Designing a statically typed language around dynamically typed data structures

Hugo Musso Gualandi
PUC-Rio
hgualandi@inf.puc-rio.br

Gabriel de Quadros Ligneul
PUC-Rio
gligneul@inf.puc-rio.br

Roberto Ierusalimschy
PUC-Rio
roberto@inf.puc-rio.br

Abstract

We are designing Pallene, a statically typed companion language to the dynamically typed Lua programming language.

Our aim is to improve the performance of Lua programs by allowing performance-sensitive parts of the program to be written in a language that can be easily compiled to efficient code, while also maintaining seamless interoperability with Lua.

Pallene is designed to be amenable to standard ahead-of-time compilation techniques, to be familiar to Lua programmers, and to seamlessly interoperate with Lua. This includes sharing the runtime and garbage collector with Lua, as well as using the same data structures.

In this paper we focus on the design of Pallene's array type, based on Lua's tables (associative arrays). We considered multiple designs, and examined the performance vs expressiveness trade-offs among them.

ACM Reference Format:
Hugo Musso Gualandi, Gabriel de Quadros Ligneul, and Roberto Ierusalimschy. 2018. Designing a statically typed language around dynamically typed data structures. In *Proceedings of XIV Dynamic Languages Symposium (DLS).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Dynamic languages are popular but their performance is typically not as good as that of static languages. There are at least three approaches to make dynamic languages faster: scripting, just-in-time (JIT) compilation, and optional types.

The scripting approach is to combine the high-level dynamically typed language with a low-level statically typed system language [11]. A main advantage of this approach is that it allows each language to be used for the tasks it is most suited for. The most visible disadvantage is that there is a mismatch between the languages. Both syntactic and semantic differences complicate working with both languages at the same time. For example, it is not a trivial task to rewrite a piece of dynamic code in the static language, if more performance is desired. There is also a run-time overhead when crossing the boundary between the languages due to the need of converting between different data representations.

Just-in-time compilers [4] use run-time information to generate specialized and optimized machine code at run-time. This approach is attractive for dynamically typed languages because it can archive higher speedups than usually is possible with ahead-of-time compilation for these languages. One challenge with JIT compilation is that implementing a JIT compiler requires a lot of low-level work. Currently, designing high-level frameworks for JITs is an active research area and existing solutions still cannot archive the same level of performance as hand-written compilers [3, 19].

Another problem of just-in-time compilers is that the optimizations are not guaranteed. Programmers often need to rewrite their code to use idioms that the compiler is able to better optimize [6]. This may happen due to functions or language features that are not optimized by the JIT compiler. For example, LuaJIT [12], a state-of-the-art JIT compiler for Lua, cannot optimize traces that create closures [15]. For example, if the `increment_number` function shown in Figure 1 is called inside a inner loop, LuaJIT will not be able to optimize the whole loop.

Optional type systems mix the static and dynamic typing disciplines by allowing programmers to add type annotations to programs from a dynamically typed language. Static type systems can be useful for compile-time error checking, as a form of lightweight documentation and as an aid for compiler optimizations. Optional type systems also seek to reduce the mismatch between the static and dynamic languages, by promoting a smooth transition between them [18]. One of the challenging aspects of optional types is to design a type system that is at the same time simple, sound, and amenable to optimizations. Many optional type systems either sacrifice soundness in search for simplicity [2] or introduce significant type-checking overhead at run-time [17].

Based on these observations, we proposed Pallene, a statically typed companion language to Lua [5]. From the scripting worldview, the idea of Pallene is to be a system language that has less of a mismatch against the dynamic language.

```
function increment_numbers(text)
  return text:gsub("[0-9]+", function(s)
    return tostring(tonumber(s) + 1)
  end)
end
```

Figure 1. This function cannot be optimized by LuaJIT because it creates an anonymous closure.

To reduce the static mismatch, Pallene is a subset of Lua augmented with typed annotations. To reduce the run-time overhead, Pallene shares Lua's data structures, garbage collector and runtime. The code generated by the Pallene compiler also operates on Lua data structures directly, bypassing the Lua–C API. The Lua interpreter loads Pallene modules just like it would load a regular C extension module. Note that Pallene runs with a standard Lua interpreter; it demands no modifications there.

In this paper, we discuss strategies on how to design and implement arrays in Pallene in order to interoperate efficiently with Lua and its data structures. Since we do not have the freedom to change Lua's data representation, we do not expect Pallene to reach the same level of performance as low-level C code using idiomatic C data structures. However, by using the same data representation, we should avoid introducing additional run-time overhead when crossing the boundaries between the two languages. We do expect this approach to be competitive with a JIT compiler; the JIT is also limited to using the same data structures as the dynamic language does.

