Under consideration for publication in J. Functional Programming

A gradually typed subset of a scripting language can be simple and efficient

HUGO MUSSO GUALANDI and ROBERTO IERUSALIMSCHY
Pontifical Catholic University of Rio de Janeiro
(*e-mail:* `hgualandi@inf.puc-rio.br`)

Abstract

Most research on gradual types for scripting languages has approached the problem from the point of view of adding types to an existing dynamic language. The natural implementation approach in this context is to compile the typed language down to the untyped one, and reuse the existing language implementation. Run-time type checks are often seen as a performance cost to pay for type safety and much research has been dedicated to reducing this overhead using type inference or just-in-time compilers.

In this paper we argue that ahead-of-time compilation is a viable approach for producing an efficient gradually typed system, if one is willing to design the typed language around the operations that benefit most from type-specialization. Instead of designing the typed language to supersede the dynamic one, the typed language can be designed to work side-by-side with it. This approach follows the classic scripting tradition of combining a dynamically typed scripting language with a statically typed system language: the dynamic language provides flexibility, while the typed one provides performance. We exemplify it with Pallene, a typed counterpart of the Lua programming language. Pallene programs are compiled ahead-of-time, and produce binary modules which can be dynamically-linked with Lua.

Pallene is heavily inspired by gradual typing. However, it intentionally is not a superset of Lua. The key difference is that dynamically typed values must be downcasted to a concrete type before they can be used. This restricted scope allows Pallene to be implemented with a simple ahead-of-time compiler. In this context, we observe that the performance gains from type specialization are large, and the overhead of run-time type checking is small and often negligible. The performance can be similar to that of a just-in-time compiler but at a fraction of the implementation complexity.

1 Introduction

Initial research on gradual typing focused on developing the theory of how to combine static and dynamic typing in a single language. However, once gradual typing was implemented in practice the problem of performance was soon brought to attention. Takikawa et al. were some of the first to raise the alarm, after noticing that run-time type checking in Typed Racket could lead to overheads of over 100x compared to dynamically typed programs (Takikawa *et al.*, 2016). Since then, there has been much research on the topic of gradual typing performance.

Often, run-time type checks are seen as an inevitable source of overhead, a price to pay for type soundness. For example, Greenman and Felleisen found that in a version of Typed Racket with first-order casts, the overhead of type checking grows linearly with the number of type annotations (Greenman & Felleisen, 2018). Some research has focused on trying to minimize the overhead of type checking. For instance Campora et al. suggest an algorithm for estimating the cost of run-time type checking, and Vitousek et al. propose an algorithm to elide redundant type checks in Reticulated Python (Campora *et al.*, 2018; Vitousek *et al.*, 2019).

One promising research avenue for improving the performance of gradually typed languages has been to use JIT compilers. For example, Bauman et al. show that the Pycket JIT compiler obtains better performance for Typed Racket benchmarks (Bauman *et al.*, 2015). Richard et al. show that run-type type checking may sometimes be subsumed by type tests that a JIT compiler would already normally perform as a part of its optimization strategy (Richards *et al.*, 2017).

Another avenue for attacking the problem is that if the gradually typed language is designed from scratch, instead of being based on an existing dynamic language, it might be possible to obtain better performance. One such example would be the Nom language from Muehlboeck and Tate, which uses nominal typing pervasively to reduce the overhead of run-time type checking (Muehlboeck & Tate, 2017).

In a previous paper, we have introduced Pallene, a typed subset of Lua designed to act as a system-language counterpart to Lua's scripting (Gualandi & Ierusalimschy, 2020). Pallene is implemented by an ahead-of-time compiler, which uses the type annotations present in the program to generate efficient machine code. This code can dynamically-loaded by Lua, in a similar manner to C extension modules. In this paper, we discuss how the combination of Lua and Pallene can be seen as a gradually typed language, with a transient type-checking semantics similar to that of Reticulated Python (Vitousek *et al.*, 2014). Our goal is to show that implementing an efficient gradually typed language based on an existing dynamic language is possible not only in theory, but also in practice. We want to bring attention to three ideas that have guided the design of Pallene.

The first idea is that compile-time type information can bring substantial performance gains if the implementation is designed to take advantage of it, and that these gains can be much larger than the overhead of run-time type checking. Types can be seen as a source of performance, not only as a source of overhead.

The second idea is that in the context of a gradually typed language, a simpler ahead-of-time compiler may be able to achieve results that are comparable with a more complex just-in-time compiler. JIT compilers optimize the program based on type information collected at run-time. Since these optimizations are speculative, they need to be able to deoptimize the program state if the actual types encountered are not what was initially predicted. Some of the most complex parts of a JIT are related to this type profiling and these deoptimizations, but both of these problems are avoided in a language with explicit type annotations.

