers being the input arguments of a Cranelift function.
- When the function starts, it copies the input arguments to the register array in the correct position.
    - This doesn't mean we're doing redudant moves. Thanks to the SSA abstraction, we're only moving the SSA values to help us generate better instructions, but instruction generation only outputs moves that are required.
- Must be defined before usage. No support for recursive calls yet.
- No P / CP logic. Everything is implemented with function calls.

Execute

- A special type of call in WAM is Execute, which is tail-optimized.
- In some cases, we can use $return_{call}$ instruction from Cranelift.
- But if there's backtracking, we can't use it :/

Proceed

- It's a no-op because the whole P / CP logic is implemented with function calls.

Stack

- Let's introduce first a stack without backtracking.
- When we are writing the instructions, we already know all the code of a predicate.
  - Allocation instructions are static, the size of them don't change in runtime.
  - We can know how much stack is going to be used. Or at least give a maximum stack size.

Stack II

- One idea we can try is: do the same as registers, making an array of the maximum stack size full of SSA values.
- Do one allocation per predicate. The stack values will be dropped by Cranelift as soon as their contents are not needed.
    - But it isn't really a stack in the sense of location. The compiler has full control over where are really stored.
- Allocations/Deallocations could be considered no-op.
- Values passed to another predicates are just moved like normal SSA values.
    - Because in the stack we will only store pointers to the heap and constants.

But what about StackVars

- Stack Vars are an optimization proposed by the WAM book!
- It saves space by making the stack also a place to store vars content, like a mini-heap.
- I argue this optimization doesn't make sense anymore:
    - It was proposed in an era where memory was much more limited than today. Since then, memory has become larger and larger. They advanced more than CPUs.
    - It was proposed in a book without garbage collector, so heap space is never reclaimed, stack space can.
    - It adds many checks in very common operations like deref, bind, unify, . . .
    - If we add Stack Vars, we can't use an array of SSA values as our implementation, we must create an actual data structure for it.
        - We lose compiler optimizations in the middle.

Backtraking

- In the WAM book, the stack also holds the backtracking information. Here we want to keep the stack as defined by now.
- But we also have another problem.
- Backtracking in WAM is implemented doing code jumps.
  - Every instruction has an index and we're able to jump an any code position by just knowing its index.
  - This is not allowed in Cranelift: only functions and blocks.
  - What the WAM is doing is a *continuation*, jumps from one place to another restoring all the local data in the moment of the choicepoint.
  - There's a solution: unrolling the continuation.
  - Make the flow of the program using call/return the flow of the backtracking.
  - The local data will be preserved by following the flow and keeping a trail data structure.

Backtracking Example

```
100 wam_do,
102 choicepoint_or_go(105),
103 unify_this, # if it fails, it goes to line 105
104 exit,
105 restore_local_data,
106 unify_that,
107 exit
```

Backtracking Example II

```
wam_do,
push_continuation_point(B),
unify_this, #if it fails, it jumps to B, which is defined
 later, but the compiler sees it in the
 same compilation unit
exit,
set_continuation_point(B), # here we set the block
 destination and we can reset variables
unify_that,
exit
```

Backtracking in short

- Continuation points in a predicate are managed using an auxiliary vector, just in compile time.
- Failable predicates have been modified to insert a jump to the next continuation point if available (the vector has at least one continuation point available)
- If there are no continuation points in the vector, we just insert a return so we can go back to the previous function.
- On a call / execute we need to analyze the return status, so that if it's false we do the same (insert a jump to the next continuation if possible, otherwise insert a return).

Trail

- In the WAM, the trail records data that needs to be reset on backtracking.
- Unlike the stack, we need to keep it, as what goes inside is determined at runtime.
- Trail operations are also implemented in CLIF.

What else?

- Other work includes making the interface between interpreter and native code
  - We need trampolines, as we change the calling conventions.
  - Calling system calls from native code is not yet done, but will be implemented together with a new Plugin interface
- Implement the actual instructions!
  - It's easier than it seems, if you already have the blocks (store, deref, unify, bind, . . .)

the end?

- Questions?