Section 2 discusses the design of Pallene and how it combines the good aspects of scripting, just-in-time compilation, and optional types. Section 3 presents how Lua tables are implemented. This is important to guide our choices on the implementation Pallene arrays. In Section 4, we discuss different possible designs for Pallene arrays, and how there is a trade-off between flexibility and performance. In Section 5, we evaluate the performance of Pallene arrays and verify that they are competitive with a good JIT compiler. Finally, in Section 6, we draw some conclusions.

2 The Pallene Language

To achieve good performance in the scripting approach, we need to rewrite our programs in the static language. Similarly, optionally typed languages demand us to add static types to optimize our code, also restricting the use of several dynamic language features. The JIT approach in theory doesn't require any change to the code; in practice, however, we often need to avoid language features that are "optimization killers" and "not yet implemented" features [1, 15]. In the end, all approaches lead programmers to use only a subset of the original dynamic language.

This realization led us to propose the Pallene language[1] [5]. Pallene is a statically typed companion language to Lua. Broadly speaking, it is a performance-oriented subset of Lua with explicit type annotations. Instead of having ill-defined restrictions, we embrace them to create a well-defined language subset. This allows us to obtain good performance while keeping type system and compiler implementation simple.

[1]Pallene is a fork of the Titan project [10], to allow us to concentrate on researching performance aspects of this approach.

```
function copy(xs: {integer}): {integer}
  local ys: {integer} = {}
  for i = 1, #xs do
    ys[i] = xs[i]
  end
  return ys
end
```

Figure 2. A function that copies an array of integers. In this case, Pallene is free to not check the types of the array elements.

Like scripting, Pallene and Lua are separate languages, each well suited for its job. But Pallene and Lua have less of a mismatch because one is a subset of the other; moreover, as we will see, Pallene works directly with Lua data structures, greatly reducing the run-time mismatch.

Pallene programs offer speedups comparable to those of JIT compilers. Unlike JIT, these speedups are predictable—programs that we cannot optimize are rejected by the language instead of being allowed to run slower than the programmer expects.

Like optional type systems, we use type annotations to aid code generation. We also borrow the idea of the Gradual Guarantee from gradually typed languages [16]. Unlike many optional type systems, it is not a goal for us to be able to type check all idioms the dynamic language supports; we intentionally restrict ourselves to only the constructs that we know we can generate good code for. This approach greatly simplifies Pallene's type system.

Unlike many optional type systems, detecting type errors at compile time is not a priority for Pallene. Our priority is using types to aid with performance. For example, if we perform a run-time type test on a dynamically typed value, we can store the result in machine registers, without the type tag. We can contrast this to optional type systems like TypeScript [2] and Typed Lua [9], which discard type annotations before actually running the program.

It is also not a priority to have a well-defined behavior for when run-time type errors should occur. We believe that since performance is our primary goal, we should give the Pallene compiler the freedom to move or remove run-time type tests, as long as doing so does not impair soundness. For example, consider the array copying operation in Figure 2. Lua can pass any value when it calls the copy function. As the function needs to traverse xs, Pallene must check that it is an array (actually a Lua table, as we will see). However, checking the type of the array elements is more subtle.

When the xs array contains a non-integer, Pallene is allowed to produce a run-time type error when attempting to read it—however, it is not required to do so. If it is more efficient to directly move the element from xs to ys, it would not check the types. If it is more efficient to move elements

```
local arr = {}
for i = 1, 100 do
  arr[i] = 10*i
end
```

Figure 3. Lua tables as arrays.

using an intermediate register to hold the integer, it will check the type.

It is important to emphasize that Pallene is a type-safe language, just like Lua. At run time, it will never use values of one type as if they were of another type. For instance, in the Figure 2 it will never attempt to index `xs` without first ensuring that it is actually an array.

3 Data Representation in Lua

Lua represents all values internally with a tagged union of type `TValue`. (A tagged union is a structure with a union and a tag.) Primitive values, such as integers, floats, and booleans, are stored directly into the union. Other values, such as strings, tables, and closures, are dynamic allocated; the union only stores a pointer to them. In 64-bit machines, standard Lua use 64-bit integers and double-precision floating-point numbers. Due to alignment, the final size of a `TValue` is 16 bytes. (In 32-bit machines, it is possible to compile Lua as what is called “small Lua”, which uses 32-bit integers and single-precision floats, resulting in `TValue` with 8 bytes.)

