

The Design of the PrimOp Calling Convention of PLAN

Benjamin Summers

September 12, 2025

Abstract

This describes the primop calling convention and provides an in-depth rational for the design.

1 Introduction

PLAN uses the following calling convention for primitive operations. This document will discuss why this design was chosen, and what important properties it has, which other options failed to provide.

```
(<0> (0 i))      -> mk_pin(i)
(<0> (1 n a b))  -> mk_law(n,a,b)
(<0> (2 p l a z m o)) -> plan_case(p,l,a,z,m,o)
```

This convention replaces the more obvious convention used in earlier designs.

```
(<0> i)          -> mk_pin(i)
(<1> n a b)      -> mk_law(n,a,b)
(<2> p l a z m o) -> plan_case(p,l,a,z,m,o)
```

One advantage of the new design is that it makes the arity logic simpler. Before, the arity of pinned nats was 3 for <1>, 6 for <2>, and 1 for everything else. With the new logic, the arity is always 1.

But more importantly, this design is forced by two core constraints:

1. The need to have a shared runtime with a common semantics between PLAN and XPLAN.
2. The need to enable some basic optimizations when compiling laws which may not yet have been fully normalized (may contain references to lazy data).

2 XPLAN and PLAN

A PLAN runtime is actually a runtime for two different languages at the same time: PLAN and XPLAN. PLAN is a lazy, purely functional, frozen language with a tiny set of built-in operations. XPLAN is an impure functional language with a much larger, growing set of built-in operations.

Extending the obvious calling convention to include XPLAN operations would require changing the arity:

```
(<3> x)  = inc(x)
(<4> x)  = dec(x)
(<5> x y) = add(x,y)
```

This creates fundamental incompatibilities between PLAN and XPLAN.

- A single execution engine cannot be used for both languages since, the graph reduction engine and an optimizing law compiler needs to be able to determine if an application is saturated or not, and this needs to know the arity.
- The universe of possible evaluated values changes. For example, the following value can exist in XPLAN, but not in PLAN: (<5> 1). Enforcing that such values never leak between contexts is not realistic.

Switching to the new convention eliminates all of the essential differences between PLAN and XPLAN. PLAN is just an execution mode of XPLAN where unlawful operations crash instead of being executed.

3 Optimizing Laws

Another option that was considered, which avoids constructing and ADT for each primop call, was the following:

```
ARITY(<i;0>) = i+1

(<1> 0 i)      = MkPin(i)
(<3> 1 n a b)  = MkLaw(n,a,b)
(<6> 2 p l a z m o) = VCase(p,l,a,z,m,o)
```

This avoids the incompatibility between XPLAN and PLAN by using the pinned nat to indicate the arity, and then having the first argument identify the operation.

However, this interacts very poorly with support for laziness within pins and laws.

In order to demonstrate this, let's look at the code for a wrapper function which calls one of these primops.

```
(Pin x)=(<1> 0 x)

Pin = <{"Pin" 1 [<1>[0] 1]}>
```

The important thing to note here is that the head of the body was collapsed into the constant closure <1>[0]. This is smaller and faster than encoding it as [[<1> [0]] 1].

The problem is that laws only guarantee that the expression itself is evaluated, including evaluating constants to WHNF. So, the law compiler cannot assume that the closure parameter is available for inspection, it could very well be a thunk:

```
<1>[thunk]
```

The result is that we cannot reliably recognize calls to primops, which destroys a number of very important optimizations.

In contrast, let's look at the law expression within the new representation:

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

Pin = <{"Pin" 1 [<0> [[0] 1]]}>
```

As you can see, the pinned number, the constructor tag, and the arity of the call are always available for such analysis.

4 Additional Benefits

There are some other slightly useful properties which fall out of this design. Let's start by noting that the the new convention is essentially just an uncurried version of the old one:

```
(<0> i)      -> (<0> (0 i))
(<1> n a b)  -> (<1> (0 n a b))
(<2> p l a z m o) -> (<2> (0 p l a z m o))
```

But because PLAN represents arrays using partially applied numbers, all of our arrays include a zero tag, which is not used for anything. Instead, we can take advantage of this tag to group calls into "modules".

```
(<0> i)      -> (<0> (0 i))
(<1> n a b)  -> (<0> (1 n a b))
(<2> p l a z m o) -> (<0> (2 p l a z m o))
```

Where <0> is used for the primitive PLAN values, <1> is used for direct invocations of jets in XPLAN, <2> is used for XPLAN effects, etc.

This has some nice advantages:

- Many fewer pinned nats. Every pinned nat requires a whole pin on the heap, which is a small but pointless waste of resources.
- Each type of operation has its own "namespace". We can add new XPLAN effects without needing to renumber other types of effects.

5 Avoiding Runtime Overhead

The new convention is notably less efficient. Instead of simply running the operation, every call to a primop must now allocate an ADT, and the actual implementation of the primop must inspect this ADT, handle edge-cases, and only *then* run the operation. This is pointlessly inefficient.

However, if we wrap each primop with a simple function:

```
(MkLaw n a b)=(<0> (1 n a b))
```

It is extremely simple to recognize and optimize these wrappers, even in an extremely dumb interpreter. These wrappers functions can then be optimized to directly run the appropriate operations, which removes all of the execution overhead.

6 Conclusion

The new primop calling maintains a common semantics between PLAN and XPLAN by keeping the arity logic uniform, while still enabling a simple interpreter to make low-overhead calls into primops. In addition, we reduce the number of pointless pinned laws, and add the ability to namespace our effects into categories.