Native Law Judgements

Benjamin Summers

February 19, 2026

Abstract

This document describes the design and implementation of the default function evaluation engine of the native PLAN runtime system.

1 Introduction

In order to achieve a system which truly has zero dependencies, our PLAN runtime system is written in assembly. This makes the code quite expensive to maintain. Worse, the runtime system is designed to have uptimes in the decades, which makes it impossible to do a live upgrade of runtime system itself. This constraint adds a number of difficult and unconventional engineering challenges.

Our high level approach to these challenges is to keep the runtime system as small as possible, and to instead move as much code into XPLAN, which can be written in a much higher-level language and which can be upgraded dynamically.

One of the most significant subsystems which we move into XPLAN is the actual optimization logic used to run functions. However, a number of non-trivial tasks must still be possible using only the basic evaluation system:

1. We need to be able to run a significant body of code in order to test the runtime system itself.
2. The optimization engine will be written in source code in a high-level language. We will need to be able to load a compiler and use it to build that optimization engine.
3. The compiler for this language will also be written in a high level language, so we will need to be able to run it against itself for bootstrapping.
4. In order to produce the actual seed-file that we need for bootstrapping, we need to be able to serialize the bootstrapping compiler.

This tension between the need to minimize mechanical complexity, and the need to achieve a usable performance baseline results in a somewhat unique implementation strategy, which we will document in detail in the rest of this document.

2 Lazy Graph Reduction

PLAN is a lazy graph reduction engine built using super-combinators. A system like this, in theory, doesn't do any actual computation when a function is called. Functions just work something like a template expansion, and the graph reduction engine does all of the actual work.

Here's a concrete example:

```
foo :: Bool -> Int -> Int -> Int
foo x y z = if x then y+y else z+z
```

If `foo` is invoked as `(foo True 3 4)`, the actual result of running the function is not 6, but tree of thunks, which is then processed by the graph reduction engine in order to find a result:

```
(((foo True) 3) 4)
((foo[True] 3) 4)
(foo[True 3] 4)
(((If True) ((Add 3) 3)) ((Add 4) 4)) -- expansion of foo
((If[True] ((Add 3) 3)) ((Add 4) 4))
(If[True ((Add 3) 3)] ((Add 4) 4))
((Add 3) 3)                           -- expansion of If
(Add[3] 3)
6                                     -- expansion of Add
```

Producing and consuming all of these thunks is highly inefficient, so sophisticated runtime systems figure out which subset of the work will always be performed, and just do all of that eagerly. But this requires somewhat sophisticated analysis and optimization work.

In order to keep the native runtime system simple, we avoid all of this work and instead adopt the naive template-expansion approach. But, in order to produce something with a usable base-line of performance, we also perform a number of rote optimizations which only require trivial analysis.

3 Optimizing Template Expansion

Here's our example again:

```
= (mapMaybe f o)
| If (Nil o) 0
| 0 (f (Ix 0 o))
```

Ignoring the issues around needing to make everything legible to GC, the most stupid possible approach to template expansion would look something like this:

```
Obj mapMaybe(Obj f, Obj o) {
    Obj result =
        App(App(App(If, App(Nil,o)),
                0),
            App(0,
                App(f,
                    App(App(Ix,0),
                        o))));
    tailcall tmp.execute()
}
```

This approach requires a very large number of small thunks to be allocated, and each of those thunks must be separately processed by the graph reduction engine.

To improve on this, we use a series of improvements on this basic output in order to get much better performance.

3.1 Optimization 1: Bigger Thunks

We can do the same thing with fewer allocations, and make life a bit easier for the graph reduction engine by constructing bigger thunks instead of only use Apps.

```
Obj mapMaybe(Obj f, Obj o) {
    Obj result =
        App3(If, App(Nil,o),
                 0,
                 App(0, App(f, App2(Ix, 0, o))))
    tailcall tmp.execute()
}
```

3.2 Optimization 2: Direct Closure Construction

Notice that the `(0 (f (Ix 0 o)))` expression is always undersaturated, like an expression like `(Add 3)` would be. So the result will always be a closure.

Producing a thunk in this case results in a lot of wasted work. The thunk will examine the head, determine that the call is undersaturated, allocate a closure, and update the thunk to point to the closure.

As long as the function call is to a known constant, it is trivial to determine that the call is undersaturated, which allows us to skip constructing this thunk, and instead directly construct a closure.

```
Obj mapMaybe(Obj f, Obj o) {
    Obj result =
        App3(If, App(Nil,o),
                 0,
                 Clz1(0, App(f, App2(Ix, 0, o))))
    tailcall tmp.execute()
}
```

3.3 Optimization 3: Single Allocation

Because the template expansion for each function is always the exact same shape, we can avoid doing many small allocations, and instead just do one big allocation.