Tables are the sole data-structuring mechanism in Lua. They can be used to represent arrays, records, modules, objects and even the global variable environment. For example, a table with integer keys can be used as an array, as shown in Figure 3.

The reason why tables are used for everything in Lua is that it simplifies the language and its implementation. In particular, having a single widely-used data structure greatly simplifies the Lua–C API, which is used for the Lua standard library and other C modules. This simplicity is one of the key reasons why Lua is suited as an embeddable and extensible scripting language [8].

Lua tables are associative arrays that can map keys of any type (including tables or functions) to values of any type. The exception is the **`nil`** value, which cannot be used as a key or value—it represents the absence of a value in the table. Reading from a key that has never been set returns **`nil`** as a default value and assigning **`nil`** to a table field removes it from the table. Iterating over a table only visits keys corresponding to non-nil values.

Arrays in Lua are merely tables with integer keys. By convention, they start at one. Nonetheless, zero and even negative numbers are also valid keys.

Because arrays are simply tables, they do not have any intrinsic concept of length. Sometimes, programmers store the length in the table itself, usually using the string `"n"` as the key. More often than not, however, they rely on the length operator #.

```
local polyline = {
  {x = 1.0, y = 2.5},
  {x = 2.0, y = -0.5},
  {x = 3.0, y = 2.5},
  color = "red",
  thickness = 3.0,
}
```

Figure 4. Example of mixed table in Lua.

The length operator naturally needs to be defined for every possible Lua table, not only those that are being used as arrays. Given the lack of an intrinsic length, the meaning of the this operator is determined by what integer keys are nil or non-nil. Lua does this in a unique way, defining the length operator to return the index of a *border* in the array. A border is an index that is present in the table but where the next index is not, as described here:

$$\text{isborder}(t, i) \overset{\text{def}}{=} (i = 0 \lor t[i] \neq \text{nil}) \land t[i+1] = \text{nil}$$

If the table has multiple borders, the length operator can return any of them (due to performance reasons). But if there is only one border, the result matches what most would intuitively consider to be the length of an one-based array. In Lua, these tables with only a single border are called *sequences.*

Keys that are not positive integers interfere neither with the concept of sequences nor with the length operator. In Lua, the use of mixed tables is quite common, as exemplified in Figure 4.

Implementing tables

The first versions of the reference Lua interpreter implemented tables using hash tables (hence the name). Hash tables are very general and can represent all uses of Lua tables, but they have sub-optimal performance when used to represent arrays. Starting with Lua 5.0, the reference interpreter switched to a new hybrid representation for tables with better performance for arrays [7].

Each Lua table is represented with a combination of an array part and a hash part. Positive integer keys up to the capacity of the array part are stored there. Everything else (non-integer keys and larger integers) is stored in the hash part. This representation has better performance for arrays for two reasons. Accessing an array is faster than accessing a hash table; arrays also require less memory because only the values need to be stored—the keys are implicit.

Lua’s hybrid representation allows tables to contain non-integer keys while still being able to use the faster array

```
local arr = {}
for i = 100, 1, -1 do
  arr[i] = 10*i
end
```

Figure 5. A Lua array being initialized backwards.

representation for the integer keys. As we already mentioned, this kind of use is common in Lua.

The array part and the hash part of a Lua table can grow over time as new values are inserted in the table. When a new key is inserted, Lua first checks whether it fits in the array part. (That is, the key is positive integer up to the array capacity.) Otherwise, it goes to hash part. If the hash table is already full, then Lua performs a *rehash* operation to resize the array and hash parts. This operation may potentially move some values between the hash part and the array part.

During a rehash, the hash part is resized to the smallest power of two that can store all keys not in the array part. What is still missing in this discussion is how Lua chooses the capacity of the array part.

The rule that Lua uses is that it resizes the array part to the largest power of two such that more than half of the indices from one to the array capacity have non-nil values. Arrays in Lua may be initialized with a capacity that is not a power of two—this is useful to save memory in array literals that don't grow. Nevertheless, arrays always end up with a power-of-two capacity after they grow.

We will now illustrate the rehash operation with some examples. Consider the array initialization loop from Figure 3. The `arr` table starts empty, with zero capacity for both the array and hash parts. When the first element is inserted (key 1), it fits neither in the array part nor the hash part, triggering a rehash. The array part grows to a capacity of 1 and the hash part stays at 0. The table is further rehashed when keys 2, 3, 5, 9, 17, 33, and 65 are inserted, resulting in an array part of capacity 128 and a hash part that remains empty.