The third and final observation is that some of the largest performance gains happen when the compiler is able to specialize the generated code for its types. This has led us to focus the design of Pallene around the parts of Lua where the performance benefits the most from type annotations. One result of this is that Pallene is a typed subset of Lua, instead of a superset. In Pallene, a dynamically typed value of type `any` must be explicitly downcasted to a concrete type before it can be used. While by some definitions this would dismiss Pallene as not being a gradually typed language, we argue that we should always look at the combination of Lua+Pallene as a whole, since Pallene is always intended to be used side-by-side with Lua.

The rest of this paper is organized as follows: Section 2 discusses the use of just-in-time compilers for dynamic languages, and how those lessons can be applied to gradually typed languages. As an example, we analyze the performance of LuaJIT, a JIT compiler for Lua. Section 3 compares ahead-of-time and just-in-time compilation for dynamic languages. We introduce Lua-AOT, an experimental ahead-of-time compiler for Lua 5.4 and measure its performance. Moving on, Section 4 discusses how gradual typing can allow an ahead-of-time compiler to be competitive with a JIT compiler, while keeping the implementation much simpler. Section 5 contains experiments evaluating the overhead of migrating programs from Lua to Pallene. These experiments indicate that in the context of Pallene, adding types usually leads to better performance, for all possible combinations of Lua and Pallene modules. Furthermore, Section 6 discusses how Pallene is intended to always be used side-by-side with Lua, and how this allowed us to simplify its implementation. Finally, in Section 7 we provide our concluding remarks.

2 JIT compilers for dynamic languages

The fastest implementations of dynamically typed languages tend to be based on just-in-time compilation. Despite the name, JIT compilers for dynamic languages usually include both an interpreter and a compiler. Programs start being executed in the interpreter, which identifies which commonly executed sections of the code are candidates for compilation. Then, these hot sections are instrumented to collect type information. Finally, this type information is used to generate efficient machine code. Since the type information is collected at run-time, there is no guarantee that interpreter will correctly identify the full set of types that will flow through that section of the program. For this reason, the compiler also introduces type guards in the compiled code, to exit the compiled code if an unexpected type is encountered. This is called deoptimization. It can be complex to implement, and it may be costly at run-time. For example, if the compiled code stores local variables in machine registers and the machine stack, the deoptimization will need to convert those values back into the virtual stack used by the interpreter. Furthermore, if the compiled code contained inlined functions, this deoptimization may also need to reconstruct the call stack.

To investigate the performance of a JIT compilation we focused on LuaJIT, a tracing just-in-time compiler for Lua (Pall, 2005). The results of this investigation are shown in Figure 1 and Figure 2. For now, we will ignore the other entries in these tables, namely Lua-AOT and Pallene. They will be discussed in sections 3 and 4.

The Fannkuch, Fasta, Mandelbrot, Nbody, and Spectral Norm benchmarks are taken from the popular Computer Language Benchmarks Game suite of benchmarks (Gouy, 2013). For this analysis, we reimplemented each of these benchmarks using idiomatic Lua. The Queens benchmark is from a Lua benchmark suite, and solves the famous N-queens puzzle. It also appears in our previous evaluation of the performance of Pallene (Gualandi & Ierusalimschy, 2020). The Stream Sieve benchmark computes prime numbers using lazy streams, and is adapted from the Typed Racket benchmark suite, where it was one of the benchmarks with the worst slowdowns from gradual typing (Takikawa *et al.*, 2016).

Benchmarks	Lua	Lua-AOT	LuaJIT	Pallene
Fannkuch	3.22 ± 0.00	1.55 ± 0.00	0.55 ± 0.00	0.39 ± 0.00
Fasta	4.33 ± 0.04	3.15 ± 0.06	0.82 ± 0.00	0.65 ± 0.01
Mandelbrot	8.20 ± 0.01	3.32 ± 0.03	0.90 ± 0.00	0.85 ± 0.00
Nbody	7.72 ± 0.55	4.88 ± 1.26	0.48 ± 0.00	1.10 ± 0.01
Spectral Norm	2.21 ± 0.00	1.12 ± 0.08	0.17 ± 0.00	0.17 ± 0.00
Queens	16.09 ± 0.08	9.47 ± 0.09	1.67 ± 0.01	1.32 ± 0.01
Stream Sieve	2.67 ± 0.17	2.29 ± 0.01	0.47 ± 0.01	1.73 ± 0.02

Fig. 1: A comparison of the performance of Pallene with ahead-of-time and just-in-time compilers for Lua. The first number in each column is the average time, in seconds. The second number is the difference between the average time and the maximum time, or the difference between the average and the minimum, whichever is larger.