Not only does this save work by avoiding repeated allocation, but it also means that the work of actually filling in the template does not have to make sure to make all of it's pointers available to the garbage collector. No GC will happen during this initialization process.

```
Obj mapMaybe(Obj f, Obj o) {
    save(f, o);
    void *buf = alloc(22);
    restore(f, o);

    fillapp1(&buf[0], Nil, o);        // a = (Nil o)     [weight=4]
    fillapp2(&buf[4], Ix, 0, o);      // b = (Ix 0 o)    [weight=5]
    fillapp1(&buf[9], f, b);          // c = (f b)       [weight=4]
    fillclz1(&buf[13], 0, c);         // d = (0 c)       [weight=3]
    fillapp4(&buf[16], If, a, 0, d);  // e = (If a 0 d) [weight=6]

    Obj *result = (Obj*) &result[17]; // 17 b/c first word is GC header
    tailcall result.execute()
}
```

3.4 Optimization 4: Precomputed Template

Not only is the size of the resulting graph constant, but so is it's structure. If we pre-compute the shape of the template, we can simply copy the whole thing to our reserved space on the heap.

There are only a few changes that need to be made to this template after it is copied:

- Pointers between nodes internal to the graph need to be updated to point to the new node locations.

- Reference to variables must be filled in.

- Because references to constants (except direct nats) can be moved around by GC, we need to maintain a table of constants, and fill each of these in by replacing them with a reference into this table.

```
Obj mapMaybe(Obj f, Obj o) {
    save(f, o);
    void *buf = alloc(22);
    restore(f, o);

    memcpy(template, buf);

    buf[2]   = Nil; // from constants array
    buf[3]   = o;   // from register
    buf[5]   = Ix;  // from constants array
    buf[7]   = o;   // from register
    buf[10]  = f;   // from register
    buf[11] += buf;  // convert offset to pointer
    buf[15] += buf;  // convert offset to pointer
    buf[17]  = If;   // from constants array
    buf[18] += buf;  // a: convert offset to pointer
    buf[20] += buf;  // d: convert offset to pointer

    Obj *result = (Obj*) &result[17]; // 17 b/c first word is GC header
    tailcall result.execute()
}
```

3.5 Optimization 5: Code Generation

As you can see from the previous example, the actual template-filling logic is just a straight-line sequece of simple memory operations, and none of this needs to be interleaved with garbage collection.

As a result, it is quite simple to just generate this machine code for each law at law construction time.

In particular, we only need to be able to construct 4 assembly expressions. If constants=r11 and output=r12, then:

1. Load a constant: `mov r10, [r12+o]`
2. Load from stack: `mov r10, [r15+o]`
3. Write a register: `mov [r14+o], reg`, (where reg can be: rax, r10, rdi, rsi, rdx, rcx, r8, or r9).
4. Write a argument: `mov [r14+o], rdi`
5. Hydrate an internal: `add [r14+o], r14`

If we use 32-bit offsets for each output instead of trying to minimize the size, we can use a simple 7-byte output for each instruction, which allows a simple table to be used instead of complex encoding logic.

```
Registers:

    r15             = stack
    r14, r13        = reserved (heap registers)
    r12             = temp
    r11             = constants
    rax             = self reference
    rdi,rsi..,r12 = arguments

Patterns:

    mov r12, [r11+x] ; 4D8BA3xxxxxxxx
    mov r12, [r15+x] ; 4D8BA7xxxxxxxx
    mov [r14+x], rax ; 498986xxxxxxxx
    mov [r14+x], rdi ; 4989BExxxxxxxx
    mov [r14+x], rsi ; 4989B6xxxxxxxx
    mov [r14+x], rdx ; 498996xxxxxxxx
    mov [r14+x], rcx ; 49898Exxxxxxxx
    mov [r14+x], r8  ; 4D8986xxxxxxxx
    mov [r14+x], r9  ; 4D898Exxxxxxxx
    mov [r14+x], r10 ; 4D8996xxxxxxxx
    mov [r14+x], r11 ; 4D899Exxxxxxxx
    mov [r14+x], r12 ; 4D89A6xxxxxxxx
    add [r14+x], r14 ; 4D01B6xxxxxxxx
```