Now consider the example in Figure 5, which creates a similar sequence but initializes it backwards. The table also starts empty. When the first key (100) is set, it is stored in the hash part because Lua doesn't think it is worth it to allocate a 128 element array to store just a single value. Similarly to the case where the keys are inserted in increasing order, the table is rehashed when the 2nd, 3rd, 5th, 9th, 17th, 33rd and 65th elements are added. In the initial rehashes, all the values are stored in the hash part and the array part remains empty. But after the final rehash, Lua switches to using a 128 element array, and the hash part becomes empty, ending up in a similar state to the increasing keys case.

The majority of Lua sequences (arrays without holes) end up being fully stored in the array part because they are initialized sequentially, as in the two previous examples.

```
local arr = {}
arr[5] = 10*5
for i = 1, 4 do
  arr[i] = 10*i
end
```

Figure 6. A pathological initialization order for Lua arrays.

```
local arr = {}
arr["a"] = 1
arr["b"] = 1
arr["c"] = 1
arr["d"] = 1
arr["e"] = 1
for i = 1, 3 do
  arr[i] = 10*i
end
```

Figure 7. Another pathological initialization order for Lua arrays.

However, there are rare corner cases that make it possible to construct a sequence in Lua such that some of the values are stored in the hash table part. For example, consider the program in Figure 6, which initializes an array with the unusual order 5, 1, 2, 3, 4. At first, the 5 is stored in the hash table part. Then, as 1, 2, and 3 are inserted, they are stored in the array part. The insertion of key 3 triggers a rehash but the array part only grows to 4 elements, since the four elements currently present in the table (1,2,3, and 5) are not enough for Lua to grow the array part to a capacity of 8—for that it would need at least 5 elements. After this, the sequence is completed with the addition of key 4, which goes into the array part. However, key 5 will continue to be stored in the hash part unless more elements are inserted to trigger an additional rehash.

Another way for sequences to end up with some keys in the hash part is if the table contains non-integer keys which were inserted before the integer keys. If there is extra space available in the hash part, the integer keys may be stored there instead of triggering a rehash and growing the array part. This is illustrated in Figure 7. The first five string keys (`"a"`, `"b"`, etc) grow the capacity of the hash part to eight, leaving space for three integer keys. In the end, all of the table keys will be in the hash part.

4 Arrays in Pallene

As we discussed previously, to obtain maximum interoperability with Lua, we want Pallene data types to be Lua data types that can be directly manipulated by both languages. We also want their semantics to be a restricted subset of the

```
function foo(f: integer -> float): float
  local i: integer = 17
  return f(i)
end
```

Figure 8. A higher order function in a gradually typed language.

Lua semantics, following the Gradual Guarantee principle. But what should these restrictions be?

According to this guarantee, Pallene code should either behave like the equivalent (untyped) Lua code or raise a type error. This still leaves us with some latitude for defining what are type errors and when to raise them.

Consider the fully annotated function `foo` in a hypothetical gradual typed language, shown in Figure 8. Consider also that this function is being called from dynamic code, so that `f` can be any value. The usual semantics for this situation would be that, when called, `foo` would immediately check whether `f` is a function. The compiler already checked that the argument `i` is an integer. The result of `f(i)`, however, is only checked after the call, and (of course) only for this particular argument. Coming from a dynamic language, the function `f` could return non-floats for other inputs.

An alternative semantics would be for the function `foo` to check that `f` is actually a "function from integers to floats", whatever that means. For instance, in a hypothetical implementation, "function from integers to floats" could mean functions whose source code could be typed with this type annotation. Clearly, that would not be viable.

With arrays, however, this design is feasible: when receiving a value to be used as an array of floats, the code checks whether the value is an array and that all its elements are indeed floats. Following the terminology of gradual typing, we can call this semantics *eager*; the semantics that checks types only as the value is used we call *lazy*. Note that these terms are relative. For instance, our lazy example could be even lazier, checking that `f` is a function only when it is actually called.

From a performance point of view, a more eager semantics introduces a high upfront cost, but it does give the compiler more room for optimization. Conversely, a lazier semantics has low upfront cost, but it gives weaker guarantees to the compiler.

In the case of Lua, the meaning of what is an array is more subtle than in most other languages, as Lua does not have an array data type. A most strict definition for arrays in Lua would be sequences without extra keys and with all elements having the correct type. Less strict definitions could lift some of theses restrictions. For instance, the array could have non-integer keys, such as a `"n"` to store its size. Or the keys could be sparse, provided that the code does not access the absent ones.

```
TValue *slot;
if (i < xs->array_capacity) {
  slot = &xs->array[i];
} else {
  slot = hashtable_get(xs->hash, i);
}
double out = check_and_get_float(slot);
```

Figure 9. Reading from a Pallene array under the hash semantics.

```
local sum: double = 0.0
for i = 1, N do
  sum = sum + xs[i]
end
```

Figure 10. A simple Pallene loop.