Benchmarks	Lua-AOT	LuaJIT	Pallene
Fannkuch	0.48	0.17	**0.12**
Fasta	0.73	0.19	**0.15**
Mandelbrot	0.40	0.11	**0.10**
Nbody	0.63	**0.06**	0.14
Spectral Norm	0.51	**0.08**	**0.08**
Queens	0.59	0.10	**0.08**
Stream Sieve	0.86	**0.18**	0.65

Fig. 2: Performance of different compilation approaches compared to the reference Lua interpreter. Each entry in the table shows the average time for that benchmark divided by the average time for the reference Lua implementation. It is the inverse of the speedup factor. The fastest implementation for each benchmark is highlighted in bold.

Our experiments were run on a desktop computer with a 3.10 GHz Intel Core i5-4440 processor and 8 GB of RAM, running Fedora Linux 31. The Lua versions used were the latest available at the time of the experiment: 5.4.0-beta2 for the reference interpreter, and 2.1.0-beta3 for LuaJIT. The C compiler used was GCC 9.2. Each benchmark was run 10 times. For each benchmark and implementation, Figure 1 lists the average time in seconds, followed by the difference between the maximum time and the average time, or the difference between the minimum time and the average time, whichever was greatest. Figure 2 summarizes the results by normalizing the average times by the average time of the reference Lua interpreter. It also highlights the fastest implementation in bold.

Our experiments show that just-in-time compilation is capable of large speedups, as expected. In our set of benchmarks, LuaJIT achieved a speedup between $5\times$ and $20\times$.

We draw a connection between JIT compilation and gradual typing as follows. The machine code that the JIT compiler produces is derived from a typed version of the program, with types that were speculatively inferred by the interpreter. This generated code also contains run-time type checks, in the form of deoptimization checks. In the JIT compiler these checks are there because the type inference is imprecise, but they serve the same purpose of the type checks that are present in gradually typed programs—they detect when the compiled (typed) part of the program receives a value with an unexpected type from the interpreted (untyped) part. Since JIT compilers are able to obtain excellent performance even in the presence of these type checks, this suggests that gradually typed programs also ought to be able to do the same, if the type checks the gradually typed language performs are similar to the deoptimization checks the JIT performs. We have confirmed this hypothesis with Pallene, which we will talk more about in Section 4. We believe that this may also be applicable to other gradually typed language that employ a first-order type checking approach, as categorized by Greenman and Felleisen (Greenman & Felleisen, 2018).

Furthermore, it is known among the JIT community that deoptimization checks account for only a small percentage of the resulting program's running time. For example, Southern and Renau have measured that in the context of the V8 compiler for JavaScript the deoptimization checks account for less than 3% of the total program running time (Southern & Renau, 2016). This adds support to the idea that type checks don't have to be expensive, as long as they happen in compiled machine code, and focus on checking the type tags (constructors) of objects, without involving wrapper objects to represent contracts. We have previously observed a similar result in Pallene, where the overhead of type checks is often less than 10% (Gualandi & Ierusalimschy, 2020). This has also been noted by Kuhlenschmidt et al., which observed that the overhead of type checking was lower in a gradually typed lambda calculus that was compiled to machine code with an ahead-of-time compiler (Kuhlenschmidt *et al.*, 2019).

Another way in which JIT compilation is relevant for gradual typing is that there have been studies applying JIT compilation to gradual typing. The traditional approach of implementing gradual typing is that the compiler for the typed language uses the dynamic language as the target language for code generation. Type checks become if statements, or are encapsulated in contract objects. In this setting it is natural to examine the effect of using JIT compilation to optimize the resulting program. For example, Bauman et al, reported that the Pycket compiler was able to reduce the overhead of type checking in Typed Racket by over 90% (Bauman *et al.*, 2015).

While JIT compilation has a high ceiling in terms of performance, implementing an efficient JIT from scratch is a daunting task. For these reasons, JITs don't always implement all language features in a compatible manner. For example, LuaJIT is only fully compatible with Lua 5.1, with no plans in the foreseeable future to make it fully compatible with Lua 5.4; and the PyPy implementation for Python was only made compatible with Python 3 after many years and a significant effort (The PyPy Project, 2011). Furthermore, JIT compilers often depend on low-level machine code, which is not portable. For these reasons, we chose to investigate ahead-of-time compilation when designing Pallene.

3 Compiling dynamically typed languages

Dynamically typed languages are traditionally implemented with interpreters, while statically typed ones are usually implemented with ahead-of-time compilers. Typed languages tend to be faster, but it isn't always obvious how much of this is due to the interpretation overhead and how much is due to other factors. One way to shed light on this question is to analyze the performance of an ahead-of-time compiler for a dynamic language (Barany, 2014). In this section we present Lua-AOT, an ahead-of-time compiler for Lua that we created to study the interpretation overhead in Lua.

The reference Lua interpreter compiles the source Lua programs into a compact bytecode for a custom register-based virtual machine (Ierusalimschy *et al.*, 2005). Each VM instruction has 32 bits. 7 bits are reserved for the operation code and the remaining bits encode the parameters, which may be VM registers, immediate numeric constants, etc.