TODO: figure out how to encode the prelude. If we store the executable code in a big nat outside of the moving heap, then we actually can do the allocation directly, if that ends up being easier.

```
judgement_prelude:
    if (r14+sz >= r13)
        gc(sz)
    r11 = self.metadata.template
    memcpy(r14, r11, sz*8)
    r11 = self.metadata.constants
    [[rest]]
```

TODO: figure out how to encode the tail logic:

```
hp += 22
Obj *result = (Obj*) &result[17];
tailcall result.execute()
```

The tail logic should probably just be a tail call into a constants function.

```
tail_logic:
    hp += rdi
    result = &heap[rsi];
    tailcall result.execute()
```

3.6 Optimization 6: Avoid Thunk Update

Loops in function programming languages are encoded using tail recursion, which makes tail recursion optimization an essential feature of any purely functional programming language.

Tail call optimization is a bit different in a minimalist, purely functional language like PLAN, because things like `if` and `case` and just normal functions.

Lazy evaluation essentially works by repeatly simplifying the outer expression until it is "done" and then repeating the process for the inner expressions. This iterative process automatically provides constant-space tail recursion except for one tricky-edge case: thunk caching.

After each thunk is evaluated, it is modified so that subsequent references return the pre-computed value, instead of running the computation again. But, in order to do this modification, we need to run a piece of logic after the evaluation. This requires a stack frame, and breaks tail recursion.

The problem is that we end up with a long chain of tiny stack frame which just update thunks. If we could find some way to use an alternate thunk structure which doesn't perform the update, all of these tiny stack frames would go away.

Consider the following example:

```
= (sillyAdd a b)
| Seq a
| Ifz b a
| sillyAdd Inc-a Dec-b

= (sillyDouble a)
| sillyAdd a a
```

When sillyDouble is run, we know that the call to sillyAdd will always be evaluated immediately, so the thunk will definitely only be evaluated once. In this case, we can trivially use a non-updating thunk and recover tail recursion.

In 'sillyAdd', that means that (Seq a ..) doesn't need to update, but what about (Ifz b a ..)? Well, we know that Seq evaluates each argument exactly once, so any expression passed as an argument to 'Seq' does not need a thunk update. And the same with 'Ifz'. Ifz will evaluate each of it's arguments at most 1 time, so there is no need for the thunk update.

This chain of reasoning eliminates the thunk updates for the whole chain of calls leading up the recursion, and the result is that we recover tail call optimization.

Doing this analysis in general requires a significant amount of work. However, in XPLAN, things like Seq and Ifz are treated as primitives, so we can simply hard-code this information on functions which are simple wrappers for primops (see the later section on recognizing primop wrappers).

Once this information is available, the process is trivial. Whenever we see a saturated call to a function, remember which arguments are evaluated at most once. When generating a thunk for such an argument, use the non-updating version.

4 Template Layout

First of all, we special case trivial functions which just return an argument or a constant value. These functions don't require any allocations at all.

Computing the template is fairly easy, we just need to figure out what all of the nodes are and where they belong in the tempate. There are only a few cases:

1. A saturated or unknown function call will be a thunk, and will require args+3 words.
2. An undersaturated function call will be a closure, and that requires args+2 words.
3. A constants value or a variable reference requires no space in the template, just a reference from the containing expression.

Because of support for recursive let, the template is a graph, not a tree. However, laying out the template remains easy, since the size of each expression is static. We just need to compute the size of each binding, and we can use that to deterimne where all of the let-references should point to.

There is one edge-case with let bindings, which is that each binding needs to have an allocation in the resulting template which can be pointed to, but constants as trivial aliases don't have that. This can be solved by simply creating a dummy thunk in that case (using the `x_var` executioner).

Before creating the template and code, we will need to know the sizes and types for each top-level expressions (let-binding or body), as well as the total space needed for the code.

```
int cSz = 0
int xSz = 0

expr :: Expr -> ()
expr e = case recognize e of
    RegRef{}      -> cSz++
    StackRef{}    -> cSz+=2
    Direct{}      -> ()
    Indirect{}    -> cSz+=2
    Closure{f,xs} -> xSz += (2 + xs.len); go(f, xs..)
    Thunk{f,xs}   -> xSz += (3 + xs.len); go(f, xs..)

bindShape :: Expr -> Type
bindShape e =
    expr e
    case recognize e of
        Closure{f,xs} -> ClosureType{tag=f, size=Sz(xs)}
        Thunk{f,xs}   -> ThunkType{}
        _             -> xSz+=3; ThunkType{}

Then use this to get an array of sizes and types, once per top-level
expression.

size e = case recognize e of
```

Once we have calculuated the size of each binding, and the body, we layout the template using the following algorithm:

At each point, we need to know:

- Where will the reference to this result be written?