We expect that a more restrictive definition for arrays should give more opportunities for the compiler, at the cost of restricting the programming idioms that are supported. Our goal with Pallene is to find a sweet spot, targeting the subset of Lua that programmers naturally use when they are writing in a performance-minded style.

The Hash Semantics

The least restrictive approach for Pallene arrays would be to allow any Lua table to be used as a Pallene array. We call this approach the *hash semantics*. Pallene would only try to read and write to positive integer keys of the table, but otherwise there would be no restrictions on the contents of the table and what Lua could do before handing it over to Pallene. For example, the integer keys may be sparse, and the table may also contain non-integer keys. This generality means that at run-time some of the keys that Pallene will try to access might be on the array part of the table, while others might be on the hash part.

Figure 9 shows C pseudocode of how Pallene would read an element from an array of floating-point numbers under the hash semantics. Even in this more general approach, Pallene presents various optimization opportunities to the compiler, compared to interpreted Lua. The program can be compiled directly to machine code, without going through a bytecode interpreter. Additionally, the code can be specialized to assume that the key in an integer. Finally, the type test at the end allows us to store the resulting floating point number in a machine register, without a run-time type tag.

However, working with a generic table representation leaves obstacles for the Pallene compiler. Although most Pallene programs access exclusively the array part, the mere presence of the else branch inhibits some optimizations. For example, consider the simple loop in Figure 10. Under the

```
if (i >= xs->array_capacity) {
  /* This call never returns */
  out_of_bounds_error();
}
TValue *slot = &xs->array[i];
double out = check_and_get_float(slot);
```

Figure 11. Reading from a Pallene array, under the strict array semantics.

Hash semantics, Pallene must perform a test at each iteration of the loop to decide whether it should read from the array part of from the hash part. This test cannot be removed or put outside the loop unless we perform a loop switching transformation, which may result in a large blowup in code size. Therefore, we are also interested in seeing if more restrictive semantics for the Pallene arrays would allow additional optimizations.

The Strict Array Semantics

On the most restrictive end of the spectrum, would be a semantics that only allows arrays where all their values are stored in the array part of the table. Pallene would be able to read from these arrays using code similar to that in Figure 11. However, what would this mean for the language semantics? It turns out that restricting Lua tables like this would reveal implementation details that are not naturally part of the Lua semantics. The largest problem is that this restriction depends on the previous history of the table, and not only on its current contents. Even if the table is a Lua sequence (with integer keys from 1 to N with no holes in them), it is still possible to have some of the integer keys stored in the hash part, as discussed in Section 3. In the end, while this more restrictive approach for Pallene arrays would open additional opportunities for optimization, we deem it unacceptable.

Additionally, this restrictive semantics would also forbid some common Lua idioms, such as mixed arrays. It would also forbid initializing an empty array by assigning to sequentially increasing integer keys. We would need to use standard library functions to grow the array instead. That said, performance-seeking programmers are usually willing to appease the compiler by changing their coding style in minor ways, as we discussed before.

The Resize Semantics

Another approach for arrays in Pallene is to force the keys we want to access to be in the array part, as illustrated in Figure 12. When reading from the array, this approach may incur in a relatively expensive array rehash operation. However, we expect that this should be a rare occurrence, since Lua arrays commonly store their keys in the array part.

At a first glance, both the hash semantics and the resize semantics should have a similar impact for the compiler.

```
if (i >= xs->array_capacity) {
  resize_array_part(xs, i);
}
TValue *slot = &xs->array[i];
double out = check_and_get_float(slot);
```

Figure 12. Reading from a Pallene array, under the resize semantics.

```
double sum = 0.0;
if (N >= xs->array_capacity) {
  resize_array_part(xs, N);
}
for (int i = 0; i < N; i++) {
  TValue *slot = &xs->array[i];
  double v = check_and_get_float(slot);
  sum = sum + v;
}
```

Figure 13. The resize semantics is more friendly to loop invariant code motion.

They both have a rarely taken `if (i >= array_capacity)` branch which nevertheless gets in the way of compiler optimizations by calling a function that may modify the environment, including the array itself and its capacity.