The Lua-AOT compiler that we created for this study compiles Lua bytecode into machine code by converting each VM instruction into a block of C code, and then feeding the resulting C program into a C compiler. The block of C code generated for each instruction is almost identical to the C code in the reference Lua interpreter, which is also written in C. The main exception is that Lua-AOT rewrites jumps as C gotos, instead of manipulating a virtual program counter. The parameters for each instruction are also converted to C constants instead of being decoded at run-time from the 32-bit instruction. The whole process can be seen as a partial evaluation of the reference Lua interpreter, also known as a Futamura projection (Futamura, 1999).

Lua-AOT implements almost the entirety of Lua 5.4. The main limitations are that it currently does not support coroutines or accurate debugging information. It also requires small modifications to the Lua virtual machine so that it can associate Lua functions with pre-compiled machine-code implementations.

The main optimization that Lua-AOT enables is that it removes the interpreter overhead from decoding and dispatching VM instructions. There are also some kinds of optimizations that the C compiler can do, such as constant propagation when compiling an instruction with parameters that are compile-time constants. However, Lua-AOT does not perform any kind of type-guided code generation or optimization. The generated code is still fundamentally dynamically typed and all Lua variables are stored in virtual registers on the heap instead of on CPU registers or the native stack.

We evaluated the time performance of Lua-AOT with the same methodology we used for Lua and LuaJIT and the results are again shown in Figure 1 an Figure 2. There were noticeable speedups compared to the reference Lua interpreter, ranging from $1.15\times$ to $2.5\times$. However, they were much smaller than those of Pallene and LuaJIT, suggesting that the overhead of dynamic typing is higher than the overhead of interpretation. (We will discuss the Pallene results in the next section.)

We also measured the size of the resulting executables. For our set of benchmarks, these sizes are listed in Figure 3. The AOT-compiled machine code was substantially larger than the equivalent Lua bytecode. The geometric mean of the size increase was $25\times$. Part of this size is due to some size overhead that is always present, as evidenced by the 3.7KB executable that is generated when compiling an empty Lua file. However, most of the size increase in the Lua-AOT executables is due to the larger code size for the compiled functions.

In summary, Lua-AOT was able to produce faster programs, at the cost of a larger code size. However, without any type annotations to guide the compilation, the resulting speedup was noticeably smaller than would be possible with a JIT compiler.

Benchmark	Lua	Pallene	Lua-AOT
Empty	0.1	2.7	3.7
Fannkuch	0.9	7.3	29.5
Fasta	1.8	10.1	43.4
Mandelbrot	0.9	4.7	24.5
Nbody	1.3	19.1	37.0
Spectral Norm	1.4	6.3	26.8
Queens	1.0	5.7	34.0
Sieve	1.5	10.2	30.1

Fig. 3: Size of the benchmark programs, in kilobytes. The Lua column refers to the size of the bytecode, and the Pallene and Lua-AOT columns refer to the size of the executable. The “Empty” row shows the resulting sizes when compiling an empty program containing no functions.

4 Adding types

Pallene is a typed variant of the Lua programming language (Gualandi & Ierusalimschy, 2020). Pallene modules are compiled to machine code, in the form of a Lua extension module. These extension modules can be loaded by Lua in a similar manner to how an extension module written in another static languages like C would be. The difference is that, as a typed variant of Lua, Pallene has been designed from the start to facilitate this interaction with Lua, and to reduce the run-time overhead of this interaction.

As an example of what a Pallene program looks like, consider the program in Figure 4, which implements the core part of the Nbody benchmark. Syntactically, this program is identical to its Lua equivalent, except for the declaration of the `Body` record type and the type annotations for function parameters and return types. Pallene infers the types for local variables, which rarely need type annotations. For example, `dx` is inferred to be of type `float`. It should be noted that in Pallene, the absence of a type annotation does not mean that the variable has the dynamic type `any`.

Semantically, Pallene programs behave the same as Lua, except that they may raise run-time type errors that Lua would have not. While we do not yet have a precise formalization of the semantics of Pallene’s run-time type checking, in general terms it is similar to the “transient” type checking of Reticulated Python (Vitousek *et al.*, 2014), or the first-order type checking approach described by Greenman and Felleisen (Greenman & Felleisen, 2018). The most important aspect is that type checks in Pallene always take the form of a constant-time type-tag check. For example, a single tag check can verify that a value is a function but not what kind of function it is. Type casts also do not introduce any wrappers or proxy objects. This is both for performance and to preserve object identity.