- Where is the end of the currently-written section?

- Where is the end of the currently-written commands section?

If the expression is a constant: If it's a direct atom: Write the atom as the reference. Otherwise: Add the constant to the references set. Emit a command to load the constant. Emit a command to write the constant to the output.

If the expression is a reference: If the reference is an argument by register: Emit a command to write the register. If the reference is an argument by stack: Emit a command to load from the stack. Emit a command to write the value. If the reference is to a let-binding. Determine which type the binding holds. Write a tagged offset to the parameter Emit a command to hydrate the slot

If the expression is an application: Collect all of the parameters.

If it's undersaturated, write a closure.

The size will be args+2. Reserve this much space in the output.

Write the GC header.

Set the multiplicity word to 0 (all parameters are assumed to be shared).

Write the function + arguments via recursive call passing in the appropriate slot as the output parameter.

Otherwise, write a thunk.

The size will be args+3. Reserve this much space in the output.

Write the GC header.

Write the executioner.

```
If the lowest mutipliciity bit is 1, it's
\lstinliner|eval_unknown_noupdate|, otherwise it's
\lstinline|eval_unknown|.
```

Set the multiplicity word to 0.

Write the function (with a multiplicity word of 0).

If the call is to a known function, load the multiplicity word from the metadata.

Write all arguments via recursive calls.

After each write, shift the multiplicity word right by one.

5 Notes on Lazy Laws

The newest version of PLAN does not force pins and laws on construction, so they can contain thunks. However, the whole *expression* is evaluated, and all direct constant references are evaluated two WHNF. This provides enough data to be able to do law codegen at law construction time.

Doing codegen at law construction time also means that we can be sure to always have the multiplicity analysis available for all of the laws directly called within the law body.

6 Recognizing Primops

By convention, all invocations of primitives are wrapped using a pinned law:

(Pin x)=(<0> (0 x))

These wrappers all have a predictable shape which can be easily recognized.

n 1 [<0> [[0] 1]] n 2 [<0> [[[0] 1] 2]] n 2 [<0> [[[[0] 1] 2] 3]]

Or:

n 1 [<0> [7 1]] n 2 [<0> [[7 1] 2]] n 2 [<0> [[[7 1] 2] 3]]

Whenever we find a valid wrapper for a primitive, we assign it's judgement to be the code for that primitive, and we record a metadata word which indicates which arguments are evaluated only once.

For example:

```
(Add a b)=(<1> (2 a b))

Add = <{"Add" 2 [<1> [[[2] 1] 2]}>
```

This will be recognized as a wrapper by looking up this record in a table:

```
{module=1, opcode=1, arity=2}
```

The function pointer will then be assigned as a direct pointer to the Inc primitive, and it's multiplicity will be 1 (the first argument is unshared).

If you were to instead invoke this primop by hand, without the wrapper, the generic primop machinery will be run, which will do:

- Evaluate the argument. - Get the argument head and size. - lookup this information in the table. - Load all of the arguments into registers+stack. - Call the approriate primitives.

This is much more work, and the runtime will not be able to infer the multiplicity information.

7 Recording Multiplicity Information

We can encode multiplicity information as a simple bit-mask. The lowest bit being set indicates that the first argument is unshared, etc.

However, where in the law allocation box should this information be stored?

8 Storing the Code

We can construct a NAT which stores the execution logic for a law, and simply store it in one of the metadata slots for a LAW.

However, we do need to allocate once in each routine, and we don't want the code to move around from underneath us, so we need some way to store this outside of the GC heap.

Fortunately, I have been exploring ideas for moving large nats off-heap anyways.

I believe that this can be achieved by adding a mark bit to the GC header, and adding a bit to the NAT tag which indicates whether or not it lives in a finalized pin or not (second or third GC generation).

Each thread will need to keep a table of large NATs (probably as a backwards linked list stored in normal GC memory). Whenever we see a reference to a local bignat, we must mark it. And, at the end of GC, we need to traverse the list of known bignats and free each one which has not been marked.

Erlang, like many other systems, does something like this, because otherwise your big nats need to be copied around on every GC.

This system would also mean that the code that we generate for laws would not move around during GC, which solves a lot of problems.

TODO: should the actual code-pointer point directly into this NAT. Is that safe?

9 Storing the Constants Table

The code will also need access to a table of constants, and this will need to be placed in the law somewhere as well.

I suppose we can do this all with a single metadata slot:

```
0[multiplicity template-and-code constant1 constant2..]
```

Where the code NAT directly includes the output template as a constant.