The main advantage of the resize semantics is that it is more amenable to loop optimizations, such as moving bound-checking tests out of the loop. For example, the resize semantics allows the Pallene compiler to generate code similar to that in Figure 13 for the simple loop from Figure 10. These loop optimizations usually depend on knowing that the array won't be resized while we are operating on it. Typically, this requires working with a fresh array that we know isn't being aliased elsewhere, or being in an inner loop that does not call other Pallene or Lua functions, which might modify the array. Nevertheless, these scenarios are not uncommon in tight array loops.

5 Performance Evaluation

In this section, we perform experiments to evaluate some of our assumptions and hypothesis about Pallene. This section has two main points:

- To confirm our assumption that the "rewrite it in C" approach is not viable in several scenarios.
- To evaluate our hypothesis that the Pallene approach can be competitive with a good JIT compiler in terms of performance.

We also measured the cost of some individual components of our implementation, such as accessing the hash part and the run-time tag checking.

We selected six micro benchmarks for this evaluation. All are array-focused benchmarks that have been commonly used in the Lua community. They are known algorithms, implemented in a straightforward manner, without any performance tuning for a particular implementation. The six benchmarks are the following:

Matmul: multiplies two floating-point matrices represented as arrays of arrays.
Binsearch: performs a binary search over an array of integers.
Sieve: computes a list of primes with the sieve of Eratosthenes algorithm.
Queens: solves the classic eight-queens puzzle.
Conway: simulates Conway's Game of Life, a cellular automaton.
Centroid: computes the centroid (average) of an array of points, each represented as an array of length 2.

We performed our tests in a 3.10 GHz Intel Core i5-4440 with 8 GB of RAM. We measured the running time of each benchmark five times and considered only the fastest result. The Matmul, Queens, and Conway benchmarks feature algorithms with super-linear run time, and we chose a large enough N to make the benchmarks take around a second to run. For the Binsearch, Sieve, and Centroid benchmarks, we repeated the benchmark inside a loop to achieve the same effect.

For the main experiment, we ran each benchmark in four ways: Lua, LuaJIT, Lua–C API, and Pallene. Lua means the standard Lua interpreter version 5.4-work2. For LuaJIT we ran the same source code under the LuaJIT 2.1.0-beta3 JIT compiler. For Lua–C API we rewrote the benchmark code in C, but operating on standard Lua data structures through the Lua–C API. Pallene means our implementation, running the same Lua source code with added type annotations.

For LuaJIT, our experimental setup did not have a warmup step, which is commonly done when measuring JIT compilers. Unlike other such compilers, LuaJIT does not require a warmup period, because its compilation phase is unnoticeable [14]. We also think that it is fairer to include the whole time, as this is actual time spent running the program. (We do not assume that all Lua programs are running web servers.)

Figure 14 shows the elapsed time for each benchmark, normalized by the Lua benchmark results. (The precise values are shown in Figure 17.) As we anticipated, Pallene running times are comparable with LuaJIT. The Lua–C results are all over the place; for benchmarks featuring lots of array operations (Matmul and Centroid), the Lua–C API overhead outweighs the gains from rewriting in C.

Let us now analyze the Pallene versus LuaJIT situation in more detail. The only benchmark where LuaJIT is significantly faster than Pallene is Matmul. This difference is

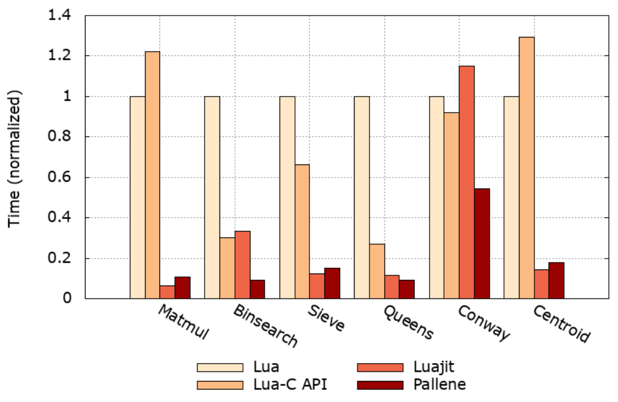

Figure 14. Comparison of Pallene with Lua, LuaJIT and Lua–C API. The time is normalized by the Lua result.