The Pallene compiler is careful to always check the types of dynamically typed values before they are used. It is committed to being as safe as Lua, in the sense that Pallene programs never segfault or access memory in a way they shouldn’t. Pallene checks the type of values that come from Lua arrays or Lua records. Function arguments are checked when Lua calls a Pallene function, and return types are checked when Pallene calls a Lua function. Type checking can also happen when Pallene calls another Pallene function, in

```
type Body = {
    x:  float, y:  float, z:  float,
    vx: float, vy: float, vz: float,
    mass: float
}

function update_speeds(bi: Body, bj: Body, dt: float)
    local dx = bi.x - bj.x
    local dy = bi.y - bj.y
    local dz = bi.z - bj.z

    local dist = math.sqrt(dx*dx + dy*dy + dz*dz)
    local mag = dt / (dist * dist * dist)

    local bjm = bj.mass * mag
    bi.vx = bi.vx - (dx * bjm)
    bi.vy = bi.vy - (dy * bjm)
    bi.vz = bi.vz - (dz * bjm)

    local bim = bi.mass * mag
    bj.vx = bj.vx + (dx * bim)
    bj.vy = bj.vy + (dy * bim)
    bj.vz = bj.vz + (dz * bim)
end

function update_position(bi: Body, dt: float)
    bi.x = bi.x + dt * bi.vx
    bi.y = bi.y + dt * bi.vy
    bi.z = bi.z + dt * bi.vz
end

function advance(nsteps: integer, bodies: {Body}, dt: float)
    local n = #bodies
    for _ = 1, nsteps do
        for i = 1, n do
            local bi = bodies[i]
            for j = i+1, n do
                local bj = bodies[j]
                update_speeds(bi, bj, dt)
            end
        end
        for i = 1, n do
            local bi = bodies[i]
            update_position(bi, dt)
        end
    end
end
```

Fig. 4: An example Pallene program, implementing the inner loop of the Nbody benchmark. The syntax is the same as the equivalent Lua program, but with added type annotations.

the context of higher-order functions, when calling an unknown function closure. When that happens, both the caller and the callee assume that the other side may be untyped.

Pallene was designed for performance, and one fundamental part of that is that its compiler generates efficient machine code. To simplify the implementation, and for portability, Pallene generates C source code instead of directly generating assembly language. This is similar to the approach that we used for Lua-AOT, except that Pallene generates more efficient, type-specialized code. For example, while Lua-AOT stores all variables in the virtual Lua stack, Pallene stores values with primitive types like integers in C local variables. This allows them to be stored in machine registers at run-time. Another optimization is that Pallene uses a more efficient calling convention when calling statically-known Pallene functions. The default Lua calling convention passes all arguments and return values on the Lua stack. Meanwhile, the optimized Pallene calling convention uses regular C function parameters and return values, which allows most of the arguments and return values to be passed via CPU registers.

To compare Pallene with Lua-AOT and LuaJIT, we evaluated the performance of Pallene using the same set of benchmarks. The Pallene programs were identical to the Lua ones, except for the type annotations. As before, the speed results are summarized in Figure 1 and Figure 2. Pallene achieved speedups between 1.5x and 12x, and was significantly faster than Lua-AOT. This reinforces the hypothesis that type-specific machine code is key for good performance. Pallene was slightly faster than LuaJIT in four of the benchmarks but noticeably slower in Nbody and Stream Sieve. Nbody involves frequent manipulation of Lua records, which are represented internally as hash tables. The performance difference with LuaJIT suggests that there may be further room for optimizing them in Pallene. The Stream Sieve benchmark features many small closures calling each other. This is a situation that trace-based JITs such as LuaJIT are well suited for, since they are able to effectively inline many of these function calls.

From the point of view of size, the generated code for Pallene was larger than the Lua bytecode by approximately $6\times$. However, it was smaller than the code for Lua-AOT by a factor of around $3\times$.

5 Program migration performance experiments for Pallene

We evaluated the performance of combining Lua and Pallene using the standard performance lattice analysis for gradual typing systems (Takikawa *et al.*, 2016). We adapted four benchmarks for this experiment: Spectral Norm, Nbody, Queens, and Stream Sieve. These were the benchmarks from the previous list which contained several functions, and which therefore could be split into separate modules. Each benchmark program was refactored to use two or three modules, each with a single function or a single group of mutually-recursive functions. We created two versions of each module. One untyped, in Lua, and one typed, in Pallene. We then analyzed the running time of all 2^N combinations of Lua and Pallene modules. The results are shown in Figure 5 and Figure 6. For each benchmark, there is a lattice of program configurations with pure Lua at the bottom, and pure Pallene at the top. Black ovals correspond to Pallene modules. White ovals are Lua, running on the reference interpreter. Gray ovals are also Lua, but using the Lua-AOT compiler. The

