Un

Luc Alexander Lüthi

December 28th, 2025

Contents

1 Architecture

I have difficulty calling Un a programming language, realistically its more like a compiler backend. It represent the internal content of a compiler. It forsakes parsing as an interface and allows you to focus on that independently. The portion of language un cares exclusively about is semantics, allowing full control over the target output while allowing the creation of any expression calculus above it as desired. The current target is an interpreted virtual machine for the sake of prototyping and simplicity, but the eventual target for all metacomputation tooling under my research will be real target instruction set architectures. This same virtual machine is present in its environment based compile time execution feature. The system comprises of 3 layers of reasoning. A high level compilation only substrate meant for reasoning about machine instructions, and intermediate representation meant for ergonomic code writing, and a target instruction set, which can really be anything as long as the intermediate representation generalized well to it after normalization.

2 Substrate

The compilation only substrate uses S expressions to express abstraction, taking a page out of the book of Lisp (See the state of the system year 1 note to understand why I hesitate to call Un a Lisp). I have experienced enough ergonomic and performance based friction to know not to pay too much effort into making a system minimal and pure. It has the features it needs. The interface is as expected, (x a b) represents a list. When invoked, the "procedure" x is recalled with parameters a and b. The head of the list, x, is evaluated before any arguments if argument or head evaluation is needed, and then the expression is evaluated once normal. Lists are otherwise simply structural groupings of symbols. The tokenizer for un does not distinguish between traditional identifiers and symbolic operator identifiers. Everything is infix afterall. The only real distinction happens with string content, and with numeric content which is always preferred if a hexidecimal numeral can be derived from the token.

The *bind* special form names a token. This is a largely unified interface and it can be used to define constants, compile time procedures (completing the macro interface in two parts: naming, and execution), and normal substrate procedures. It takes a name, a list of arguments, and a body expression.

```
(bind add (a b) (+ a b))
(bind seven () 7)
(bind compute_seven ()
    (comp default (add 1 6)))
```

The *comp* special form subsequently allows the procedure it defines to be computed at compile time, it expects a token for defining which virtual machine to run the execution in, and an expression which is expected to either normalize and compile fully to bytecode, or result in a substrate structural component which can be reduced at compile time via standard procedure execution rules. Arguments in procedures are distributed before execution or simplification of any kind, and as comp binds are simply considered standard bind procedures with bodies contianing a comp expression, all arguments are symbolically expanded to full list structures before compilation begins for that virtual machine. The memory of any given virtual machine is persistent between comp invocations, and can be used to store and reason about compiler internals during program runtime.

It is worth noting here that the bytecode/intermediate representation have full reification down and full reflection up on lists and buffers. When the the list (a b) is passed as an argument that is loaded into a register as such

```
(bind f (x)
    (comp default (
        (reg a)
        (mov a x)
        (a))))
(f (a b))
```

The tokens *a* and *b* are written to a buffer in the ephemeral static memory buffer at the beginning of the virtual machine in question with a word sized length at the beginning of the buffer. The symbolic value x in the evaluation of f is converted to an address to that buffer. When returned from the procedure (a), the outer calling code receives an address to a buffer which it reconstructs into a substsrate level list using the information it has about symbols, strings, and literals. This works with any form of value valid in either the substrate or the IR, and works with nested lists as well.

The *use* special form allows the aggregation of binds from a separate un program file into the program memory of the current compilation process, and will return the primary expression in that file as its result. Wrapping use in a comp evaluator will allow that file to be compile time evaluated.

```
(use "std.un")

(bind main () (
    (reg x)
    (reg y)
    (mov x (flat (arena ff ffff)))
    (comment (mov y (flat (alloc x 80))))
    (mov x 0)
    (int x)
))

(main)
```

The *flat* special form flattens a list into an outer scope. Ive had some trouble with getting these to compose exactly how I want them to in the past, but I believe everything is in order now.

```
(f (flat (a b))) -> (f a b)
```

The final substrate level special form is *uid*, which allows the generation of a series of aliases for its subsequent block, these aliases are unique tokens and exist only for the given expression.

```
(bind arena (adr size) (
    (uid (x) (
        (reg x)
        (mov x adr)
        (psh x)
        (psh x)
        (mov x size)
        (add x x adr)
        (psh x)
        (mov x sp)
        (x)
    ))
))
```

3 IR

An intermediate representation block can include a series of instructions and optionally a token to return. This allows you to nest blocks within instructions for preevaluation, hoisting the block above its invocation and using the resulting token in place of the block itself. This representation is meant to be as generalizable to real architectures as possible without compromising efficiency. It features a small set of standard opcodes and a labeling scheme. There are three opcodes which exist in the IR which dont exist in the bytecode itself. *reg* defines that the symbolic argument is to be considered a unique register until it is overwritten. *label* defines a token to consider a valid label, until it is overwritten. *reif* manually reifies an external argument and places the buffer in ephemeral static memory. The register token provided is then given the full haalf word address to that buffer.

```
(bind alloc (arena size) (
    (uid (ptr val bound end) (
        (reg ptr)
        (reg val)
        (reg bound)
        (add ptr arena 8)
        (mov val (at ptr))
        (add val val size)
        (add bound arena 10)
        (mov bound (at bound))
        (cmp val bound)
        (jlt end)
        (mov val 0)
        (label end)
        (mov (at ptr) val)
        (val)
    ))
))
```

In truth I haven't had the opportunity to write a lot of un yet, so I dont have an exorbitant amount of code examples quite yet.It is also worth noting that currently register coloring is broken, and there is a chance that writing to a register token does not accurately persist the change if enough other registers are mutated between initial mutation and usage. Thats next on the chopping block.

4 Bytecode

The bytecode is very simple, and im still implementing some of the interrupts for things like forking to a new core, user input, etc. That being said, here is the final bytecode pure instruction set:

```
mov,
add, uadd,
sub, usub,
mul, umul,
div, udiv,
mod, umod,
shl, shr,
and, or, xor,
not, com,
cmp, jmp,
jne, jeq,
jlt, jgt,
jle, jge,
psh, pop,
call, ret
```

Jumps and calls require an explicit relative address. This means you cannot pass a label into a register and pass the register to a jump or a call. There is one intrinsic in the language, (at register) which allows dereferencing in mov instruction arguments. Instructions are 4 bytes wide, mov allows 2 byte literals but alu operations allow only 1 byte literals.