N	M	Time ratio	Pallene time	LuaJIT time	Pallene LLC miss	LuaJIT LLC miss
800	2	1.64	2.17	1.32	44.13%	33.31%
400	16	1.05	1.26	1.20	3.38%	0.61%
200	128	1.02	1.30	1.27	0.13%	0.07%
100	1024	0.96	1.37	1.42	0.02%	0.07%

Figure 15. Time spent and cache misses for the Matmul benchmark on different input sizes. N is the size of the input matrix. M is how many times we repeated the multiplication. Time ratio is Pallene time divided by LuaJIT time. Times are in seconds. LLC load misses are a percentage of all LL-cache hits.

due to memory access; LuaJIT uses the *NaN-boxing* technique to represent values [13]. This means that an array of floating-point numbers in LuaJIT only uses 8 bytes per number against the 16 bytes used by from Lua and Pallene, reflecting in a higher cache miss rate for Pallene. To confirm this explanation, we ran the same benchmark under smaller values of N, as shown in Figure 15. Smaller matrices fit better in the cache and result in faster execution speeds and less cache misses for both. For this benchmark, we measured the run time and cache-miss rates with the Linux `perf` tool. The NaN-boxing also explains the smaller difference LuaJIT and Pallene in the Centroid benchmark.

The NaN-boxing technique, however, is accompanied by other problems that usually do not show up in benchmarks. LuaJIT does not support unboxed 64-bit integers. (This prevented us from using the xorshift128+ PRNG algorithm as one of our benchmarks). It also limits the total memory that LuaJIT can address to 4GB. These limitations are why NaN-boxing is not used in standard Lua anymore.

The Binsearch benchmark highlights a particularly bad scenario for trace-based JITs, such as LuaJIT. The inner loop of the binary search features a highly unpredictable branch,

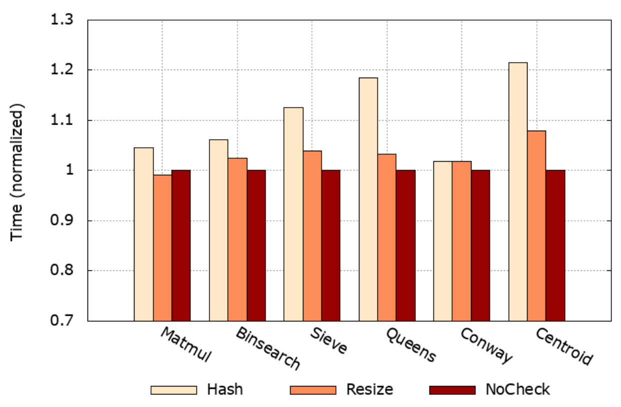

Figure 16. Comparison of Hash, Resize, and NoCheck array implementations. The time is normalized by the NoCheck result.

Benchmark	Lua	LuaJIT	Lua–C API	Hash	Resize	No Check
Matmul	10.21	0.66	12.47	1.16	1.10	1.11
Binsearch	8.96	2.99	2.70	0.86	0.83	0.81
Sieve	7.14	0.89	4.73	1.17	1.08	1.04
Queens	14.19	1.61	3.85	1.47	1.28	1.24
Conway	2.12	2.44	1.95	1.15	1.15	1.13
Centroid	9.31	1.33	12.04	1.86	1.65	1.53

Figure 17. Exact running times for our benchmarks, in seconds. Resize is the default Pallene implementation.

forking the hot path. This is not an issue for Pallene and other ahead-of-time compilers.

The Sieve and the Queens benchmark need no further explanation as the results were quite expected. Both LuaJIT and Pallene are around ten times faster than Lua.

The Conway benchmark is a little surprising as LuaJIT performed worse than standard Lua. The bulk of the time in this benchmark is spent doing string manipulation and garbage collection. This benchmark's unusual result is most likely due to the new garbage collector introduced in Lua 5.4.

We also ran experiments to measure the differences between individual components in our implementation of arrays in Pallene. We compared three different implementations of the compiler: Hash, Resize, and NoCheck. Hash means the lazy implementation of Pallene arrays, which allows access to arbitrary integer keys (they may be stored either in array or hash parts of the table). Resize refers to the implementation that resizes the array part as needed, so it never has to use the hash part. Finally, NoCheck is a variant of the Resize implementation without any run-time type checks. This is obviously unsafe, but it allows us to estimate the cost of the tag checks.

Figure 16 shows the results of this experiment, normalized against the NoCheck implementation. As with the other benchmarks, the real times are in Figure 17. Clearly, the Resize implementation performed better than Hash.

For the Hash implementation, it is important emphasize that these particular benchmarks never accesses elements in the hash part of the tables, as the array elements are always in the array part. The performance differences are due to compiler optimizations. First, range analysis allows the bound checks to be moved outside the loop. Second, the mere presence of a function call inside loop (to access the hash part) hinders some compiler optimizations, even if that function is never called.