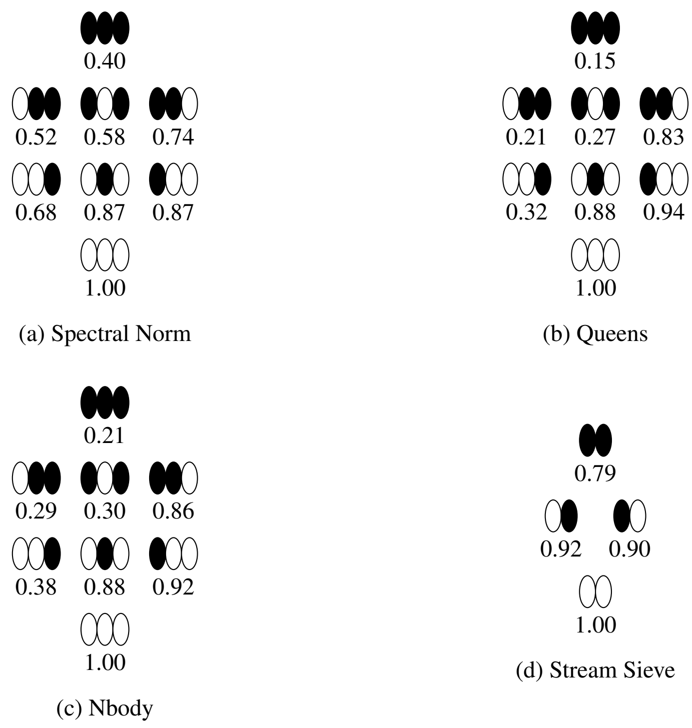

Fig. 5: Performance lattice for combinations of Pallene and interpreted Lua modules. Dark ovals represent (typed) Pallene modules and white ovals represent (untyped) Lua modules, running on the Lua 5.4 interpreter. The running times for each benchmark are normalized by the running time of the pure Lua configuration. Lower numbers are faster. In all configurations, moving from untyped to typed always speeds up the program and there are no pathological cases.

number shown under each configuration is the running time, normalized by the time of the pure Lua configuration. Lower numbers are faster.

To clarify, let's use the example of the Nbody benchmark. In preparation for this experiment, we refactored the original code in Figure 4 so that each of the three functions was put on a separate module. In the performance lattice shown in Figure 5(c), the three ovals correspond to `advance`, `update_position`, and `update_speeds`, respectively. For instance, in the configuration denoted by ●○○ the black oval means that the `advance` function is implemented in Pallene, and the two white ovals mean that both the `update_position` and `update_speeds` functions are implemented in Lua.

In the lattice of configurations we can move from one configuration to another by changing one module at a time from Lua to Pallene or vice-versa. For example, we can go from ○○○ to ●○○ by changing the implementation of the first module from Lua to Pallene. In this set of benchmarks, for every possible configuration, converting a module from Lua to Pallene never degraded the performance. Adding types always made the programs go faster, or at least had no effect on performance. When comparing Pallene

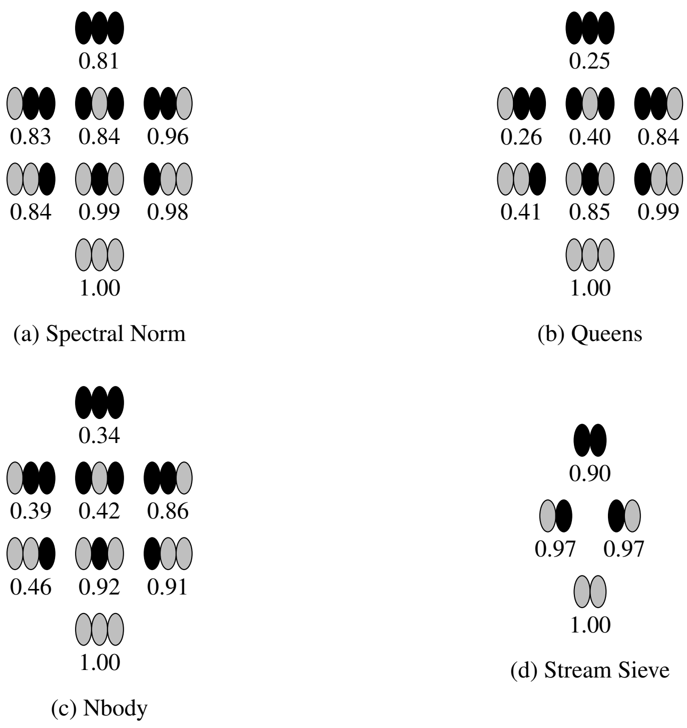

Fig. 6: Performance lattice for combinations of Pallene and Lua, this time with the Lua modules being compiled by Lua-AOT. Black ovals are Pallene, and gray ovals are Lua-AOT. As in Figure 5, lower numbers are faster. In some configurations, performance was unchanged by adding types. However, just as before, there were no configurations where adding types slowed down the program.

with interpreted Lua, the more typed configurations were always faster. Meanwhile, when comparing Pallene with Lua-AOT some transitions showed no performance benefit from adding types. This suggests that in those cases, the performance gain from Pallene was from avoiding the interpreter, not from the types. Nevertheless, even in this case there were no configurations where adding types caused the program to go slower. This departs from typical results for other gradually typed languages, where it is expected that some partially-typed configurations will be slower, sometimes by a large amount.

We should note that the speedups for the full-Pallene configurations in this version of the benchmarks was smaller than the speedups shown in Figure 2. Those versions of the benchmarks used a single module per benchmark, while the performance-lattice version used multiple modules. We believe that the main explanation for this performance difference is due to function calling conventions. Pallene uses the more efficient calling convention when directly calling other Pallene functions from the same module, and it uses the default Lua calling convention when calling functions from other modules. All benchmarks in this experiment intentionally have many cross-module function calls, exactly to test the communication and type checking overhead between Lua and Pallene. The performance of these cross-module calls may improve in a future version of Pallene, as we have plans to allow them to use the more efficient calling convention as well.

6 Focusing the design of Pallene

Our main goal for Pallene was to design a language with good performance, aimed at the Lua ecossystem. But we also had the goals of keeping the type system simple and of keeping the implementation portable and maintainable. Gradual typing helped us to achieve both of them.

Firstly, the types help optimize the programs. Furthermore, they allow the implementation to be a simple ahead-of-time compiler instead of a more complex JIT. There is no need to collect type information at run-time, and the implementation does not need to worry about deoptimizing after a failed type check, since failed type checks raise errors instead.

However, Pallene's good performance is only possible because of its specialized implementation, which is inherently more complex than the approach of using Lua as a target language. This led us to focus the design of Pallene around the features where the type annotations allow for improved performance.

In particular, Pallene has some restrictions about what operations are allowed for dynamically typed values (Gualandi & Ierusalimschy, 2020). Dynamically typed values of type `any` are first-class: they can be assigned to variables, passed as arguments to functions, etc. But other than being passed around, that only operation that is allowed on them is a downcast. For instance, they must be converted to a numeric type (integer or float) before being used in arithmetic, as exemplified in Figure 7. Similarly, they must be casted to a function type before they can be called as a function.

The motivation for this restriction comes from our experiments with Lua-AOT. For these dynamically-typed operations, Pallene would only be able to offer a modest performance improvement when compared with Lua. So instead of duplicating the implementation of these features in Pallene, we encourage the programmer to use Lua instead.

Pallene has some other minor restriction as well. For example, functions may not be redefined (monkey-patching). This is seldom used in the kind of programs that would benefit most from Pallene's performance.

The interplay between Lua and Pallene is fundamental. Our intention is that performance-sensitive modules may be written in Pallene, but the rest can remain in Lua. Pallene is therefore intended to play to the strengths of the classic scripting architecture (Ousterhout, 1998), with Lua playing the role of scripting language and Pallene playing the role of system language. From this point of view, Pallene's objective is not to supersede Lua, but to provide a more Lua-compatible alternative to other typed languages such as C or C++.

This brings us to the question of whether Pallene can be classified as a gradually typed language or not. One definition we can use for that would be the Refined Criteria for Gradual typing from Siek et al. (Siek *et al.*, 2015). That definition lists three criteria for determining if a language is gradually typed: 1) it should be a superset of both the typed and the dynamic language; 2) It should be as sound as the dynamic language, with no untrapped errors; 3) It should provide the gradual guarantee, which states that type annotations do not affect the evaluation of the program, except perhaps by introducing run-time type errors.

```
-- Not allowed!

function dynplus(x: any, y: any): any
    return x + y
end

function dyncall(f: any, x: any): any
    return f(x)
end
```

```
-- OK

function plus(x: any, y: any): any
    return ((x as integer) + (y as integer)) as any
end

function call(f: any, x: any): any
    return (f as any->any)(x)
end
```

Fig. 7: In Pallene, values of type `any` must be converted to a numeric type before they can be used in arithmetic. Similarly, dynamically typed values must be casted to a function type to be called as a function.

Pallene fits the second and third criteria, as we discuss in more detail in our previous work (Gualandi & Ierusalimschy, 2020), but it does not fit the first one because it is not a superset of Lua. However, Pallene is always intended to be used in combination with Lua and never as a standalone language. Viewed as a whole, the combination of both languages fits all three criteria, since the since the union of Lua plus Pallene is trivially a superset of Lua.

7 Conclusion

In this paper, we have shown that ahead-of-time compilation can be a pragmatic approach for producing an efficient gradually typed language, based on an existing dynamic language. We use the example of Pallene, a typed gradually-variant of Lua, designed with performance in mind.

To support our arguments, we have presented two experiments. The first was the evaluation of the impact of ahead-of-time compilation for untyped Lua programs. This evaluation found that picking the low-hanging fruit of ahead-of-time compilation for Lua provided a speedup of at most $2.5\times$. It hints that larger performance gains depend on being able to optimize based on type information.

The second experiment evaluated the type-checking of Pallene, by comparing the performance different combinations of Lua and Pallene modules. We found that in the case of Lua and Pallene, adding types usually improves performance and there were no instances where adding types degraded the performance.