In all our benchmarks, the tag checks introduce little overhead (up to 10%). Analysis with `perf` shows that for all them presence of tag checks increases the number of machine instructions the programs run. For instance, the Matmul benchmark executes twice as many instructions when tag checks are enabled. However, this is not reflected in the time spent. Because our benchmarks are memory bound, the Intel Core CPU is able to execute most of these extra instructions "for free", due to pipelining. (This can be seen through `perf` in the instructions per cycle measurement). For other CPUs, the cost of tag checking may be higher. Additionally, if the benchmarks were CPU-bound then we expect that the tag-checking overhead would be higher.

6 Conclusion

In this paper we discussed the implementation of arrays in Pallene, a companion language for Lua. A key ingredient in the Pallene approach is to use the same data structures and runtime as Lua. This poses some design problems that we addressed in this paper: how to define what values correspond to the array type, given that Lua doesn't have one; and how to maximize performance sacrificing a minimum of flexibility.

We showed that by being aware of how Lua implements its arrays and by taking into account how programmers typically use them, we were able to implement arrays in Pallene with performance compatible with LuaJIT, a state-of-the-art JIT compiler. The final implementation imposes no restrictions on how arrays are used. For instance, Pallene supports mixed tables, that is, tables containing both integer keys implementing an array and other keys with additional data.

One of the most promising aspects of our approach is its simplicity. By focusing on a restricted subset of the dynamic language, we were able implement a straightforward ahead-of-time compiler, leveraging mature compiler backends. Implementing a JIT compiler with similar performance would be much more challenging.

Although the type system we obtained in the end is specific to Lua, we believe that the same ideas could also be used with a different language as starting point.

References

[1] Antonov, P., et al. V8 optimization killers, 2013. Retrieved in 2017-01-08. Full author list available at https://github.com/petkaantonov/bluebird/wiki/Optimization-killers/_history.

[2] Bierman, G., Abadi, M., and Torgersen, M. Understanding typescript. In *28th European Conference on Object-Oriented Programming* (2014), R. Jones, Ed., ECOOP '14, Springer Berlin Heidelberg, p. 257–281.

[3] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (2009), ICOOOLPS '09, ACM, p. 18–25.

[4] Deutsch, L. P., and Schiffman, A. M. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1984), POPL '84, ACM, p. 297–302.

[5] Gualandi, H. M., and Ierusalimschy, R. Pallene: A statically typed companion language for lua. http://www.inf.puc-rio.br/%7Eroberto/docs/pallene-sblp.pdf, 2018. submitted to the XXII Brazilian Symposium on Programming Languages.

[6] Guerra, J. LuaJIT hacking: Getting next() out of the NYI list. CloudFare Blog, Feb. 2017. https://blog.cloudflare.com/luajit-hacking-getting-next-out-of-the-nyi-list/.

[7] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. The implementation of lua 5.0. *Journal Universal Computer Science 11*, 7 (July 2005), 1159–1176.

[8] Ierusalimschy, R., De Figueiredo, L. H., and Celes, W. Passing a language through the eye of a needle. *Communications of the ACM 54*, 7 (July 2011), 38–43.

[9] Maidl, A. M. *Typed Lua: An Optional Type System for Lua.* PhD thesis, PUC-Rio, Apr. 2015.

[10] Maidl, A. M., Mascarenhas, F., Ligneul, G., Muhammad, H., and Gualandi, H. M. Source code repository for the Titan programming language. https://github.com/titan-lang/titan.

[11] Ousterhout, J. K. Scripting: Higher-level programming for the 21st century. *Computer 31*, 3 (Mar. 1998), 23–30.

[12] Pall, M. Luajit, a just-in-time compiler for lua, 2005. http://luajit.org/luajit.html.

[13] Pall, M. Luajit 2.0 intellectual property disclosure and research opportunities. lua-l mailing list, nov 2009. http://lua-users.org/lists/lua-l/2009-11/msg00089.html.

[14] Pall, M. Luajit: Performance comparison, 2018. http://luajit.org/performance.html.

[15] Pall, M., et al. Not yet implemented operations in luajit. LuaJIT documentation Wiki, 2014. Retrieved 2017-01-08. Full author list available at http://wiki.luajit.org/history/NYI.

[16] Siek, J. G., Vitousek, M. M., Cimini, M., and Boyland, J. T. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages* (Asilomar, California, USA, May 2015), SNAPL '2015, p. 274–293.

[17] Takikawa, A., Feltey, D., Greenman, B., New, M. S., Vitek, J., and Felleisen, M. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2016), POPL '16, p. 456–468.

[18] Tobin-Hochstadt, S., and Felleisen, M. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, p. 964–974.

[19] Würthinger, T., Wimmer, C., Wöss, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., and Wolczko, M. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (2013), Onward! 2013, p. 187–204.