Our analysis brings attention to three ideas. The first is that if the compiler uses types for optimization, the performance gains from adding types can outweigh the cost of the run-time type checking necessary to support those optimizations. When this happens, types are not a source of performance overhead, but quite the opposite. This is specially true if the types allow representing data in an unboxed form.

The second idea is that the optimizations performed by just-in-time compilers for dynamic languages are in many cases similar to the optimizations that an ahead-of-time compiler for a gradually typed language might be able to perform. This raises the question of whether efficient gradual typing should be possible not only for Lua, but also for other dynamic languages that are known to benefit from just-in-time compilation.

The final idea is that when performance is the main goal, the compiler for a gradually typed language does not necessarily need to implement a superset of its untyped counterpart. If the typed and the untyped languages are combined following the classic scripting approach, the type system and the implementation of the typed language can focus on the parts of the design that benefit the most from the added types.

Acknowledgments

This study was financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico— Brasil (CNPQ)—Finance Codes 153918/2015-2 and 305001/2017-5, and by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brasil (CAPES)—Finance Code 001.

Conflicts of Interest

The authors have no conflicts of interest.

References

Barany, Gergö. (2014). Python interpreter performance deconstructed. *Page 5:1–5:9 of: Proceedings of the Workshop on Dynamic Languages and Applications*. Dyla'14.

Bauman, Spenser, Bolz, Carl Friedrich, Hirschfeld, Robert, Kirilichev, Vasily, Pape, Tobias, Siek, Jeremy G., & Tobin-Hochstadt, Sam. 2015 (Aug.). Pycket: A tracing JIT for a functional language. *Page 22–34 of: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015.

Campora, John Peter, Chen, Sheng, & Walkingshaw, Eric. (2018). Casts and costs: Harmonizing safety and performance in gradual typing. *Proceedings of the ACM on programming languages*, **2**(ICFP), 98:1–98:30.

Futamura, Yoshihiko. (1999). Partial evaluation of computation process, revisited. *Higher order symbolic computation*, **12**(4), 377–380.

Gouy, Isaac. (2013). *The computer language benchmarks game*. `https://benchmarksgame-team.pages.debian.net/benchmarksgame`.

Greenman, Ben, & Felleisen, Matthias. (2018). A spectrum of type soundness and performance. *Proceedings of the ACM on programming languages*, **2**(ICFP), 71:1–71:32.

Gualandi, Hugo, & Ierusalimschy, Roberto. (2020). *Pallene: a companion language for Lua*. To appear in Science of Computer Programming. `http://www.inf.puc-rio.br/~hgualandi/papers/Gualandi-2020-SCP.pdf`.

Ierusalimschy, Roberto, de Figueiredo, Luiz Henrique, & Celes, Waldemar. (2005). The implementation of Lua 5.0. *Journal of universal computer science*, **11**(7), 1159–1176.

Kuhlenschmidt, Andre, Almahallawi, Deyaaeldeen, & Siek, Jeremy G. (2019). Toward efficient gradual typing for structural types via coercions. *Page 517–532 of: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019.

Muehlboeck, Fabian, & Tate, Ross. (2017). Sound gradual typing is nominally alive and well. *Proceedings of the ACM on programming languages*, **1**(OOPSLA), 56:1–56:30.

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

Pall, Mike. (2005). *LuaJIT, a just-in-time compiler for Lua*. `http://luajit.org/luajit.html`.

Richards, Gregor, Arteca, Ellen, & Turcotte, Alexi. (2017). The VM already knew that: Leveraging compile-time knowledge to optimize gradual typing. *Proceedings of the ACM on programming languages*, **1**(OOPSLA), 55:1–55:27.

Siek, Jeremy G., Vitousek, Michael M., Cimini, Matteo, & Boyland, John Tang. 2015 (May). Refined criteria for gradual typing. *Page 274–293 of: 1st Summit on Advances in Programming Languages*. SNAPL '2015.

Southern, Gabriel, & Renau, Jose. (2016). Overhead of deoptimization checks in the V8 JavaScript engine. *Page 1–10 of: Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE.

Takikawa, Asumu, Feltey, Daniel, Greenman, Ben, New, Max S., Vitek, Jan, & Felleisen, Matthias. (2016). Is sound gradual typing dead? *Page 456–468 of: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16.

The PyPy Project. (2011). *Call for donations - PyPy to support Python3!* `https://pypy.org/py3donate.html`.

Vitousek, Michael M., Kent, Andrew M., Siek, Jeremy G., & Baker, Jim. (2014). Design and evaluation of gradual typing for Python. *Page 45–56 of: Proceedings of the 10th ACM Symposium on Dynamic Languages*. DLS '14.

Vitousek, Michael M., Siek, Jeremy G., & Chaudhuri, Avik. (2019). Optimizing and evaluating transient gradual typing. *Page 28–41 of: Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2019.